2011 22nd IEEE International Symposium on Software Reliability Engineering

Injecting Memory Leaks to Accelerate Software Failures

Jing Zhao, Yuliang Jin
Computer Science and Tech. Dept.
Harbin Engineering University
Harbin, China
{jingzhao.duke, jyl198803}@gmail.com

Kishor S. Trivedi
Electrical and Computer Eng. Dept.
Duke University
Durham, USA
kst@ee.duke.edu

Rivalino Matias Jr.
School of Computer Science
Federal University of Uberlandia,
Uberlandia, Brazil.
rivalino@facom.ufu.br

***Abstract*—A number of studies have reported the phenomenon of "Software aging", caused by resource exhaustion and characterized by progressive software performance degradation. We develop experiments that simulate an on-line bookstore application, following the standard configuration of TPC-W benchmark. We study the application failures caused by memory leaks, using the accelerated life tests method. In our experiments, the memory consumption rate is selected as the acceleration factor, and an IPL-lognormal model is used to estimate the time to failure at each acceleration level. Subsequently, the estimate of the time to failure distribution at normal condition is obtained. Our acceleration experimental results based on the IPL-lognormal model show that it can be used to greatly reduce the cost to obtain the time to failure at normal level, which can be used in scheduling software rejuvenation. Finally, we select the Weibull time to failure distribution at normal level, to be used in a semi-Markov process, to optimize the software rejuvenation trigger interval.**

Keywords—accelerated life tests; memory leaks; optimal software rejuvenation; semi-Markov process; software aging

I INTRODUCTION

Studies show that operational software failures are transient in nature, caused by phenomena such as overloads or timing and exception errors [1]. Grottke et al. classified software faults into three types according to potential manifestation characteristic: Bohrbug, Mandelbug, and Aging-related bug, and then analyzed the faults discovered in the on-board software for 18 JPL/NASA space missions based on this classification method [2]. Aging-related bugs cause an increasing failure rate, gradual software performance degradation, and may eventually lead to a system hang or crash. Software aging is mainly caused by the successive accumulation of the effects of aging-related fault activations. It leads to the exhaustion of system resources, mainly due to memory-leaks, unreleased locks, non-terminated threads, shared-memory pool latching, storage fragmentation, or comparable causes [3], [4]. Many of the causes of software aging are very hard to identify due to their randomness [5]. Hence, it is not uncommon to have unknown aging faults causing known aging effects. This undesired phenomenon exists not only in regular software such as web and application servers, but also in critical applications that require high dependability levels. Software aging could cause great losses in safety-critical systems [6], including the loss of human lives [7]. To counteract software aging, researchers have proposed a proactive approach called software rejuvenation (SR) [3]. Rejuvenation has been implemented in various computing systems, such as billing data collection systems, telecommunication systems, transaction processing systems, and spacecraft systems [8], [9], [10]. It involves occasionally terminating an application process, cleaning its internal state and restarting it in order to release system resources, so that the software performance is recovered. One or more indicators of aging can capture the aging behavior [1], [4], [19]. Such indicators are measurable metrics of the target system likely to be influenced by software aging.

The most popular web server on the Internet, the Apache web server [11], is known to suffer from software aging [12]. It has been demonstrated that the extent of software aging depends on the workload imposed on the system. For examples see [1], [12], [13], [14] for Apache web server, and see [15], [16] for Axis. Most of the previous experimental research on software aging and rejuvenation employed Apache web server as a test bed, and then used statistical methods to predict the time to resource exhaustion [3], [12], [14], [16]. Analytic models used for capturing software rejuvenation are based on the assumption that the distribution of time to failure due to software aging is known, and the aim is to determine the optimal times to trigger rejuvenation in order to maximize system availability or related measures [4], [15], [17]. Whatever approach is used rejuvenation scheduling, such as measurement based, analytic, or both, estimated time to failure should be obtained more efficiently. Due to the difficulty in experimentally studying aging-related system failures by observation of failure times, Matias et al. develop a systematic approach to accelerate the aging effects at the experimental level [18]. They introduce the concept of aging factors and use different levels of accelerated workload to increase the system degradation. Based on the degradation data of selected system characteristics, captured through measurements, they apply the statistical technique of accelerated degradation tests (ADT) to estimate the time to failure in normal condition (without acceleration). Alternatively, in [19] the authors do not use degradation data, but directly observe failures obtained also under accelerated workloads. In this case, they use another technique called accelerated life tests (ALT)

1071-9458/11 $26.00 © 2011 IEEE
DOI 10.1109/ISSRE.2011.24

to estimate the time to failure in normal conditions. In both studies the system under test was based on the Apache web server.

Memory leaks are recognized to be one of the major causes of resource exhaustion problems in complex software, which represent the one of the most serious cause of aging. In [20], the authors focus on two types of memory problems (fragmentation and leakage) that cause software aging, presenting an experimental study on the cumulative effect of these problems in software systems [20], [21]. Alonso et al. [22] inject memory leaks to intensify memory consumption to derive the nonlinear memory resource behavior, and then use machine-learning algorithms to predict whether software aging has reached a given threshold. In this paper, motivated by the ALT method discussed in [19], we also inject memory leaks to intensify memory consumption so as to accelerate application failure. We then derive the estimates of time to failure (TTF) at different acceleration levels as well as in normal condition. Such an estimate is then used in scheduling software rejuvenation. Memory leak is one of the many types of aging effects, but any other aging effect (e.g., fragmentation problems) can be included in our approach. To the best of our knowledge, there are no other approaches to accelerate the aging effects in software aging. Comparisons can be done using failure data obtained at acceleration levels with data collected without acceleration (at normal level) and without rejuvenation. The major problem is that without acceleration, observing aging related failures would take too long a time, which provides the motivation to investigate this approach. The requirements to apply it are to select the stress variable and their effects on the system under test. It is not different than in other engineering areas, where several techniques are combined.

The contribution of this paper is as follows. First, the MTTF at normal conditions is obtained by ALT method, which greatly reduces the experimental time to derive the MTTF. Second, MTTF along with Weibull distribution of time to failure is used in a semi Markov model to compute the optimum rejuvenation trigger interval. Thus, the uniqueness of this paper is in combining experimentation, statistics, probabilistic models, and optimization.

The rest of the paper is organized as follows. In Section II we show how to use ALT applied to systems suffering from software aging. In Section III, the experimental setup and data collection are explained, where we describe how memory leak is injected to derive the system TTF samples at different acceleration levels. In Section IV, we discuss how to use an IPL-lognormal analytical model to estimate the mean time to failure for the system running at normal level. In Section V, we explain the use of the Weibull time to failure distribution along with a semi-Markov process in order to optimize the software rejuvenation trigger interval with the system availability and operational cost as objective functions. Finally, we present our conclusions in Section VI.

II ALT Method Applied to Software Aging

Accelerated life tests method (ALT) is successfully applied in many engineering fields [23] to significantly reduce the experimentation time, which is designed to quantify the life characteristics (e.g., mean time to failure) of a system under test (SUT). By applying controlled stresses to reduce the SUT's lifetime, the SUT is tested in an accelerated mode, and results are then adjusted to its normal operational condition. Thus, ALT uses the lifetime data obtained under accelerated stresses to estimate the lifetime distribution of the SUT for its normal condition. This systematic approach can be divided into four main steps: 1) selection of accelerating stress, 2) ALT planning and execution, 3) definition of the life-stress aging relationship, and 4) estimation of underlying life distribution (pdf) for the normal condition. The following sections will discuss each step in detail.

A. Selection of accelerating stress

A fundamental element during test planning is the definition of accelerating stress variable and its levels. Typical engineering accelerating stresses are temperature, vibration, humidity, voltage, and thermal cycling [23]. However, software reliability engineering does not have standards, related to software accelerating stresses for ALT. Given the nature of aging related faults, we can determine suitable accelerating stresses based on experiments. Based on [18], we employ memory consumption rate as the stress factor and use constant stress loading scheme in this paper. We inject memory leaks into a web server software to intensify its memory consumption rate.

B. ALT planning and execution

After selecting the acceleration factor, we can plan the ALT. This activity includes the following elements: number of stress levels, the amount of stress applied at each level, the allocation proportion in each level, and the sample size. In our approach to apply ALT for software components, the sample size is the number of test replications. According to the theory, the ALT test plans can be classified as: traditional, optimal, and compromise plans [23]. The traditional plans usually consist of three or four stress levels, with the same number of replications allocated at each level. The optimal plans specify only two levels of stress, high and low. The compromise plans usually work with three or four stress levels, and use an unequal allocation proportion. A more detailed description of the three plans can be found in [23]. In our approach, the traditional plan with four levels is used.

C. Life-Stress aging relationship

Once the SUT is tested at the selected stress levels, the estimate of the mean time to failure (MTTF) at normal condition is to be obtained from the TTF samples obtained at different stress levels. From the ALT theory, we know that the Inverse Power Law (IPL) model fits well ALT

applications for positive stress. Therefore, we need to build the relationship between life-stress at accelerated and normal levels. As an example, consider the life-stress model that is known as the Inverse Power Law (IPL) [23]:

$$L(s) = \frac{1}{k \cdot s^w} \quad (1)$$

where L represents a SUT life characteristic (e.g., mean time to failure), s is the stress level, k (k>0) and w are model parameters to be determined from the observed failure time samples.

D. Lifetime Distribution Estimation

Assuming that the TTF sample is exponentially distributed, IPL yields the pdf (probability density function) of TTF as:

$$f(t,s) = ks^w e^{-ks^w t} \quad (2)$$

Maximum-likelihood estimation (MLE) method can be applied to estimate the model parameters (k, w), and then use them to estimate the MTTF, L, for the SUT under normal stress level.

III EXPERIMENTAL SETUP

We adopt ALT to experimentally study application failure affected by software aging. Based on failure time samples collected under different stress loadings, the estimate of the time to failure distributions at different acceleration levels as well as at normal condition are obtained.

A Test Bed

To study the aging effects of application failures caused by memory leaks, we execute experiments that reproduce a typical web application. Our test bed is composed of a web server, a database server, and a set of clients. The database and web servers are on the same physical machine while all the clients occupy another physical machine. We have used a multitier e-commerce web site that simulates an on-line bookstore. We ran the experiments on Tomcat web container, and all HTML pages were dynamically generated by the server. In TPC-W standard, there are 14 different web pages including home page, best selling page, new books page, search page, shopping cart and order status page, etc. The web traffic is generated by a Remote Browser Emulator (RBE), which emulates users of the website. RBE maintains the mix of web interactions for EBs. These include three standard mixes of web interactions: Browsing Mix, Shopping Mix, and Ordering Mix. We use Shopping Mix in our tests. The details of these Mixes are shown in the table of TPC-W benchmark [24]. The workload is based on the standard configuration of TPC-W benchmark. This environment also includes Java servlets, MySQL as the database server, and Apache Tomcat as the application server [25]. TPC-W allows us to run different experiments using different parameters and under a controlled environment. TPC-W clients, the so-called Emulated Browsers (EBs), access the web site in sessions. A session is a sequence of logically connected requests (from the EB point of view). Between two consecutive requests from the same EB, TPC-W undergoes a "thinking phase" [24], representing the time between the users receiving the web page requested and generating the next request. TPC-W has three kinds of workload (Browsing, Shopping, and Ordering). We conduct our experiments using Shopping transactions only. Figure 1 presents the experimental environment used in this work.

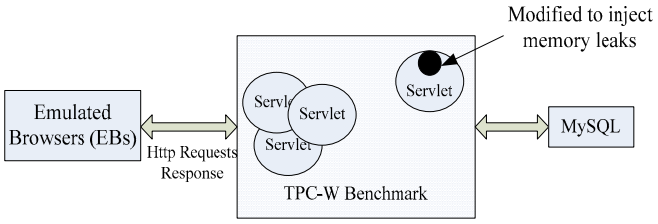

Figure 1. Experimental environment

B Injecting memory leaks

To emulate the aging effects consuming resources until the application failure, we have modified the TPC-W implementation by changing the TPCW_search_request_servlet to inject memory leaks. Each memory injection causes about 1 megabyte of memory leak. In our case, the maximum memory that can be used by JVM is 128 megabytes. The servlet class relationship including TPCW_search_request_servlet is shown in Figure 2. Furthermore, we add a piece of code to the servlet so as to modify the *doGet*() method inside TPCW_search_request_servlet.

The *doGet*() modification is illustrated in Figure 3. A random number from 0 to N is generated, where N is specified in a configuration file. The *randomNumber* value determines how many requests can use the servlet before the next memory leak is injected. This number is decreased by one on each invocation of *doGet*(), i.e., on every visit of the search request page. When this number is reduced to zero, a new data is appended to the BigObjList and in the same time a new *randomNumber* is generated. Thus, the time between memory leaks depends on the frequency of the servlet invocations. According to the TPC-W specification, this frequency depends on the workload chosen. In our experiment, we select the workload to be 100 EBs. Hence, under high workload our servlet injects memory leaks quickly. On the other hand, under low workloads the leak rate is lower. But, in the long term the average leak rate would depend on the average value of the random variable *randomNumber*, with fluctuations that become less relevant when averaged over time. Therefore, since the memory consumption rate would depend on the value of N, we can simulate this effect by varying N.

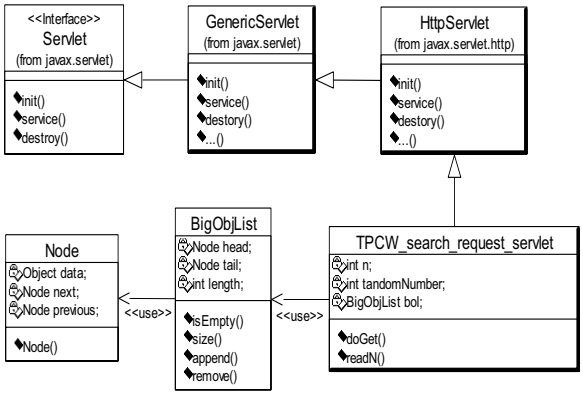

Figure 2. Java servlet class

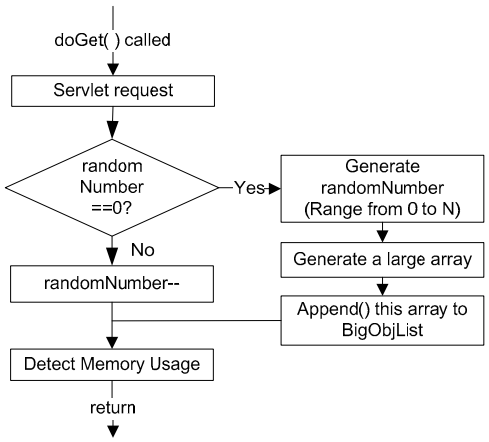

Figure 3. Modification of doGet( )

The Java heap memory is divided into three main zones: Young, Old, and Permanent. A Java object is created in one of these heap zones and is garbage-collected after there are no references to it. Resource behavior can look quite different depending on our monitoring strategy. Memory usage by a Java application looks quite different if we monitor it from the operating system (OS) level or from inside the JVM. From the OS level, when a Java application frees up memory objects, it is not possible to perceive the changing trend for the memory used. However, if we observe it from inside the JVM, then we can obtain accurate measures. Permanent Generation stores JVM's internal representation of the Java Classes that are mostly static, unchanged during the lifetime of JVM. In our experiment, we focused on the change of JVM heap, Java Objects. It can be justified because the JVM starts reserving a maximum amount of memory to be used by dynamic memory allocations, and JVM takes care of it without involving the OS. This strategy prevents the OS to be aware of the allocations and deallocations occurring inside the JVM. Therefore, in our experiment we measure memory consumption inside the JVM, and collect data on the Young and Old heaps [26].

We use JVM monitoring tool jmap [26] to collect the JVM memory exhaustion data. A script written in java on the server invokes jmap periodically, and thence obtains the memory usages of Young, Old and Permanent. We collect Young plus Old used at each five-second interval. The Young plus Old used space as well as the capacity when *N* equals 7 is shown in Figure 4(a), and Old memory free is shown in Figure 4(c). In addition, Java's Runtime class provides a lot of information about the resource details of Java Virtual Machine. Java's Runtime API is invoked when TPCW_search_request_servlet is injected, so that we collect runtime memory used by JVM from the servlet perspective. This can further account for the memory exhaustion at each injection point. Run time memory used is shown in Figure 4(c). In Figure 4(a), we see that the Young plus Old memory of JVM is used up, since it is close to both heaps' maximal capacity, while in Figure 4(b) we see that the runtime memory used is close to the JVM capacity. The Young plus Old capacity is shown in Figure 4(a), and from Figure 4(c), we see that the Old memory free is close to zero. From these figures, we can see the memory exhaustion of JVM due to memory leaks from different perspectives and see that different views are consistent.

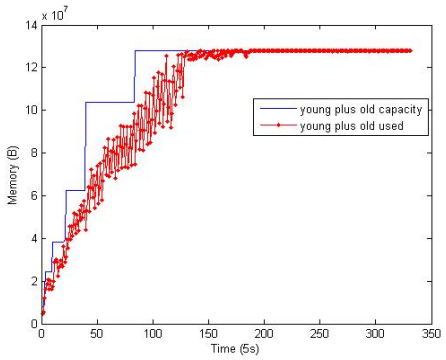

Figure 4(a). Young + Old heaps used

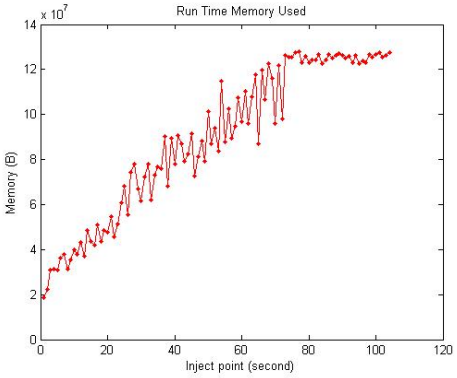

Figure 4(b). Run time memory used

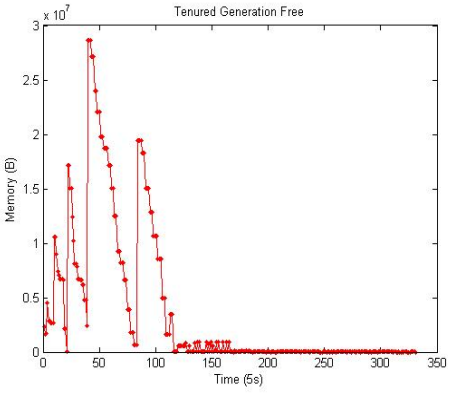

Figure 4(c). Old heap memory free

IV RESULTS ANALYSIS

In our experiment, we design four acceleration levels (S1 to S4) with *N* equals to 4, 8, 12, and 16, for each level, respectively. We run 7 replications at each acceleration level, thus 28 samples are obtained in total. Also, we use the algorithm described in [23] to calculate the sample size, n_{ALT}, thus verifying whether the initial number of samples satisfies the criteria of statistical analysis required by the ALT method. The n_{ALT} can be calculated by solving equations from (3) to (7):

$$\bar{x} = (n_1 x_1 + \ldots + n_j x_j)/np \tag{3}$$

where x_j is the stress level value transformed ($\log_e$), n_j is the number of replications executed in the *j*th level of stress, and *np* is the total number of tests executed, given that $np = n_1 + \ldots + n_j$.

$$\overline{y_j} = (y_{1j} + \ldots + y_{n_j j})/n_j \tag{4}$$

where $y_{n_j j}$ is the transformed ($\log_e$) value of the *j*th failure time obtained in the level of stress *j*.

$$s_j = \left\{ \frac{[(y_{1_j j} - \overline{y_j})^2 + \ldots + (y_{n_j j} - \overline{y_j})^2]}{v_j} \right\}^{\frac{1}{2}} \tag{5}$$

where v_j ($v_j = n_j - 1$) is the degrees of freedom of the s_j, and s_j is the standard deviation of failure times obtained at *j*th strees level.

$$s = \left\{ \frac{v_1 {s_1}^2 + \ldots + v_j {s_j}^2}{v} \right\}^{\frac{1}{2}} \tag{6}$$

where *v* is the number of degrees of freedom calculated as $v = v_1 + \ldots + v_j$, and *s* is the pooled estimate of the log transformed ($\log_e$) standard deviation (σ) in (7).

$$n_{ALT} = \left\{ 1 + (x_0 - \bar{x})^2 \left[\frac{np}{\sum (x - \bar{x})^2} \right] \right\} \left(\frac{z_{\alpha/2}\sigma}{\xi} \right)^2 \tag{7}$$

where x_0 is the log transformation ($\log_e$) of the stress value assumed at normal use rate, and *z* is the tabulated value for the standard normal distribution at a given significance level α, ξ is the precision of the estimate, which depends on the fitted pdf and the metric of interest for the accelerated failure times, where $\xi = r$ (for mean) or $\xi = \log_e r$ (for median); *r* is the precision for the estimator of interest [23]. When $\xi = r$, *r* is the half width of the interval used to calculate the confidence interval for the mean. Alternatively, $r = (1+m)$, where ($m \cdot 100\%$) is the tolerated error for the estimator of the median.

Based on the experiment results, we calculate the mean memory consumption rate from runtime memory used, at each acceleration level. For each replication, the memory consumption rate is calculated using the Sen's slope estimate method [27]. Sen's non-parametric method is a way to estimate the true slope of the data, that is, if the data shows an upward trend, there is evidence of an upward trend. These results are shown in Table I. Next, we conduct the experiment removing acceleration factors so as to calculate the memory consumption rate at normal level. We observe that when the workload is equals to 100 EBs, the total experimental time is 398790 seconds, or 4.615625 days. This target is to obtain the memory consumption rate at normal level calculated by Sen's slope. The memory consumption rate of Young and Old heaps is approximately 0.0121kB/s, based on the Sen's slope estimation method (see Figure 5). We use the random number to simulate the randomness of requests, and different random number values correspond to different memory consumption rates. The mean consumption rate is obtained using 7 replications at each acceleration level. For example, when the random number *N* equals 16, the mean memory consumption rate is 56.18 kB/s.

TABLE I. MEMORY CONSUMPTION RATE AND *N*

Memory consumption rate (kB/s)	*N*	Memory consumption rate per replication
0.0121		*normal level*
56.18	16	51.521, 55.986, 57.66, 62.102, 51.851, 52.844, 61.295, 56.18
69.341	12	69.838, 73.844, 65.769, 70.645, 65.832, 63.443, 76.019, 69.341
97.613	8	90.067, 93.847, 102.38, 100.16, 98.2, 101.87, 96.765, 97.613
166.39	4	157.44, 160.57, 179.87, 176.88, 155.07, 164.93, 169.97, 166.39

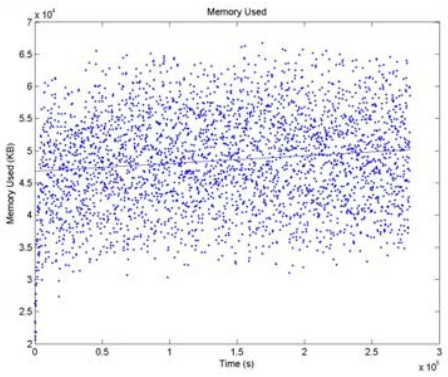

Figure 5. Memory consumption rate of Young plus Old heaps calculated using Sen's slope method.

The samples of failure times at each acceleration level are shown in Table II. It corresponds to the application failures caused by out of memory. According to [18], we evaluate the probability distributions Weibull, Lognormal, and Exponential to identify the best fit. The criterion used to build the best-fit ranking is the log-likelihood function (*Lk*) [28].

TABLE II. SAMPLE OF FAILURE TIMES (SECONDS)

TTF (S1)	TTF(S2)	TTF(S3)	TTF(S4)
455	665	965	1230
425	670	930	1155
430	655	1005	1160
440	685	930	1060
465	720	1000	1065
465	640	1000	1185
430	715	935	1360

TABLE III. SAMPLE OF LOG TRANSFORMED ACCELERATION DATA

TTF (S1)	TTF(S2)	TTF(S3)	TTF(S4)
6.1203	6.4998	6.8721	7.1148
6.0638	6.5073	6.8352	7.0519
6.0521	6.4846	6.9127	7.0562
6.0868	6.5294	6.8352	6.9660
6.1420	6.5793	6.9078	6.9707
6.1420	6.4615	6.9078	7.0775
6.0638	6.5723	6.8405	7.2152

TABLE IV. RESULTS OF MODEL FITTING FOR ACCELERATED FAILURE TIMES

Accelerated level	Model	*Lk*	Best-fit Ranking
16 (S1)	Lognormal	-41.6922	1st
	Weibull	-42.5207	2nd
	Exponential	-56.4746	3rd
12 (S2)	Lognormal	-34.3337	2nd
	Weibull	-34.3237	1st
	Exponential	-55.1153	3rd
8 (S3)	Lognormal	-33.3200	1st
	Weibull	-33.6147	2nd
	Exponential	-52.6399	3rd
4 (S4)	Lognormal	-29.3888	1st
	Weibull	-29.5596	2nd
	Exponential	-50.2522	3rd

The natural logarithms of acceleration sample data sets are shown in Table III. We calculate n_{ALT} to be 25 using formula (3) to (7), which is the minimum number of samples needed for our ALT test plan. Since this number is smaller than we obtained in our sampling, we satisfy the ALT assumptions for sample size.

The fitting results for these four models are shown in Table IV. The Lognormal distribution provides the best fitting results for acceleration levels S1, S3, S4, and its fitting result is close to Weibull's at acceleration level S2, so we chose Lognormal combined with IPL model to create our life-stress relationship model. The probability density function (pdf) of Lognormal is shown in Equation 8.

$$f(t)=\frac{1}{t\sigma_{t'}\sqrt{2\pi}}e^{-\frac{1}{2}(\frac{t'-\bar{t}'}{\sigma_{t'}})^2} \tag{8}$$

where, t = time to failure, t'=ln(t), $\bar{t}'$ = mean of the natural logarithms of the time to failure, $\sigma_{t'}$=standard deviation of the natural logarithms of the failure times.

The life characteristic for the Lognormal distribution is its median value that is given by (9)

$$\breve{t}=e^{\bar{t}'}. \tag{9}$$

The pdf for the IPL-lognormal model can be obtained first by setting $\breve{t}=L(v)$ in (1). Then

$$e^{\bar{t}'}=L(v)=\frac{1}{k\cdot v^{w}}\quad , \tag{10}$$

therefore,

$$\bar{t}'=-\ln(k)-w\ln(v), \tag{11}$$

thus substituting (11) into (8) yields the IPL-lognormal pdf as shown in (12).

$$f(t)=\frac{1}{t\sigma_{t'}\sqrt{2\pi}}e^{-\frac{1}{2}(\frac{t'+\ln(k)+w\ln(v)}{\sigma_{t'}})^2} \tag{12}$$

As a result, (13) can be directly derived from (12) and used to estimate the mean time to failure, MTTF, of the SUT at a specific use rate.

$$MTTF(v)=e^{-\ln(k)-w\ln(v)-\sigma^2_{t'}} \tag{13}$$

TABLE V. PARAMETER ESTIMATION OF LOGNORMAL MODEL

Accelerated level	Parameter	ML estimate	90% confidence interval	
			Lower	Upper
S1	μ_1	7.0646	7.0072	7.1220
	σ_1	0.0923	0.0526	0.1622
S2	μ_2	6.8730	6.8498,	6.8962
	σ_2	0.0373	0.0222	0.0627
S3	μ_3	6.5192	6.4894	6.5490
	σ_3	0.0479	0.0269	0.0854
S4	μ_4	6.0958	6.0705	6.1212
	σ_4	0.0408	0.0236	0.0705

In order to verify the scale of invariance, we verify whether the estimated μ and σ values at each acceleration level are inside the same confidence interval. Table V gives the parameter estimation of lognormal model by the ML estimation method [23].

The estimated IPL-lognormal parameters are listed in Table VI. We use Equation 12 to estimate the underlying life distribution for the normal condition. Figure 6(a) presents the fit of the IPL-lognormal model estimated from the four stress levels, and for the normal condition. Figure 6(b) shows the failure and stress relationships. The calculated estimate for time to failure (x-axis) at the normal level starts at 0.0124. As a result, we can obtain the MTTF for the normal level from this model that equals to 2.1114E+6 seconds (24.4375 days). The 90% confidence interval is (1.3503E+6, 3.3014E+6) seconds - about (15.63, 38.21) days.

TABLE VI. IPL-LOGNORMAL PARAMETERS

Parameter	ML Estimate	90% confidence interval	
		Lower	Upper
k	2.3225E-5	1.8529E-5	1.8529E-5
w	0.8979	0.8478	0.9480
σ	0.0555	0.0427	0.0721

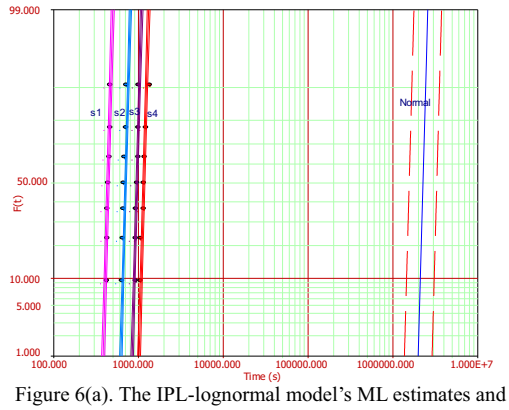

Figure 6(a). The IPL-lognormal model's ML estimates and 90% confidence intervals for *F(t)* at normal level

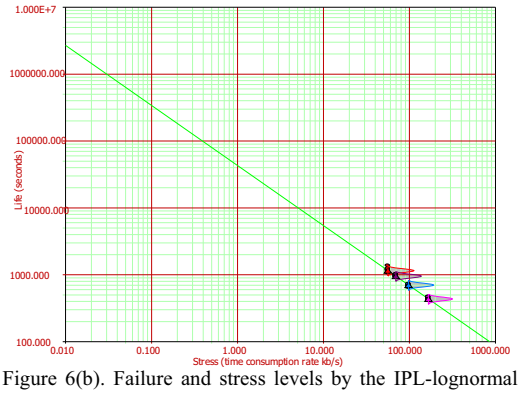

Figure 6(b). Failure and stress levels by the IPL-lognormal model estimation

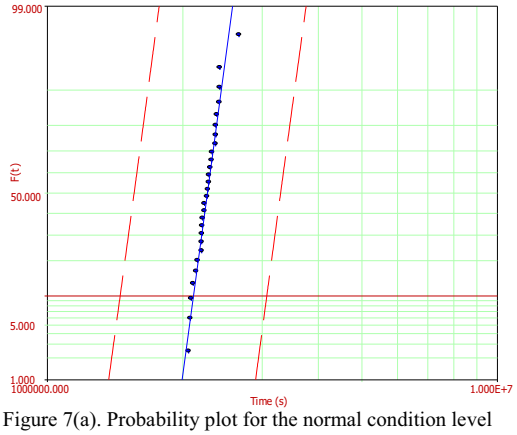

Figure 7(a). Probability plot for the normal condition level

Figure 7(a) presents the probability plot for the normal condition level, and the standardized residuals plot is shown in Figure 7(b), which confirms the good fit of the estimated model.

From these analysis results of tables and figures, we can see the experimental costs to obtain the metric of the normal TTF at experimental level, is greatly reduced. In addition, the estimates of the time to failure distributions at different acceleration levels as well as normal condition are obtained. These results can be used to further schedule software rejuvenation, and thus improve the software availability, and reduce the maintenance costs.

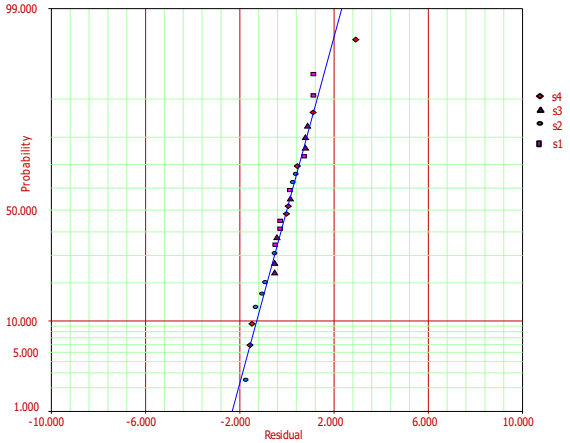

Figure 7(b). Standardized residuals.

IV OPTIMAL SOFTWARE REJUVENATION

Based on the results discussed in Section V, we employ the preventive maintenance model presented by Chen and Trivedi in [29], using the Weibull time to failure distribution. We optimize the software rejuvenation trigger interval in order to maximize the system availability or minimize the operational cost. Figure 8 shows this model. It consists of three states: UP state, or state 0, in which the system is up; RJ state, or state 1, in which the system is undergoing software rejuvenation, and DOWN state, or state 2, in which the system is down and under reactive

repair. State 0 is the only available state. From state 0 the system will enter state 1 with a general distribution function $F_0(t)$, for the software rejuvenation trigger interval, or fail and enters into state 2 with a general time to failure distribution $F_2(t)$. The distribution function for the duration of software rejuvenation (proactive repair) is $F_1(t)$, and the distribution function for the duration of reactive repair is $F_3(t)$. This model is a semi-Markov process [28].

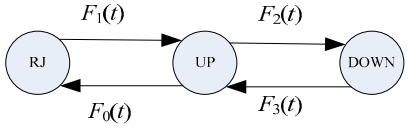

Figure 8. Rejuvenation model.

We assume that the rejuvenation trigger interval is deterministic (t_0) and the mean time to carry out the rejuvenation and reactive repair are t_1 and t_2, respectively. The two-parameter Weibull pdf for TTF is given by:

$$f(t) = \frac{\beta}{\eta}(\frac{t}{\eta})^{\beta-1} e^{-(\frac{t}{\eta})^{\beta}} \quad (14)$$

where,

$f(t) \geq 0$, $t \geq 0$, $\beta > 0$, $\eta > 0$,

η = scale parameter,

β=shape parameter (or slope),

The CDF of this Weibull distribution is given by:

$$F(t) = 1 - e^{-(\frac{t}{\eta})^{\beta}} \quad (15)$$

The sojourn time in UP state is then given by:

$$h_0 = \int_0^{t0} (1-F(t))dt = \frac{\eta}{\beta}\Gamma(\frac{1}{\beta})G(\frac{1}{\eta^{\beta}}t_0^{\beta}, \frac{1}{\beta}) \quad (16)$$

where $G(x,\beta) = \frac{1}{\Gamma(\beta)}\int_0^x e^{-u}u^{\beta-1}du$ is the incomplete gamma function. Hence, we can get the steady state availability:

$$A_{weib} = \frac{h_0}{h_0+(1-F(t_0))t_1+F(t_0)t_2} \quad (17)$$

The IPL-Weibull model can be derived by setting $\eta = L(v)$, yielding the following IPL-Weibull pdf:

$$f(t,V) = \beta k V^w (KV^w t)^{\beta-1} e^{-(KV^w t)^{\beta}} \quad (18)$$

This is a three-parameter model. The estimated IPL-Weibull parameters are listed in Table VII.

TABLE VII. IPL-WEIBULL PARAMETERS

Parameter	ML Estimate	90% confidence interval	
		Lower	Upper
β	15.2094	11.7317	19.7182
k	2.9252e-5	2.3644e-5	3.6189e-5
w	0.8696	0.8212	0.9180

We obtain the MTTF at normal level as 8.7804E+5 seconds when memory consumption rate at normal level is 0.0124 kB/s. The 90% confidence interval of MTTF at normal level is (5.4424E+5, 1.4166E+6) seconds. Correspondingly, for the parameter η confidence interval is denoted by (η_{low}, η_{high}) and is computed as (5.4424E+5, 1.4166E+6). Also, the parameter β confidence interval, denoted by, (β_{low}, β_{high}) is (11.7317, 19.7182) as shown in Table VII.

Therefore, from (17) we derive the steady state availability $A=A_{weib}(\eta,\beta)$, and its confidence interval $A_{low}=A_{weib}(\eta_{low}, \beta_{low})$, $A_{high}=A_{weib}(\eta_{high}, \beta_{high})$. We assume that the mean duration for carrying out software rejuvenation, t_1, is 1 minute, and the mean time for reactive repair, t_2, is 5 minutes. Steady-state availability *vs* time to rejuvenation trigger, t_0, assuming the Weibull time to failure distribution is shown in Figure 9. The optimal time to trigger rejuvenation and the corresponding availability are marked in this figure. In this case, the optimal choice of rejuvenation trigger interval could accrue availability improvement.

Another objective function is to minimize the expected cost. A cost of C_f per minute is incurred when the system is down due to system failure, and a cost of C'_f is incurred for each reactive repair carried out; a cost of C_p per minute is incurred when the system is down for carrying out software rejuvenation, and a cost of C'_p is incurred for each rejuvenation action carried out. The total expected cost per minute is thus

$$C = C_f\pi_2 + C'_f\pi_2/t_2 + C_p\pi_1 + C'_p\pi_1/t_1,$$

where π_2/t_2 and π_1/t_1 are the average number of reactive repairs and rejuvenation executions per minute, respectively.

We assume that $C_p = C'_p = 1/60$, $C_f = C'_f = 5/60$. Let $C=C(\eta,\beta)$, $C_{low}= C(\eta_{low},\beta_{low})$, and $C_{high}= C(\eta_{high},\beta_{high})$, so we derive the cost C, C_{low} and C_{high}. The average cost *vs*. time to rejuvenation t_0 is shown in Figure 10. The optimal intervals for the cost models are as short as 5930 to 17810 minutes, while the optimal intervals for the availability models are 6590 to 18980 minutes.

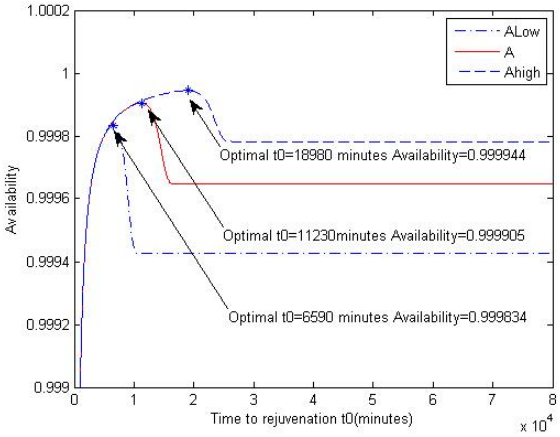

Figure 9. Steady-state availability *vs* time to rejuvenation t_0

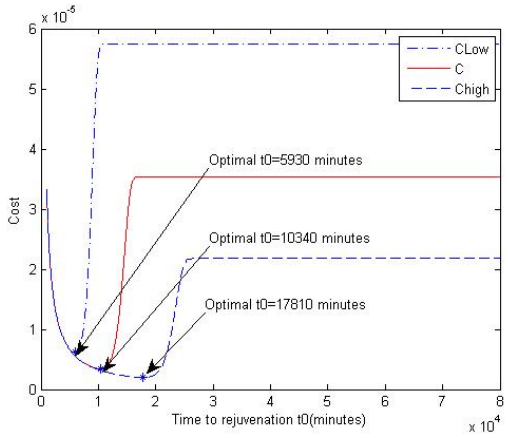

Figure 10. Average cost vs. time to rejuvenation t_0

V CONCLUSION

In this paper, we develop experiments that simulate software aging in an on-line bookstore application. We study the effects of software failures caused by memory leaks using the accelerated life tests method, following the standard workload of the TPC-W benchmark. Based on the collected data on the JVM memory consumption, first the memory consumption rate was selected as the acceleration factor. Secondly, the IPL-lognormal model was built to estimate TTF at normal level. Thirdly, Weibull time to failure distribution is used in a semi-Markov process, to optimize the software rejuvenation trigger interval so as to maximize the availability or minimize the operational cost.

Results show that the MTTF at normal use condition is 24.4375 days. A considerable reduction in experimentation time is achieved by using the ALT method. Four stress levels and 7 replications are used at each stress level. The experimental time of each replication varies from 425 seconds to 1360 seconds. Another contribution of this paper is that we use the results of TTF distribution estimates to optimize the software rejuvenation trigger interval. Since the known closed form result is only for the Weibull time to failure distribution we use that over Lognormal. Our results show only minor difference between the goodness of fit between the Weibull and the Lognormal. We also compute the availability confidence interval and operational cost at the optimal rejuvenation trigger interval using the estimated parameters from our experiments.

ACKNOWLEGEMENT

This work is supported in part by the National Natural Science Foundation of China under Grant No. 60873036. This work was also supported in part by FAPEMIG and CNPq, Fundamental Research Funds for the Central Universities (award number HEUCF100601, HEUCFT1007), and Fundamental Research Funds for Harbin Engineering University.

REFERENCES:

[1] M. Grottke, L. Li, K. Vaidyanathan, K.S. Trivedi, "Analysis of Software Aging in a Web Server," *IEEE Transactions on reliability*, 55(3), pp. 411-420, 2006.

[2] M. Grottke, A.P. Nikora, K.S. Trivedi, "An empirical investigation of fault types in space mission system software," *2010 IEEE/IFIP international conference on dependable systems & networks (DSN)*, pp. 447-456, 2010.

[3] S. Garg, A. van Moorsel, K. Vaidyanathan, K.S. Trivedi, "A methodology for detection and estimation of software aging," *in Proc. 9th international Symposium on software Reliability Engineering*, pp. 283-292, 1998.

[4] Y. Huang, C. Kinatla, N. Kolertis, "Software Rejuvenation: Analysis, Module and Applications," *in Proc. 25th Symposium on Fault Tolerant Computing*, pp. 381-390, 1995.

[5] M. Grottke, R. Matias, and K. Trivedi, "The fundamentals of software aging," In Proc of Workshop on Software Aging and Rejuvenation, in conjunction with IEEE International Symposium on Software Reliability Engineering. 2008.

[6] Y.F. Jia, L. Zhao and K.Y. Cai, "A Nonlinear approach to modeling of software aging in a web server", *in Proc. 15th Asia-Pacific Software Engineering Conference*, pp. 77-84, 2008.

[7] E. Marshall, "Fatal Error: How Patriot Overlooked a Scud," *Science*, 255(5050), pp. 1347, 1992.

[8] K.Y. Cai, "Software Reliability and control," *Journal of computer science and technology*, 21(5), pp. 697-707, 2006.

[9] K.J. Cassidy, K.C. Gross, and A. Malekpour, "Advanced pattern recognition for detection of complex software aging in online transaction processing servers," *in Proc. International conference on dependable systems and networks*, pp. 478-482, 2002.

[10] X.M. Zhang and H. Pham, "Predicting operational software availability and it's applications to telecommunication systems," *international journal of systems science*, 33(11), pp. 923-930, 2002.

[11] Apache web server, http://www.apache.org

[12] Y. Bao, X. Sun, K.S. Trivedi, "A workload-based analysis of software aging and rejuvenation," *IEEE Transactions on Reliability* 54 (3), pp. 541-548, 2005

[13] T. Dohi, K. Goseva-Popstojanova, and K.S. Trivedi, "statistical non-parametric algorithms to estimate the optimal software rejuvenation schedule," *proc 2000 pacific Rim int'l symp dependable computing PRDC*, pp. 77-84, 2000.

[14] S. Garg, A. Puliafito, M. Telek,, K.S. Trivedi, "Analysis of preventive maintenance in transactions based software systems," *IEEE Transactions on Computers* 47(1), pp. 96–107, 1998.

[15] L. Silva, H. Madeira, J.G. Silva , “Software aging and rejuvenation in a SOAP-based server,” *in proc. Intl. Symposium on Network computing and applications*, pp. 56-65, 2006.

[16] J. Alonso, L. Silva, A. Andrzejak, P. Silva, J. Torres, “High-Available Grid Services Through the Use of Virtualized Clustering,” *Proc. Intl. Conf. on Grid Computing* , pp. 34-41, 2007.

[17] K. Vaidyanathan, K.S. Trivedi, “A comprehensive model for software rejuvenation,” *IEEE Transactions on Dependable and Secure Computing*, 2(2), pp. 124–137, 2005.

[18] R. Matias, P. A. Barbetta, K. S. Trivedi, “Accelerated Degradation tests applied to software aging experiments”, *IEEE Transactions on reliability*, 59(1), pp 102-114, 2010.

[19] R. Matias, K. S. Trivedi, P. R.M. Maciel, “Using accelerated life tests to estimate time to software aging failure”, *ISSRE 2010*, pp. 211-219.

[20] A. Macêdo, T. B. Ferreira, R. Matias, “The Mechanics of Memory-Related Software Aging,” *2010 IEEE Second International Workshop on Software Aging and Rejuvenation (WoSAR)*.

[21] R. Matias, I. Beicker, B. Leitao, P.R.M Maciel, “Measuring software aging through OS kernel instrumentation,” *2010 IEEE Second International Workshop on Software Aging and Rejuvenation (WoSAR)*.

[22] J. Alonso, J. Berral, R. Gavaldà, J. Torres, “Adaptive on-line software aging prediction based on Machine Learning,” *Proc. Intl. Conf. Dependable Systems and Networks*, DSN 2010.

[23] B.N.Nelson, *Accelerated testing : statistical method, test plans, and data analysis*, New Jersey: Wiley, 2004.

[24] TPC-W Benchmark Java Version, http://www.ece.wisc.edu/~pharm/tpcw.shtml

[25] Apache Tomcat. http://tomcat.apache.org/

[26] Jstat:http://java.sun.com/j2se/1.5.0/docs/tooldocs/share/jstat.html

[27] P.K. Sen, “Estimates of the regression coefficient based on Kendall’s tau,” *Journal of the American Statistical Association*, 63(4), pp. 1379-1389, 1968.

[28] K.S. Trivedi, *probability and statistics with reliability, queuing, and computer science applications*. Second edition. John wiley & sons, New York, 2002.

[29] D. Chen and K.S. Trivedi, “Analysis of periodic preventive maintenance with general system failure distribution,” *Proc. of 2001 Pacific Rim International Symposium on Dependable Computing*, pp. 103-107