2014 IEEE 33rd International Symposium on Reliable Distributed Systems

A Systematic Differential Analysis for Fast and Robust Detection of Software Aging

Rivalino Matias Jr.*, Artur Andrzejak†, Fumio Machida‡, Diego Elias*, Kishor Trivedi§
*Federal University of Uberlandia, UFU, Uberlandia, Brazil
†Heidelberg University, Heidelberg, Germany
‡NEC Knowledge Discovery Research Laboratories, Kawasaki, Japan
§Duke University, Durham, USA
rivalino@fc.ufu.br, artur@uni-hd.de, f-machida@ab.jp.nec.com, diegoelias@comp.ufu.br, ktrivedi@duke.edu

***Abstract*—Software systems running continuously for a long time often confront software aging, which is the phenomenon of progressive degradation of execution environment caused by latent software faults. Removal of such faults in software development process is a crucial issue for system reliability. A known major obstacle is typically the large latency to discover the existence of software aging. We propose a systematic approach to detect software aging which has shorter test time and higher accuracy compared to traditional aging detection via stress testing and trend detection. The approach is based on a differential analysis where a software version under test is compared against a previous version in terms of behavioral changes of resource metrics. A key instrument adopted is a divergence chart, which expresses time-dependent differences between two signals. Our experimental study focuses on memory-leak detection and evaluates divergence charts computed using multiple statistical techniques paired with application-level memory related metrics (RSS and Heap Usage). The results show that the proposed method achieves good performance for memory-leak detection in comparison to techniques widely adopted in previous works (e.g., linear regression, moving average and median).**

***Index Terms*—anomaly detection, memory leak, software aging**

I. Introduction

The software aging phenomenon has been observed in many software systems that run continuously for long time [15]. This phenomenon is often caused by software faults, so-called aging-related bugs, that leads to the accumulate errors during the software execution and causes undesirable consequences including performance degradation or system failures, after long time of execution [11]. A well-known example of software aging effects is the memory leaking, which is caused by software faults in the application memory management usage, and it leads to memory resource depletion in a system running uninterruptedly for a long period of time [20]. Memory leak is one of the most prevalent software aging problems reported in the literature, and thus the experimental study conducted in this paper is focused on it.

In the common software life cycle, complete removal of aging-related bugs in the development phase is very difficult and sometimes unfeasible. Although techniques to verify source code help to improve software reliability, they are not perfect in removing all aging-related faults, especially those related to memory leaks given that their aging effects only manifest during the run time and most of the time under particular workload conditions. Even when developers carry out system tests covering a large extent of the software operational profile, relatively short testing time is not enough to expose the potential aging effects caused by memory leaks [26]. Indeed, it has been observed that relatively short test durations increase the rate of false alarms of aging when using resource depletion metrics in conjunction with detection techniques based on trend analysis approaches, commonly adopted in the software aging literature [22]. It is a fundamental challenge to detect the existence of software aging in a short period of test time, especially when dealing with memory leaks.

In this paper, we propose a systematic approach to detect software aging with a shorter test duration and with high accuracy. Our approach is designed for a system test based on *differential software analysis* [18], [19], i.e. the comparison between a new software version under test and its previous stable version that has passed the test and thus is considered a robust version of the software. System metrics affected by the software aging effects under study, in this case memory leaks, are monitored periodically during the software test. Their observed values are then analyzed following a data analysis protocol that compares them with a baseline signal obtained by executing the robust (stable) version of the software. When the deviation of the target signal (obtained by monitoring the new version) from the baseline signal becomes significant, we suspect the existence of software aging effects in the new version under test. In order to quantify the deviation from the baseline signal, we introduce a *divergence chart*, in which the normalized difference between the target signal and an upper control limit of the baseline signal is plotted as a function of time (see Section III-D). Since the computation of the divergence chart is applicable to any type of signal from time-series data, the proposed approach can be combined with different data analysis techniques (e.g., linear regression, moving averages, and statistical process control techniques). By comparing various divergence charts against each other, we can identify the most effective technique to be used for the rapid detection of aging for a given scenario.

The contribution of this paper is twofold. The first contribution is the proposal of a systematic approach to detect software aging related to memory leaks during the software test phase.

1060-9857/14 $31.00 © 2014 IEEE
DOI 10.1109/SRDS.2014.38

IEEE computer society

Authorized licensed use limited to: Concordia University Library. Downloaded on May 14,2024 at 15:49:22 UTC from IEEE Xplore. Restrictions apply.

This approach combines differential software analysis with advanced statistical techniques for anomaly detection. We evaluate the effectiveness of our approach through controlled experiments, where various levels of workloads and aging rates are considered. The experiments show that it can detect aging more accurately and within shorter test durations than traditional approaches (e.g., [4], [8], [9], [21]) which are based on stress testing, regression models and trend detection techniques. The second contribution is the experimental study itself, focusing on memory leak detection. We compare the accuracy of using a heap-oriented metric against an application-oriented metric. So far, many previous researches have used these metrics, especially the latter, in an *adhoc* manner, without a more detailed investigation. Based on the results presented in [22], we further investigate the theoretical aspects of memory leak mechanism and show experimental results of memory leak detection.

The rest of the paper is organized as follows. Section II describes the mechanism of memory-related software aging from a theoretical viewpoint. Section III presents the proposed systematic approach to detect software aging caused by memory leaks using divergence charts. Section IV describes our experimental study focused on memory leak detection, and Section V presents the experimental results. Section VI discusses the related research, and finally Section VII states our conclusions.

II. Memory Leak Detection

As a representative example of software aging, in this paper we address the memory leak problem. Memory leak is the most investigated software aging issue because of its impact on system availability as well as the large number of occurrences found in real systems. In this section we describe two key aspects to be considered for any online memory leak detection approach, which are the system metrics to be monitored and the threshold values to reliably decide the existence of memory leak.

A. System metrics (aging indicators)

In order to detect memory leaks in a running program, application-specific metrics should be used instead of system-wide metrics [23], [20], [22]. RSS (resident set size), VSZ (virtual memory size), HSZ (heap size), and HUS (heap usage) are examples of such application-specific metrics in the Linux operating system. These are general metrics and similar ones can be found in different operating systems, although sometimes they are referred to by different terminologies (e.g., in Windows OS family the equivalent to RSS is named WS - working set).

The RSS is the working set size of the monitored process. It considers only parts of the process (text, data, stack, and heap) currently loaded in main memory. VSZ represents the total amount of virtual memory occupied by the process, including in-memory and on-disk pages. HSZ and HUS are, respectively, the total amount and the currently used amount of the process' heap, which include both in-memory and on-disk pages. It is not obvious which metric is the most important aging indicator among these for detecting memory leaks. Although previous approaches have used these application-specific metrics, there is no previous study comparing them in terms of their efficacy for aging detection. In order to decide which metrics to use in our experimental study (see Section IV), we compare their efficacy from both theoretical and practical viewpoints. In the rest of this section we present the theoretical analysis, and in Section V we present the results obtained through controlled experiments.

We compare in the following the applicability of each metric mentioned above; most of them have been used in the software aging literature. The RSS is not only related to the heap area, but also to other resident (in-memory) parts of the monitored process, such as text, data and stack. If the size of these other parts varies or increases significantly (more than the amount of leaks), they may dominate the RSS value and this could cause a significant number of false positive alarms. The HSZ is lesser noisy than RSS, because it is related only to the heap - where memory leaks take place. However, this variable does not tell us the complete story, since it cannot capture the amount of leaks inside the heap. Similar to RSS, the VSZ captures not only the heap but also other process parts. HUS captures only the memory used inside the heap; differently from HSZ, in this case only allocated blocks are counted (not free space). However, HUS does not distinguish from usable (active/inactive) or unusable (leaked) allocated blocks. Of course, the best choice would be measuring only the leaked blocks, which is completely noise free. However, it is currently not practical to monitor for majority of real applications, due to the high overhead it could cause at run time.

From this investigation, our approach considers the use of HUS as a less noisy and standard system metric that may be used in different OS platforms. We submit that it can capture the amount of memory leak more efficiently than other widely adopted aging indicators (or system metrics) can. Since the RSS is currently extensively used in experimental research in this area, in Section V-D we compare the results obtained using both RSS and HUS.

B. Threshold on memory leak detection

In addition to selecting the adequate system metrics to be monitored, it is also necessary to specify how their values are used to detect the presence of memory leaks. Many previous papers (e.g., [10]) have adopted a trend-based approach for this purpose. In general, after monitoring the selected system metric(s) for certain period of time they apply trend analysis techniques (e.g., Mann-Kendall tests, Sen's slope, regression analysis) to discover increasing trends in memory usage. However the trend detection itself does not ensure the existence of memory leaks, since the size of the process' heap may be increasing not only because memory leaks exist, but also due to the application's memory usage pattern [22]. Therefore, trend detection itself is not enough to support a safe decision for this problem, even if it is calculated with high statistical confidence.

Authorized licensed use limited to: Concordia University Library. Downloaded on May 14,2024 at 15:49:22 UTC from IEEE Xplore. Restrictions apply.

Moreover, some studies (e.g., [5], [10]) have assumed that while applying a constant workload, observing a consistent increase of the process size, implies the existence of memory leaks. The fact of using a constant workload is not a guarantee of a non-increasing usage of memory in a leak-free application. An increasing of heap size could be the consequence of new memory pages being added to the heap pool, and not necessarily due to memory leaks, which can be observed with both varying and constant workloads.

Thus, in order to detect the presence of memory leaks in a target application, we need to distinguish the monotonicity property of the memory leaks from the natural increase of the application's heap size. Hence, we need to compare the heap usage pattern of a target application against a baseline pattern, which we know is representative of the expected natural increase in the application's heap size. The monitored target application is a new version of the software being tested, and the comparative baseline is obtained through monitoring the latest stable version of the software, which is considered a robust version that has been cleared in previous tests and thus accepted to be deployed in production.

In order to implement the above-mentioned procedure, we advocate that, among the most used metrics so far, HUS is the most effective, since it is the lesser noisy variable. Based on monitoring this metric, we can capture the heap usage pattern of the target application and compare it against the baseline pattern. To this end, a comprehensive comparative analysis is required. In the next section, we propose a systematic approach to carry out system tests in order to detect software aging caused by memory leaks, based on a judicious comparative method.

III. Systematic Approach to Aging Detection

In this section, we present our general approach to detect software aging in a target software. The approach consists of three steps: *i*) measurements from a target software version under test, *ii*) processing of the collected data for statistical analysis, and *iii*) detecting unexpected resource usage patterns. The third step is based on the comparison of the collected data from step *i*, against baseline data obtained from a previous stable version of the software under test. Since our approach is independent of system metrics and data processing techniques, it is a robust software aging detection that can be applied to several combinations of metrics and techniques in a systematic way.

A. Overview

Figure 1 shows an overview of the steps and instrumentation used in our proposal. The software version under test (SVUT) is installed on a given execution environment, in which the selected test workload is assigned by the aging test controller. In the first step, the values of selected system metrics are collected periodically by the monitoring instrumentation. The collected data is organized as one time series per monitored metric.

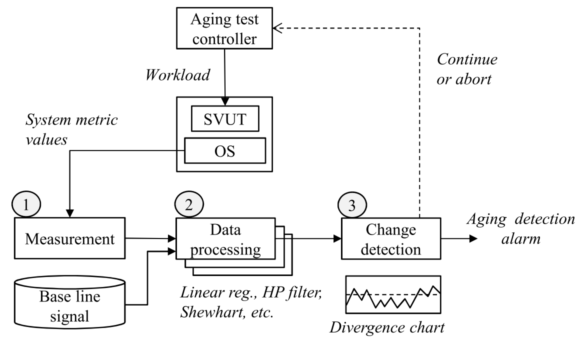

Figure 1. Aging detection workflow

In the second step, the obtained time series are used to compose target signals by means of different data processing techniques such as moving average, moving median, Hodrick-Prescott filter, and so on. The data analysis can be performed in parallel among different combinations of metrics and processing techniques. In the third step, the obtained target signal is compared against the baseline signal for detecting abnormal divergence between the two signals. For this purpose, a *divergence chart* is computed and used to detect significant changes in the SVUT resource usage pattern; e.g., significant and consistent memory usage increase in comparison with the baseline. Since the computation of the divergence chart is independent of specific data processing techniques, the results of change detections are comparable to each other and thus we can make more reliable decisions within a short test time. The following subsections detail each step of this approach.

B. Measurements

Measurements are conducted for metrics that potentially indicate software aging effects on the software under test. As discussed in Section II-A, selecting appropriate system metrics is important for aging detection. Especially for memory leak detection, RSS and HUS are relevant metrics among other system-level metrics (e.g., free memory and swap space). In the Linux operating system, several application-specific metrics, such as RSS, are available by simply accessing /proc file system or using system programs like *vmstat*, and *ps*. For more specific metrics, such as HUS, that require the application's heap information, it is necessary to use proper monitoring tools such as DTrace (Solaris) [7], Detours (MS Windows) [16], and SystemTap (Linux) [17]. Periodic monitoring of these system variables can be automated by regular shell script programming along with proper instrumentation. The collected values are organized as time series that are used for generating the target signals by data processing techniques.

C. Data processing

The performance of the aging detection are affected mainly by factors such as the monitored metrics and data processing techniques adopted. Different systems and types of aging effects would require specific data processing techniques to

Authorized licensed use limited to: Concordia University Library. Downloaded on May 14,2024 at 15:49:22 UTC from IEEE Xplore. Restrictions apply.

Table I
SYMBOLS OF METRICS AND PROCESSING TECHNIQUES

Symbol	Type	Meaning
RSS	Metric	Resident set size
HUS	Metric	Heap usage
LR	Trend model	Rolling linear regression
MA	Trend model	Moving average
MM	Trend model	Moving median
HP	Trend model	Hodrick-Prescott filter
SH	SPC technique	Shewhart control chart
EW	SPC technique	Exponentially weighted moving average
CS	SPC technique	Cumulative sum (CuSum)

be applied for the time series observed in the target system. Therefore, our approach is general enough to support the use of many data analysis techniques concurrently in a common set up.

In this paper, we cover four different statistical methods for trend analysis, which are rolling linear regression (LR), moving average (MA), moving median (MM), and Hodrick-Prescott filter (HP). Moreover, we use three statistical process control (SPC) methods, namely Shewhart control chart (SH), exponentially weighted moving average (EW), and cumulative sum (CS). Table I summarizes the abbreviated terms we use hereafter.

The first four methods in Table I are mainly used for smoothing and fitting a potential trend. Linear regression (LR) is the most commonly used approach to model the relationship between two variables by fitting a linear equation to observed data. Here we use a rolling linear regression, where line fit is made over data in a sliding window. Moving average (MA) [13] is applied to time series data to smooth out short-term fluctuations and highlight longer-term trends or cycles. If the time series data contains outliers or surges, moving median (MM) [13] gives a more robust estimate of the trend. When the data is assumed to consist of trend and cycles, the Hodrick-Prescott filter [14] is useful to extract the trend from time series containing cycle components. The effectiveness of HP filter for software aging detection was studied in [31] as well.

The last three entries in Table I are SPC techniques. To the best of our knowledge, this is the first paper that evaluates SPC techniques applied to software aging detection. We studied and selected three SPC control chart techniques [27] to use in this paper: Shewhart control charts (SH), Exponentially weighted moving average (EW), and Cumulative sum (CS). SH charts are appropriate to capture large shifts in the process mean ($\geq 1.5\sigma$), while CS is suitable to detect small shifts ($< 1.5\sigma$). EW is able to detect both small and medium to large shifts [27]. In addition to covering all our needs in terms of shift size detection, the three above mentioned techniques are mature and well tested, making their implementation for automatic aging detection very appropriate. We adjust the parameter values for the three SPC techniques as follows. For CS and EW, we set their parameters to have comparable 3-sigma Shewhart's (SH) control limits. In a SH control chart with 3-sigma control limits, there is approximately a 0.27% probability of a value falling outside of the control limits under normal behavior. In this case, false alarms are expected to occur on average once every 370.37 ($1/0.0027$) observations. Based on the algorithms described in [27] and [29], we found this parametrization relationship and used in our experimental study. The parameter values for each SPC model adopted are: SH ($3\sigma, d_2 = 1.128$), CS ($k = 0.5, h = 4.77$), and EW ($L = 2.701, \lambda = 0.1$). For SH, the monitored time series is directly used as the target signal. The target signal for EW is obtained by computing the exponential moving average of the original data, while the target signal for CS is generated by the CuSum algorithm using the mean and standard deviation of the baseline signal [27].

D. Change detection

Change detection attempts to find significant difference between baseline signal and target signal from current SVUT; both signals are created with the same metric and data processing technique. For this purpose, we first compute a series of *divergence values*, where each value relates to a normalized difference between the target and baseline signals, at the same sampling rate, from the test start time.

1) Computation of divergence values: To obtain the divergence values we compute, from the baseline and target signals, three derived time series: target signal $\{f_t\}$, lower $\{L_t\}$ and upper $\{U_t\}$ bounds of a control limit interval. These bounds indicate the range of the filtered target signal. The computation of series $\{f_t\}$, $\{L_t\}$, and $\{U_t\}$ depends on the individual technique adopted (explained below). For a fixed time t (i.e., corresponding elements of the three series), the divergence value is computed by

$$\left(\frac{f_t - U_t}{U_t - L_t}\right)^+ \qquad (1)$$

where X^+ is X if $X \geq 0$ or 0 otherwise. Evidently, if the f_t is within the limit interval (or below), we get 0, otherwise the distance of f_t from the upper bound in units being the width of the control interval. The latter property is the normalization allowing uniform thresholds for all techniques.

For the trend-based detection techniques (LR, MA, MM, HP - see Table I), the target signal $\{f_t\}$ is computed by applying the respective smoothing function to the monitored time series, e.g., moving average to RSS or HUS. The bounds of the control limits are obtained by computing rolling standard deviation of the baseline signal, and adding it (for $\{U_t\}$) or subtracting (for $\{L_t\}$) from the smoothed base signal (smoothed by the same technique as the target signal).

For the SPC techniques (SH, EW and CS) the approach is a bit more complex. In the case of CS, $\{f_t\}$ is obtained by applying the CuSum algorithm to a series obtained by Z-normalizing the observed time series with the mean and standard deviation of the baseline signal. The control interval in this case is constant $[-4.77\sigma, 4.77\sigma]$.

In the case of SH, $\{f_t\}$ is the target signal itself, and the upper/lower control bounds are obtained from the baseline signal. For this purpose we add to the mean of the baseline signal (for $\{U_t\}$) or subtracts from it (for $\{L_t\}$) the term

Authorized licensed use limited to: Concordia University Library. Downloaded on May 14,2024 at 15:49:22 UTC from IEEE Xplore. Restrictions apply.

Term	Definition
Divergence Event	Event f_{i+5} of subseries $f_i, \ldots, f_{i+5}$, where $f_i < \alpha$ and all $f_{i+1}, \ldots, f_{i+5} \geq \alpha$
DivFirstTime	Time of occurrence of first divergence event
DivLastTime	Time of occurrence of last divergence event
DivNumEvents	Number of divergence events after *DivFirstTime*

Table II
TERMS AND METRICS ASSOCIATED WITH DIVERGENCE SERIES

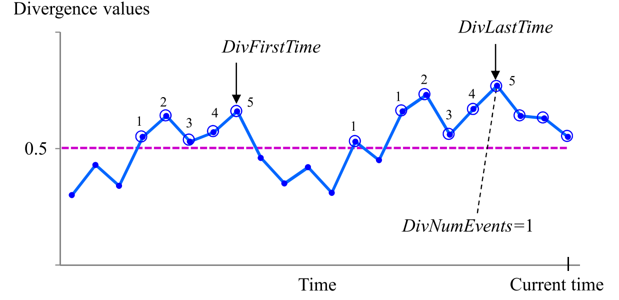

Figure 2. A divergence chart and associated metrics. Both events labeled with "5" are divergence events. There is just one such event after time *DivFirstTime* (namely at *DivLastTime*), and so *DivNumEvents* = 1.

$3MR/d_2$, where d_2 value is shown in Section III-C, and MR is the average of moving ranges, MR_i, given by

$$MR = \frac{1}{m}\sum_{i=1}^{m} MR_i \quad \text{with} \quad MR_i = |f_i - f_{i-1}|\,. \qquad (2)$$

For EW, the target signal is obtained by the exponential moving average of the original time series, and the upper/lower bounds are computed by

$$U_t = \mu_0 + L\sigma\left(\frac{\lambda}{(2-\lambda)}\left[1-(1-\lambda)^{2t}\right]\right)^{0.5} \qquad (3)$$

$$L_t = \mu_0 - L\sigma\left(\frac{\lambda}{(2-\lambda)}\left[1-(1-\lambda)^{2t}\right]\right)^{0.5}, \qquad (4)$$

where $0 \leq \lambda \leq 1$ is a constant, L is the width of the control limits in number of standard deviations.

2) Divergence chart: Our aging detection approach is carried out according to the following protocol. Assume that the current divergence value f_i is below a given threshold value α. When a sequence of five consecutive values, f_{i+1} to f_{i+5}, are above or equals the threshold α, we consider that a *divergence event* has occurred. We associate this event with the sampling time of f_{i+5}. In other words, in a sequence of six consecutive divergence values, if the first one is below a threshold, and all others are above it, the last one is flagged as a divergence event. In Figure 2, both samples labeled with "5" are divergence events. The series of divergence values and the threshold constitute the *divergence chart*.

The choice of five consecutive values came from the SPC theory, based on the probability of false alarms for a given control limit range. In preliminary analysis we observed that for higher values (e.g., 10) the quality of the results does not change significantly. Due to normalization of divergence values, we can use a common threshold for all metric/technique combinations. In our experimental study we set it to 0.5, given that this value was found to be suitable by preliminary experiments, resulting in a good balance between detection latency and robustness.

To evaluate and compare different metrics/techniques combinations, we introduce the following concepts. The time of the first divergence event is called *DivFirstTime*. Essentially, this is the earliest indication of aging effects and its value is dependent on the latency of a combination (technique and metric). Note that the corresponding event may be a false alarm, and more divergence events could occur afterwords.

We estimate the robustness of a metric/technique combination by *DivNumEvents* that is the number of divergence events after *DivFirstTime*. *DivNumEvents* obviously quantifies the number of false alarms. Since software aging progresses over time, the occurrences of false alarms are expected to disappear after a certain amount of test time. This gives rise to another metric: *DivLastTime*, which is the time of the last divergence event. Thus, an ideal metric/technique combination would have *DivNumEvents* = 0 and *DivFirstTime* = *DivLastTime*. The higher is the spread between *DivFirstTime* vs. *DivLastTime* and/or larger *DivNumEvents*, the less robust is the given combination. Table II summarizes the introduced terms.

Note that values of *DivLastTime* and *DivNumEvents* are available only after a sufficiently long minimal test execution time. This minimal test time must be fixed in advance depending on the system under test (see Section IV-A). Terminating test execution earlier and adaptively (e.g. after first occurrence of *DivFirstTime* or a certain number of *DivNumEvents*) can potentially further shorten the detection time. However, adaptive test termination is inherently less robust and is not considered in this work.

IV. EXPERIMENTAL STUDY

We implement the proposed approach and conduct experimental studies to evaluate the effectiveness of our proposal on the memory-leak aging detection in a Linux operating system. This section describes the experimental setup and configurations used in this evaluation.

A. Experimental setup

Our test bed is composed of one virtual machine with three processors (*vcpus*) and 5 GB RAM. The hypervisor is Xen 4.1.1 and the guest operating system is Linux OpenSuSe 12.2 (kernel 3.4.6-2.10). All experiments are carried out with the Linux OS configured in runlevel 3. Our experimental plan considered three main controlled factors: workload intensity, leak rate, and workload type. These factors are controlled inside of a synthetic workload generator (SWG) we created for this purpose. The workload intensity is controlled by varying the size of memory blocks allocated by SWG. We consider

Authorized licensed use limited to: Concordia University Library. Downloaded on May 14,2024 at 15:49:22 UTC from IEEE Xplore. Restrictions apply.

Algorithm 1 Workload load generator algorithm

```
SWG (p, w, th)
  p: percentage of leak
  w: workload intensity
  st: thread status
  th: rate of thread creation
  rt: run time of application
  loop
    st = thread_create ( Load(p, w) );
    if (st != ZERO) then break;
    if (th == CONSTANT) then
       sleeps for 500000 microseconds;
    if (th == VARYING) then
       if (rt ≤ 30min.) then
          sleeps for 500000 microseconds;
       else if (rt ≤ 60min.) then
          sleeps for 250000 microseconds;
       else if (rt ≤ 90min.) then
          sleeps for 166666 microseconds;
          if (rt ≥ 90min.) then rt = 0;

Function Load (p, w)
  t: sleep time in seconds
  k: leak activation factor
  c: allocated memory address
  t = random (1..30);
  if (w == LOW) then
     c = malloc (32 * random (1..16));
  if (w == NORMAL) then
     c = malloc (512 * random (1..55));
  if (w == HIGH) then
     c = malloc (1024 * random (1..200));
  if (c == NULL) then thread_exit;
  for each position in c
     c[position] = 0;
  sleeps for t seconds;
  k = random(1..100);
  if ( (p == 0.0) or
        (k ≤ (100 - p)) ) then
     free (c);
  thread_exit;
```

three levels of workload intensity, namely low, normal, and high. The allocation block sizes varied randomly according to an uniform distribution, ranging from 32 to 512 bytes (for low workload), 512 to 28,160 bytes (for normal workload), and 1024 to 204,800 bytes (for high workload), respectively. We use four percentage values (0%, 0.1%, 0.5% and 1%) to express the memory leak rates, where 0% represents a leak-free execution. For all other leaking rates, previously allocated blocks are unreleased according to the respective percentages. The workload type, categorized into constant and varying, controls the variations of the interval of memory allocation requests. While memory allocations are requested in a regular interval by constant workload, they have different intervals in the varying workload (see Algorithm 1).

Algorithm 1 shows the algorithm implemented in SWG. Given that multi-threading is widely adopted in today's applications, and it has a significant impact on the user-level memory allocator [3], the SWG was designed to create multiple threads that allocate and release memory blocks simultaneously. The thread life cycle follows five steps: *i*) according to the workload intensity (low, normal, high), it allocates a specific size of memory block; *ii*) fills out the allocated memory to make sure it will be used; *iii*) wait a random time (1 to 30 seconds); *iv*) release the allocated block with leaking according to the selected leak rate; *v*) thread ends. Note that in step (*iv*), the leak injection is implemented based on random numbers, drawn from an uniform distribution, U(1,100), where the allocated memory is released only if the percentage of leak is equal to zero or when the random number is greater than the percentage for the selected leak rate.

When the workload type is constant, the SWG creates two threads per second. In case of the varying workload, the number of threads increases in a cyclical way, starting in two threads per second and ending with six threads per second after two successive increases, then the cycle restarts. The entropy caused by the simultaneous thread executions depends on the other parameters, such as the allocation block size and the leak rate. Also, the life time of a thread is governed for random values, as observed in real-world applications.

For each test we monitored 66 system metrics in total, every 5 seconds. Based on preliminary analyses, applying machine-learning algorithms (C4.5 decision tree algorithm) to all of the monitored metrics, we found that RSS (resident set size) and HUS (heap space usage) provided the stronger correlation to the different percentages of leaking investigated. Hence, based on these practical results and the theoretical analysis presented in Section II-A, the rest of this paper presents only the relevant results obtained with these two metrics, RSS and HUS.

In order to have a sample size large enough to observe the aging effects, each experiment lasted four hours. Our experimental plan is based on a factorial design [27], so we evaluate the effect of changes applied to each of the three above-mentioned factors, in order to measure their effects on the application's memory usage. The baseline signal for experiments is based on the time series obtained through all executions with 0% leak rate, while we assume the baseline is obtained through the system test on the previous robust version.

V. Result Analysis

In this section, we present a performance comparison of the seven techniques (LR, MA, MM, HP, SH, CS, EW, see Table I) used in our experimental study.

A. Exemplary comparison of divergence charts

Figure 3 compares divergence charts for a single test scenario (load=high; leak rate=0.1%), considering both constant and varying workloads. For this test case, it can be seen that SPC techniques (SH, EW, and CS) detect aging faster and with higher stability, where CS outperforms all the other techniques. Also, in varying workloads the robustness of all techniques suffers in comparison with their performance for the respective tests based on constant workload.

Figure 4 provides a better grasp on the robustness of the techniques by comparing *DivFirstTime* and *DivLastTime* (same data as in Figure 3). Round markers indicate that both times were equal (i.e. no false alarms), otherwise bars stretch between both times. The results corroborate the observed in the

Authorized licensed use limited to: Concordia University Library. Downloaded on May 14,2024 at 15:49:22 UTC from IEEE Xplore. Restrictions apply.

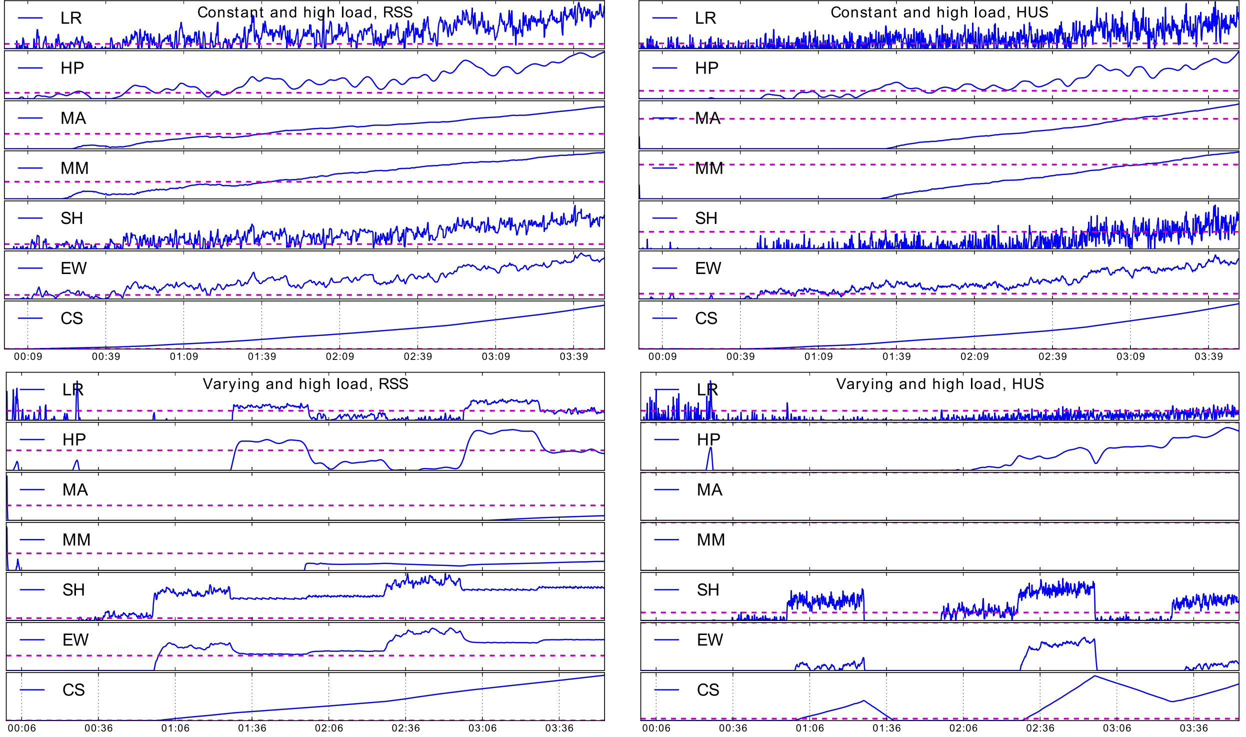

Figure 3. Divergence charts for RSS (left) and HUS (right) under constant (upper) and varying (lower) workloads (all: high load, leak rate 0.1%)

divergence charts, where detection times obtained in constant workload are better, and also here SPC techniques show better performance.

The analysis exemplified above was performed for each of the 36 test scenarios we evaluated.

B. Technique ranking

Based on *DivFirstTime* and *DivLastTime* defined in Section III-D, we rank the techniques for both constant and varying workload scenarios. The best techniques are the top-most in the rank, which are ascending sorted based on their *DivLastTime* and accorded to the following rule: If the *DivNumEvents*=0, then *DivLastTime*=*DivFirstTime*. The rationale is that techniques with lesser *DivLastTime* show faster stability. If competing techniques show the same *DivLastTime*, then the *DivNumEvents* is used as a second sorting criterion (lower *DivNumEvents* is better since *DivNumEvents* counts the number of false alarms after *DivFirstTime*). For example, for techniques T1 (*DivFirstTime*=2h, *DivLastTime*=30h, *DivNumEvents*=3) and T2 (*DivFirstTime*=10h, *DivLastTime*=15h, *DivNumEvents*=5), our comparison protocol considers T2 better than T1, even T2 presenting higher values for *DivFirstTime* and *DivNumEvents*. The reasoning is that T2 is faster to produce reliable results than T1, which occurs after 15 hours of test in T1 against 30 hours in T2.

Based on this comparison protocol, we analyze in details the three top-most techniques in regard to the rank, for each test scenario in the both constant and varying workloads. The rest of this section presents these analyzes, with ranks given in Table III.

We can observe that CS appears, among the three top-ranked techniques, in 41% of the tests for constant workload, followed by EW and HP (see Table III). Note that for the same test there is more than one occurrence of the same technique (e.g., both CS (RSS) and CS (HUS) appeared in normal workload with 0.5% leak rate). This happens because we apply each technique twice, using both RSS and HUS. The techniques not listed are not present among the three best positions. For varying workload (see Table III), CS and HP again showed the best results, where both CS and HP are present in 26% of the tests, followed by SH in 22%. Based on these results, we conclude that CS and HP, followed by SH and EW, are the best techniques for the evaluated scenarios.

C. Aging Detection time

Next we compare the best detection times obtained in all tests in both experiments (constant and varying workloads). The results are shown in Figure 5. As expected, in general, the best detection times for constant workloads are lower than in varying workloads, except for one case (varying load=normal; leak rate=0.1%). We investigated this specific case and found no special reason for this difference, so we consider this as an outlier. For the worst case scenario (load=low; leak rate=0.1%) in varying workload the detection time is approximately one hour, and for constant workload sixteen minutes. These results are very promising when compared to related test times reported in the literature (e.g., 2.5 hours [8]). Also, for each

Authorized licensed use limited to: Concordia University Library. Downloaded on May 14,2024 at 15:49:22 UTC from IEEE Xplore. Restrictions apply.

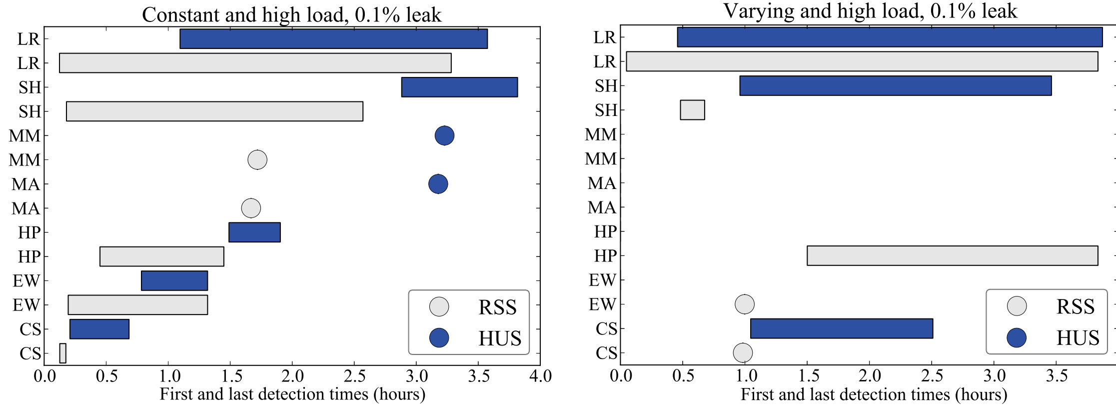

Figure 4. First (DivFirstTime) and last (DivLastTime) detection times for constant (left) and varying (right) workloads (all: high load, leak rate 0.1%)

Table III
THE BEST TECHNIQUES AND METRICS OBSERVED FOR THE CONSTANT AND VARYING WORKLOAD TESTS

		Constant workload			Varying workload		
		1st	2nd	3rd	1st	2nd	3rd
Low	0.1%	CS (HUS)	EW (HUS)	HP (HUS)	CS (HUS)	EW (RSS)	SH (RSS)
	0.5%	MM (RSS)	CS (HUS)	EW (HUS)	LR (RSS)	CS (HUS)	HP (HUS)
	1.0%	CS (HUS)	HP (HUS)	EW (HUS)	HP (HUS)	CS (HUS)	SH (HUS)
Normal	0.1%	CS (HUS)	EW (HUS)	HP (HUS)	MA (RSS)	MM (RSS)	HP (RSS)
	0.5%	CS (RSS)	CS (HUS)	EW (RSS)	CS (HUS)	SH (RSS)	CS (RSS)
	1.0%	CS (HUS)	HP (RSS)	EW (HUS)	HP (HUS)	LR (HUS)	SH (HUS)
High	0.1%	CS (RSS)	CS (HUS)	EW (HUS)	SH (RSS)	CS (RSS)	EW (RSS)
	0.5%	CS (HUS)	HP (RSS)	SH (RSS)	SH (RSS)	CS (RSS)	HP (HUS)
	1.0%	CS (HUS)	EW (RSS)	EW (HUS)	HP (RSS)	HP (HUS)	LR(HUS)

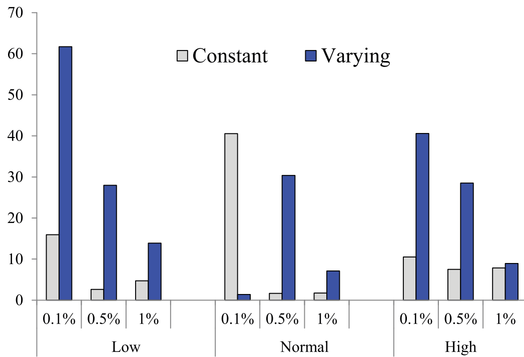

Figure 5. Best detection times per test (y-axis in minutes)

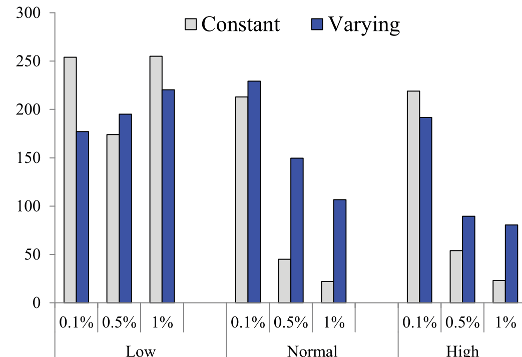

Figure 6. Difference of best and worst detection times (y-axis in minutes)

test we calculate the difference between the best and worst detection times (see Figure 6). We can see significant differences, especially for varying low workloads, which means the best techniques identified by the divergence charts show higher efficiency for these cases. Particularly, varying low workloads are the hardest scenarios to fast detecting the investigated aging effects, because the memory leaks accumulate slowly at lower workload levels.

D. Comparison of HUS vs. RSS

In order to compare the efficacy of the selected system metrics, HUS and RSS, we firstly computed how many times, per test case, these variables participate in the best detection times for the ten first techniques in the rank (see Section V-B). The results are summarized in Figures 7 and 8. For the low constant workload, HUS shows better in all the tests. On the other hand, RSS is better for normal load with 1% of leaking and all the tests with high constant workload. A possible explanation is that higher workloads make less important the sensibility advantage provided by HUS. This occurs because HUS captures the effects of memory leaks some time before they reflect on the RSS (see Section II). If these effects accumulate fast (higher workloads), then this advantage may

Authorized licensed use limited to: Concordia University Library. Downloaded on May 14,2024 at 15:49:22 UTC from IEEE Xplore. Restrictions apply.

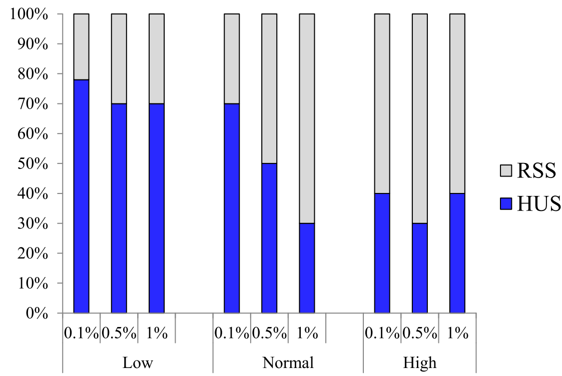

Figure 7. HUS vs. RSS for constant workload

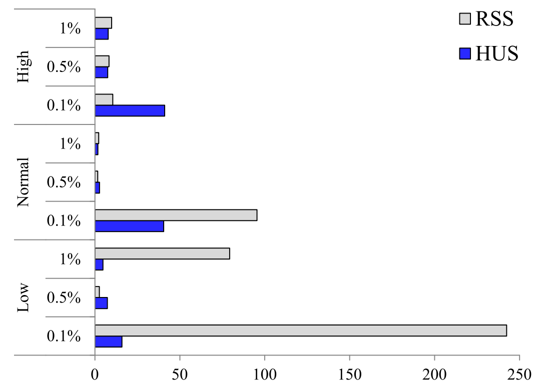

Figure 9. Best detection times for constant workload (x-axis in minutes)

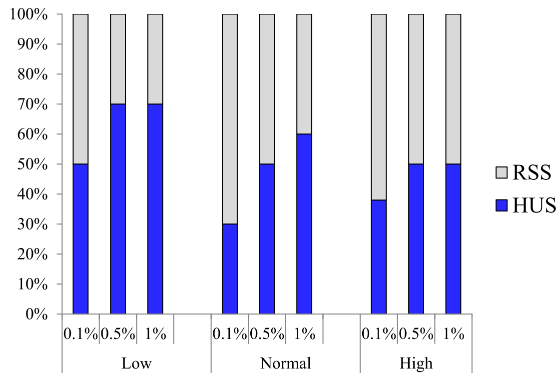

Figure 8. HUS vs. RSS for varying workload

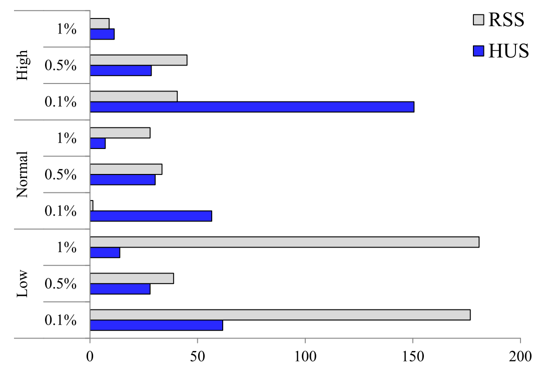

Figure 10. Best detection times for varying workload (x-axis in minutes)

disappear. However, we know from the literature (e.g., [28]), that most residual memory leak problems, undetected during the test phase, are exactly the ones with lower probability of occurrences, where HUS demonstrates the better results. For the tests in the varying workload experiment (see Figure 8), we have more balanced results, with HUS also demonstrating better for low workload scenarios.

Next, we compare the best detection times obtained with both RSS and HUS. Figures 9 and 10 show the results for constant and varying workloads, respectively. We observed that HUS contributes for the shorter detection times more than RSS. The experimental results corroborate our theoretical discussion presented in Section II.

VI. Related Work

Techniques to mitigate software aging effects are broadly classified into two categories by the phase of software life cycle, namely development phase and operational phase. Due to the difficulty of complete removal of aging-related bugs in the development phase, techniques to counteract software aging during operational phase are useful and widely investigated.

Software rejuvenation [15], [2] is a recognized approach to mitigate aging effects which works by resetting or restarting the application or execution environment when aging effects are observed. According to [1], this method is currently widely adopted in many software systems e.g in telecommunication systems [4], server cluster managers [9], and various others.

An alternative countermeasure is *software life-extension*. It attempts to prolong the lifetime of software execution [21]. When users can locate the cause of aging-related bugs in the operational phase (and they are not contained in a third party library or a commercial software product), hot-fixes also can be considered as a measure to eliminate aging [28].

In this paper, we focus on the approach to detect and eliminate software aging in software development phase, especially in system test. The approach to characterize software aging by system test is not new, while most of existing aging tests are intended to be applied in operational phase (i.e., after software release). Matias et al. introduce a design of experiments to characterize software aging in web servers, under different workloads [24]. Controlled experiments such as accelerated life tests [26] and accelerated degradation tests [25], [30] have also been proposed and applied to reduce the aging detection time in operational phase. Another proposal to reduce the test time for detecting software aging using statistical tests based on trend-slope estimates is presented in [8]. Bovenzi et al. present a general procedure to characterize the aging-workload relationship among different applications [6]. The impact of

Authorized licensed use limited to: Concordia University Library. Downloaded on May 14,2024 at 15:49:22 UTC from IEEE Xplore. Restrictions apply.

software test on the operational behavior of software systems suffering from aging is evaluated in [12]. In the same context, our study also focuses on reducing the aging detection time, however not only based on a single detection technique but combining several methods in a judicious way.

The comparison-based approach outlined in Section III-A was first introduced in [18] and preliminarily evaluated there for non-automated aging detection via visual comparison of raw system metrics. In [19] software version comparison is used for identifying leaking memory allocation sites in Java programs. Contrary to [18], in this paper we apply advanced statistical processing techniques in combination with divergence charts in order to achieve a more general framework for accurate and automated detection of aging.

VII. Conclusions

In this work, we present a systematic approach to detect software aging by divergence analysis based on comparison with previous version of the same software. Our approach is general enough to support various signal-processing techniques combined with different numbers and types of monitored system metrics (aging indicators). In addition to aging detection, the introduced divergence charts allow test engineers to conduct visual analysis, helping them to make more educated decisions in regard to selection of candidate techniques and system metrics.

The SPC techniques (CS, SH, EW) and HP filter show the best and most robust results, with CS and HP excelling among them, on average. Another important finding is related to the RSS and HUS comparisons. Since HUS yields better results for all scenarios with low rate of leak manifestation, we recommend this metric specially for test plans where the software under test presents low rates of memory-leak related failures.

VIII. Acknowledgments

This work is supported in part by grant AN 405/2-1 entitled *Automated, minimal-invasive Identification and Elimination of Defects in Complex Software Systems* financed by the Deutsche Forschungsgemeinschaft (DFG), and Brazilian Research Agencies CNPq, CAPES, and FAPEMIG.

References

[1] J. Alonso, A. Bovenzi, J. Li, Y. Wang, S. Russo, and Kishor Trivedi, Software rejuvenation - do IT & Telco Industries use it?, Proc. 4th Int'l Workshop on Software Aging and Rejuvenation (WoSAR), 2012.

[2] A. Andrzejak, L. Silva, Using Machine Learning for Non-Intrusive Modeling and Prediction of Software Aging, Proc. 11th IEEE/IFIP Network Operations and Management Symp. (NOMS), 2008.

[3] J. Attardi, and N. Nadgir, A Comparison of Memory Allocators in Multiprocessors, 2003.

[4] A. Avritzer and E. J. Weyuker, Monitoring smoothly degrading systems for increased dependability, Empirical Software Engineering, vol. 2, no. 1, pp. 59–77, 1997.

[5] Y. Bao, X. Sun, and K. S. Trivedi, A Workload-based Analysis of Software Aging and Rejuvenation, IEEE Transactions on Reliability, Vol. 54, No. 3, pp. 541-548, 2005.

[6] A. Bovenzi, D. Cotroneo, R. Pietrantuono, S. Russo, Workload characterization for software aging analysis, Proc. Int'l Symp. on Software Reliability Engineering (ISSRE 2011), pp. 240-249, 2011.

[7] B. Cantrill, M. Shapiro, A. Leventhal, Dynamic instrumentation of production systems, USENIX Annual Technical Conference, 2004.

[8] G. Carrozza, D. Cotroneo, R. Natella, A. Pecchia, and S. Russo, Memory leak analysis of mission-critical middleware, Journal of Systems and Software, Vol. 83, Issue 9, pp. 1556-1567, 2010.

[9] V. Castelli, R. E. Harper, P. Heidelberger, S. W. Hunter, K. S. Trivedi, K. Vaidyanathan, and W. P. Zeggert, Proactive management of software aging, IBM Journal of Research & Development, Vol. 45, No. 2, pp. 311-332, 2001.

[10] D. Controneo, R. Natella, R. Pietrantuono, and S. Russo, A survey on software aging and rejuvenation studies, IACM Journal on Emerging Technologies in Computing Systems (JETC), Vol. 10, No. 1, 2014.

[11] M. Grottke, R. Matias, and K. S. Trivedi, The fundamentals of software aging, Proc. 1st Int'l Workshop on Software Aging and Rejuvenation (WoSAR), 2008.

[12] M. Grottke and B. Schleich, How does testing affect the availability of aging software systems?, Perf. Eval., Vol. 70, pp.179-196, 2013.

[13] J. D. Hamilton, Time Series Analysis, Princeton Univ Press, 1994.

[14] R. Hodrick, E. C. Prescott, Post-war U.S. business cycles: An Empirical investigation, Journal of Money, vol. 29, No. 1, 1980.

[15] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton, Software rejuvenation: Analysis, module and applications, Proc. Int'l Symp. on Fault Tolerant Computing (FTCS 1995), pp. 381-390, 1995.

[16] G. Hunt, and D. Brubacher, Detours: Binary Interception of Win32 Functions, 3rd USENIX Windows NT Symposium, USENIX, 1999.

[17] B. Jacob, P. Larson, B. Leitao, S. A. M M. Da Silva, SystemTap: Instrumenting the Linux Kernel for Analyzing Performance and Functional Problems, IBM Redbook, 2008.

[18] F. Langner, A. Andrzejak, Detecting Software Aging in a Cloud Computing Framework by Comparing Development Versions, Proc. 13th IFIP/IEEE Symposium on Integrated Network and Service Management (IM 2013), May 2013.

[19] F. Langner, A. Andrzejak, Detection and Root Cause Analysis of Memory-Related Software Aging Defects by Automated Tests, Proc. MASCOTS 2013.

[20] A. Macêdo, T. B. Ferreira, R. Matias Jr., The mechanics of memory-related software aging, Proc. 2nd Int'l Workshop on Software Aging and Rejuvenation (WoSAR), 2010.

[21] F. Machida, J. Xiang, K. Tadano and Y. Maeno, Software life-extension: a new countermeasure to software aging, Proc. 23rd Int'l Symp. on Software Reliability Engineering (ISSRE2012), pp.131-140, 2012.

[22] F. Machida, A. Andrzejak, R. Matias Jr., E. Vicente, On the effectiveness of Mann-Kendall test for detection of software aging, Proc. 5th Int'l Workshop on Software Aging and Rejuvenation (WoSAR), 2013.

[23] R. Matias Jr., B. Evangelista, and A. Macedo, Monitoring memory-related software aging: an exploratory study, Proc. 4th Int'l Workshop on Software Aging and Rejuvenation (WoSAR), 2012.

[24] R. Matias Jr., P. J. Freitas Filho, An experimental study on software aging and rejuvenation in web servers, Proc. 30th Int'l. Computer Software and Applications Conference, pp. 189-196, 2006.

[25] R. Matias Jr., Pedro A. Barbetta, K. S. Trivedi, and P. J. Freitas Filho, Accerelated degradation tests applied to software aging experiments, IEEE Transactions on Reliability, vol. 59, no. 1, pp. 102-114, 2010.

[26] R. Matias Jr., K. S. Trivedi, and P. R. M. Maciel, Using accelerated life tests to estimate time to software aging failure, Proc. Int'l Symp. Software Reliability Engineering (ISSRE2010), pp. 211-219, 2010.

[27] D. C. Montgomery, Introduction to statistical quality control, John Wiley & Sons, 1996.

[28] K. S. Trivedi, R. K. Mansharamani, D. Kim, M. Grottke, M. Nambinar, Recovery from Failures Due to Mandelbugs in IT Systems, Proc. Pacific Rim Int'l Symp. on Dep. Computing (PRDC2011), pp. 224-233, 2011.

[29] J. Yang and V. Makis, On the performance of classical control charts applied to process residuals, Computers and Industrial Engineering, Vol. 33, no. 3-4, pp. 121-124, 1997.

[30] J. Zhao, Y. Jin, K. S. Trivedi, and R. Matias Jr., Injecting memory leaks to accelerate software failures, Proc. Int'l Symp. on Software Reliability Engineering (ISSRE 2011), pp. 260-269, 2011.

[31] P. Zheng, Q. Xu, and Y. Qi, An advanced methodology for measuring and characterizing software aging, Proc. 4th Int'l Workshop on Software Aging and Rejuvenation (WoSAR), 2012.

Authorized licensed use limited to: Concordia University Library. Downloaded on May 14,2024 at 15:49:22 UTC from IEEE Xplore. Restrictions apply.