Software Rejuvenation Scheduling Using Accelerated Life Testing

JING ZHAO and YULIANG JIN, Harbin Engineering University
KISHOR S. TRIVEDI, Duke University
RIVALINO MATIAS JR., Federal University of Uberlandia
YANBIN WANG, Harbin Institute of Technology

A number of studies have reported the phenomenon of "Software aging", caused by resource exhaustion and characterized by progressive software performance degradation. In this article, we carry out an experimental study of software aging and rejuvenation for an on-line bookstore application, following the standard configuration of TPC-W benchmark. While real website is used for the bookstore, the clients are emulated. In order to reduce the time to application failures caused by memory leaks, we use the accelerated life testing (ALT) approach. We then select the Weibull time to failure distribution at normal level, to be used in a semi-Markov process, to compute the optimal software rejuvenation trigger interval. Since the validation of optimal rejuvenation trigger interval with emulated browsers will take an inordinate long time, we develop a simulation model to validate the ALT experimental results, and also estimate the steady-state availability to cross-validate the results of the semi-Markov availability model.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*Reliability, statistical methods*; G.3 [**Probability and Statistics**]: *Reliability and life testing*

General Terms: Performance, Reliability

Additional Key Words and Phrases: Accelerated life tests, memory leaks, optimal software rejuvenation, semi-markov process, simulation

ACM Reference Format:
Zhao, J., Jin, Y., Trivedi, K. S., Matias Jr. R., and Wang, Y. 2014. Software rejuvenation scheduling using accelerated life testing. ACM J. Emerg. Technol. Comput. Syst. 10, 1, Article 9 (January 2014), 23 pages.
DOI: http://dx.doi.org/10.1145/2539118

1. INTRODUCTION

Studies show that operational software failures are transient in nature, caused by phenomena such as overloads or timing and exception errors [Grottke et al. 2006]. Grottke et al. [2006] classified software faults into three types according to potential manifestation characteristics: Bohrbugs, Mandelbugs, and aging-related bugs, and

This research was supported in part by the National Natural Science Foundation of China, under Grant No. 60873036 and by the National Aeronautics and Space Administration's Office of Safety and Mission Assurance Software Assurance Research Program under a task managed by JPL's Assurance Technology Program Office.
This work was also supported in part by Brazilian CNPq, Fundamental Research Funds for the Central Universities (award number HEUCF100601, HEUCFT1007), and Fundamental Research Funds for Harbin Engineering University.
Authors' addresses: J. Zhao and Y. Jin, Computer Science and Tech. Dept., Harbin Engineering University; K. Trivedi, Electrical and Computer Engineering Department Duke University, Durham, NC 27708-0291, R. Matias, Jr., School of Computer Science, Federal University of Uberlandia, Uberlandia, Brazil; Y. Wang, College of Electrical and Mechanical, Harbin Institute of Technology. Correspondence email: jingzhao.duke@gmail.com.

DOI: http://dx.doi.org/10.1145/2539118

then analyzed the faults discovered in the on-board software for 18 JPL/NASA space missions based on this classification method [Grottke et al. 2010]. 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 [Matias et al. 2010c]. It leads to the exhaustion of system resources, mainly due to memory leaks, unreleased locks, nonterminated threads, shared-memory pool latching, storage fragmentation, or similar causes [Huang et al. 1995; Garg et al. 1998b; Grottke et al. 2008]. This undesired phenomenon exists not only in ordinary server applications, but also in critical applications that require high dependability levels. Software aging could cause great losses in safety-critical systems [Jia et al. 2008], including the loss of human lives [Marshall 1992]. To counteract software aging, researchers have proposed a proactive approach called software rejuvenation (SR) [Huang et al. 1995; Castelli et al. 2001]. Rejuvenation has been implemented in various computing systems, such as billing data collection systems, telecommunication systems, transaction processing systems, and spacecraft systems [Castelli et al. 2001; Cassidy et al. 2002; Zhang and Pham 2002; Cai 2006]. It involves occasionally terminating an application process or the operating system, 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 [Huang et al. 1995; Grottke et al. 2006; Matias et al. 2010c]. 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 [Apache 2011a], is known to suffer from software aging [Grottke et al. 2006]. It has been in general demonstrated that the extent of software aging depends on the workload imposed on the system [Vaidyanathan and Trivedi 2005; Bao et al. 2005; Grottke et al. 2006]. For examples, see Grottke et al. [2006] for Apache web server and see Silva et al. [2006] and Alonso et al. [2007] 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 [Grottke et al. 2006; Alonso et al. 2007]. With the exception of Vaidyanathan and Trivedi [2005], Bobbio et al. [2001], and Dohi et al. [2000] on optimal rejuvenation scheduling, most other 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 [Huang et al. 1995; Garg et al. 1998a; Silva et al. 2006]. Whatever approach is used for 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. [2010a] developed a systematic approach to accelerate the aging effects at the experimental level. They introduced the concept of aging factors and used 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). In Matias et al. [2010c], 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) to estimate the time to failure in normal condition. 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 one of the most serious causes of aging. Macêdo et al. [2010], focussed on two types of memory problems (fragmentation and

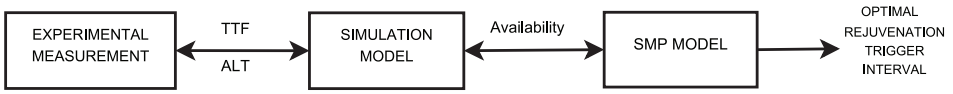


Fig. 1. Cross validation among ALT, simulation, and SMP model.

leakage) that cause software aging, presenting an experimental study on the cumulative effect of these problems in software systems [Macêdo et al. 2010; Matias et al. 2010b]. Alonso et al. [2010] injected memory leaks to intensify memory consumption to derive the nonlinear memory resource behavior, and then used machine-learning algorithms to predict whether software aging has reached a given threshold.

Motivated by the ALT method discussed in Matias et al. [2010c], we inject memory leaks to intensify memory consumption so as to accelerate application failures. We thus assume that the primary cause of aging in our system is memory leak and hence we neglect any other potential causes of aging (e.g., memory fragmentation). In the actual environment, memory consumption rate is affected by workload imposed on the system, but it is reasonable to consider and employ the long-term average memory consumption rate for the injection in our experiments. In our experiments, we inject memory leaks in the test bed by explicitly appending objects that cannot be recycled by the garbage collector, in order to obtain the average memory consumption rate. Experimental results are then used to derive the estimate the parameters of time to failure (TTF) distribution at different acceleration levels as well as in the normal use condition. These estimates are then used in determining an optimal rejuvenation schedule via a semi-Markov process (SMP) [Zhao et al. 2011].

In this article, we extend the results of our previous paper [Zhao et al. 2011], to further develop a detailed simulation model of the ALT experiment. The need for the simulation model arose from the fact that even with ALT, the measurement experiments on a real browser take an inordinate amount of time when rejuvenation is introduced in the test bed. By contrast, simulation is still possible to complete in a reasonable time even when rejuvenation is introduced. We sketch our idea in Figure 1. In the first phase of cross validation between ALT and simulation, we obtain the TTF results at different accelerated levels from the simulation model and thence we obtain the non-accelerated TTFs. We then cross-validate the results of the simulation with measurements on the real browser. In the second phase, we cross validate the availability results estimated from the simulation model with rejuvenation and semi-Markov availability model. We then go on to use the analytic semi-Markov model to determine the optimal rejuvenation trigger interval. We note that simulation includes individual client request generation and processing, besides memory leak injection, ALT and rejuvenation.

The main contributions of this article are as follows. First, we combine experimentation, statistical analysis using ALT, probabilistic (semi-Markov) models, discrete-event simulation and optimization in a single effort. Second, we cross-validate the ALT experimental results from a real browser against detailed simulation, and also the semi-Markov availability model with the detailed simulation model. The validated semi-Markov model is then used to determine the optimal rejuvenation trigger interval.

The rest of this article is organized as follows. In Section 2, we show how to use ALT in systems suffering from software aging. In Section 3, 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 4, we discuss how to use an IPL acceleration model to estimate the mean time to failure for the system running at (normal) use level. In Section 5, we explain the use of the Weibull time to failure distribution along with a semi-Markov process model in order to optimize

the software rejuvenation trigger interval with the system availability and operational cost as objective functions. In Section 6, we develop a detailed simulation model to cross validate the ALT experiment and the semi-Markov availability model. Finally, we present our conclusions in Section 7.

2. ALT METHOD FOR SOFTWARE AGING

Accelerated life tests method (ALT) is successfully applied in many engineering fields [Nelson 2004] 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 (probability density function, pdf) for the normal condition. The following sections will discuss each step in detail.

2.1. 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 [Nelson 2004]. 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 Matias et al. [2010a], we employ memory consumption rate as the stress factor in order to accelerate the memory consumption rate. We also adopt a constant stress loading scheme [Nelson 2004] in this article.

2.2. 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 [Nelson 2004]. 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 Nelson [2004]. In our approach, the traditional plan with four levels is used.

2.3. 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. 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) [Nelson 2004]:

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

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.

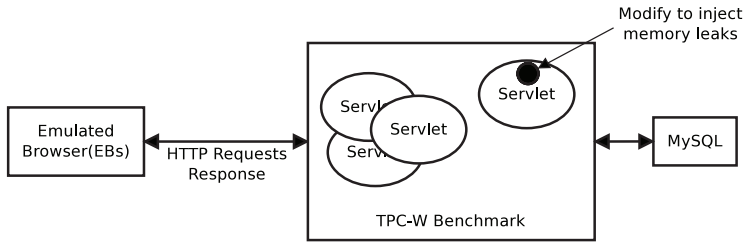


Fig. 2. Experimental environment.

2.4. Lifetime Distribution Estimation

Assuming that the TTF sample is exponentially distributed, IPL yields the pdf of TTF as:

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

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

3. 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.

3.1. 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 multi-tier e-commerce web site that simulates an on-line bookstore [Bezenek et al. 2011]. The workload is based on the configuration of TPC-W benchmark [TPC 2002]. This environment also includes Java servlets, MySQL as the database server, and Apache Tomcat as the application server [Apache 2011b]. 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", 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) [TPC 2002]. We conduct our experiments using Shopping transactions only. Figure 2 presents the experimental environment used in this article.

3.2. 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. The servlet class relationship including TPCW_search_request_-servlet is shown in Figure 3. 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 4. 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.

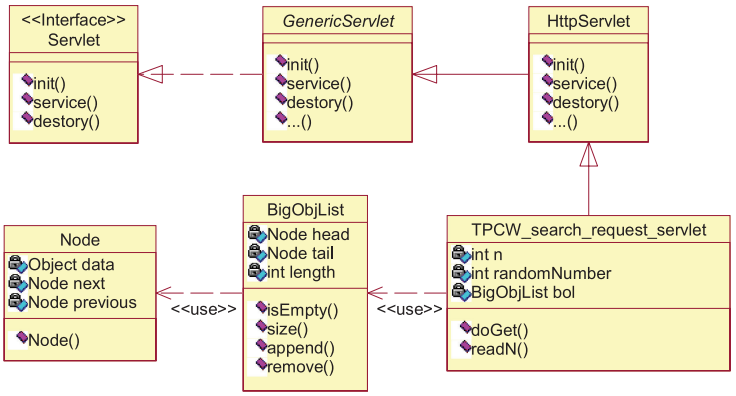


Fig. 3. Java servlet class.

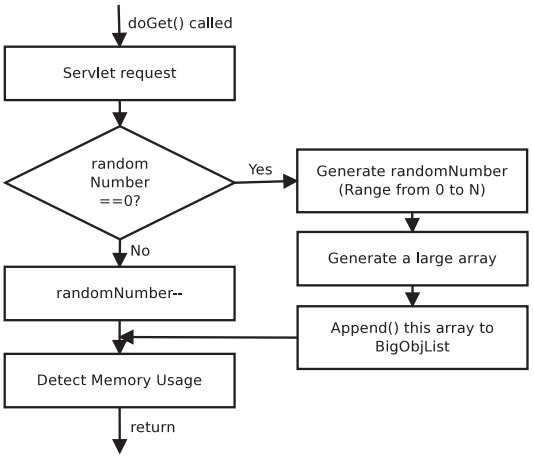
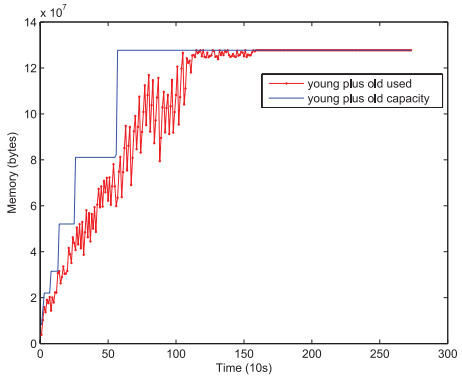


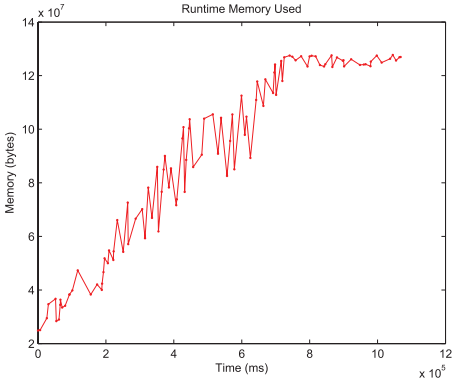
Fig. 4. Modification of *doGet()*.

This number is decreased by one on each invocation of *doGet()*, that is, 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 chosen workload. 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.

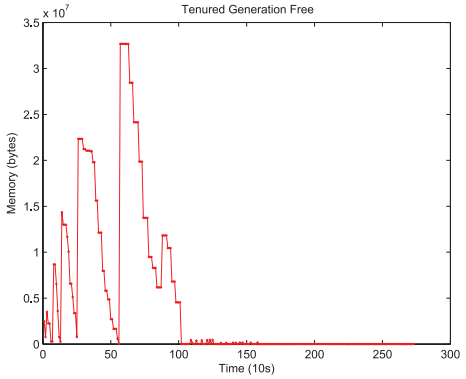
The Java heap memory is divided into three main zones: Young, Old, and Permanent. A Java object is created in one of these heap zones and is garbage-collected after there are no references to it. Resource behavior can look quite different depending on the adopted monitoring strategy. Memory usage by a Java application looks 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



(a) Young + Old heaps used



(b) Runtime memory used



(c) Old heap memory free

Fig. 5. JVM memory exhaustion.

JVM, then we can obtain accurate measures. 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 [Oracle 2010].

We use a JVM monitoring tool, jmap [Oracle 2010], to collect the JVM memory exhaustion data. A script written in perl on the server invokes jmap periodically, and then obtains the memory usages of Young, Old and Permanent. We collect Young plus Old used memory space at an interval of ten seconds. The Young plus Old memory used space as well as capacity when N equals 7, is shown in Figure 5(a), and Old memory free is shown in Figure 5(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. Runtime memory used is shown in Figure 5(b). In Figure 5(a), it can be seen that the Young plus Old memory of JVM is used up, since it is close to both heaps' maximal capacity, while in Figure 5(b) we see that the runtime memory used is close to the JVM capacity. The Young plus Old capacity is shown in

Table I. Memory Consumption Rate and N

Memory consumption rate (kB/s)	N	Memory consumption rate per replication
0.0124		*normal level*
149.61	4	146.86, 149.22, 149.54, 157.04, 136.2, 153.44, 154.94
82.518	8	64.892, 84.042, 90.645, 88.752, 82.509, 85.847, 80.943
58.132	12	55.331, 58.284, 56.496, 59.213, 56.084, 55.148, 66.368
47.321	16	43.256, 48.649, 46.918, 50.081, 48.227, 44.348, 49.768

Figure 5(a), and from Figure 5(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.

4. ANALYSIS OF EXPERIMENTAL RESULTS

In our experiment, we use four acceleration levels (S1 to S4) with N equals 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 Matias et al. [2010a] to calculate the sample size, n_{ALT}, thus verifying whether the initial number of samples satisfies the criteria of statistical analysis used in the ALT method.

Based on the experimental 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 [Sen 1968]. 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 equal to 100 EBs, the total experimental time is 398790 seconds, or 4.615625 days. The memory consumption rate of Young and Old heaps is approximately 0.0124kB/s, based on the Sen's slope estimation method (see Figure 6).

We may use two approaches to calculate the TTFs. The Young plus Old memory free space which is obtained from the same data set in Figure 5(a), is shown in Figure 7. The developed algorithm calculate the TTFs by using the Young plus Old memory free shown in Figure 7. It searches the TTF point from the last sample, and ends when the difference between the last sample and the current sample reaches the threshold which is a fixed value, so that the TTF point can be found. But, from Figure 7, it can be seen that when the Young plus Old memory is exhausted, the JVM usage may fluctuate irregularly due to complex generation memory management and garbage collection (GC) behavior, so that it is not easy to define the exact TTF point. Hence, using this approach of calculating TTF samples may lead to inaccuracy of locating the TTF point.

Then we use the runtime memory usage data in Figure 8 to calculate the TTFs. The memory usage data from the first injection point to the last successful injection point in Figure 8 is from the same data set in Figure 5(b). It can be seen in Figure 8 when the last injection behavior occurs, the free memory is nearly exhausted. But it can be observed that the server can still provide the service such as the response to the Home page, which requires only a small quantity of memory. Because of GC, a little remaining memory can be used to serve quite a few normal requests. After this moment when the memory injection behavior occurs again, this injection will fail, and it can be observed that the server stops responding to EBs' requests. This can be explained

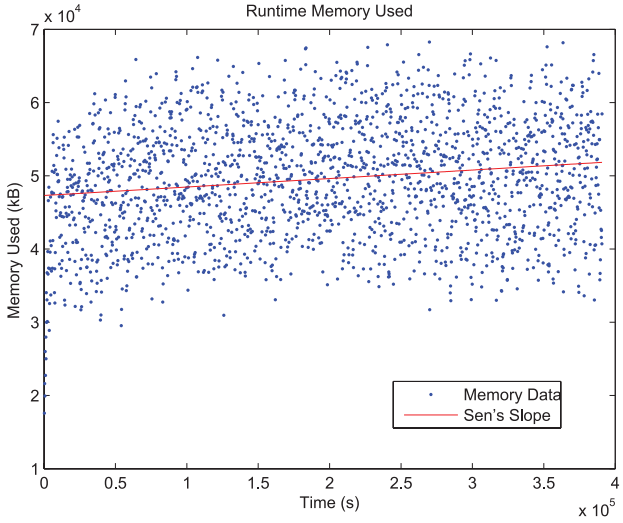


Fig. 6. Memory consumption rate of Young plus Old heaps calculated using Sen's slope method.

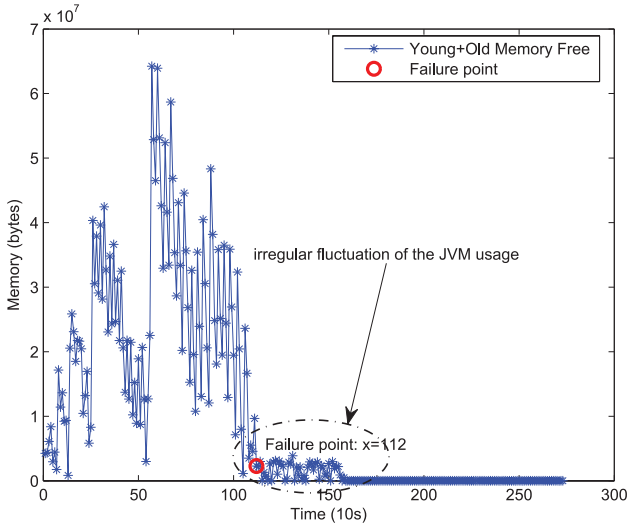


Fig. 7. The algorithm to locate the TTF point of TTF.

that the server fails due to the insufficient memory. The previously mentioned interval is the time between the last successful injection to the failed injection as shown in Figure 8, and we define this point as the TTF point. As a result, the TTF is the total time for the three phases as shown in Figure 8, in which the first phase is from the beginning of the experiment to the first injection point, the second phase is the time between the first injection point to the last successful injection point, and the third phase is from the last successful injection point to the failed injection point.

Our experiment recorded detailed injection point time that has a millisecond precision as well as the random numbers between 0 to N when the injection behavior occurs. Note that the number of visiting Search Request Page equals the corresponding recorded random number. We made a statistics analysis of the time interval between

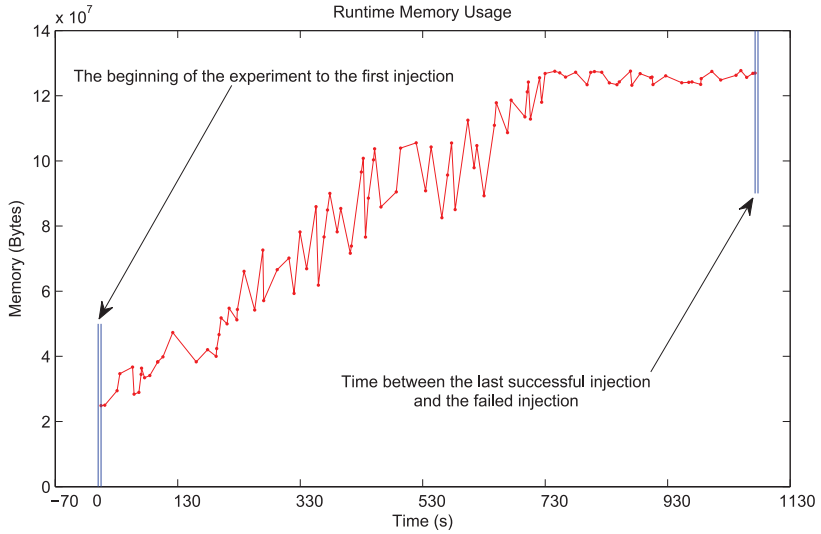


Fig. 8. Three phases of TTF sample.

Table II. Sample of Failure Times (seconds)

TTF(S1)	TTF(S2)	TTF(S3)	TTF(S4)
731.101	1322.731	1769.281	2071.165
737.962	1342.926	1770.842	2188.907
738.284	1360.289	1898.677	2202.236
755.586	1382.329	1994.836	2266.199
764.761	1394.484	2014.693	2443.743
766.159	1419.333	2018.450	2558.261
862.257	1476.640	2098.482	2609.083

visiting Search Request Page, $T_{interval}$, and obtained the mean value of $T_{interval}$ that equals 4.7489s, with its 90% CI (confidence interval) being (4.5460, 4.9518)s. If we use m and n to represent the number of visiting Search Request Page of the first phase as well as the third phase respectively, and the time durations of the second phase is T_{second}, the total time for the three phases, TTF, can be calculated as $(m+n) \cdot T_{interval} + T_{second}$. The samples of time to failure at each acceleration level are shown in Table II. Each TTF sample is the time from the beginning of a test to the time when the server's memory is exhausted.

The TTF is determined by the memory consumption rate, and the memory consumption rate is affected by two main factors: one is the frequency of visiting Search Request Page, and the other is the memory injection rate. The frequency of visiting Search Request Page depends on the TPC-W specification [TPC 2002], while the memory injection rate is controlled by an integer N. For example, if N equals 4, there are *randomNumber* visits of Search Request Page between two injection points, where *randomNumber* is a random number between 0 to 4. Hence, the amount of change of the memory injection rate is small at the same value of N, also, at each stress level, the memory injection is the main cause of memory consumption, so the values of the memory consumption rate have small changes among replications. Therefore, at each stress level, the variance of TTF data is low.

The next step is to select the Lifetime distribution. The most used probability distributions in ALT experiments are from the location-scale family [Meeker and Escobar 1998]. Examples of distributions from this family are Normal, Weibull, Lognormal,

Table III. Results of Model Fitting for Accelerated Failure Times

Accelerated level	Model	Lk	Best-fit Ranking
4(S1)	Lognormal	−35.7755	1st
	Weibull	−37.5017	2nd
	Exponential	−53.4806	3rd
8(S2)	Lognormal	−36.9586	1st
	Weibull	−37.7957	2nd
	Exponential	−57.6369	3rd
12(S3)	Lognormal	−43.5112	2nd
	Weibull	−43.1158	1st
	Exponential	−59.9855	3rd
16(S4)	Lognormal	−46.6123	1st
	Weibull	−46.8550	2nd
	Exponential	−61.2881	3rd

Logistic, LogLogistic, and Extreme Value distributions. Location-scale distributions have an important property in analyzing data from accelerated life tests, which is related to their cumulative distribution function (cdf). A random variable Y belongs to a location-scale family of distributions if its cdf can be written as:

$$P_r(Y \leq y) = F(y; \mu, \sigma) = \Phi\left(\frac{y-\mu}{\sigma}\right), \tag{3}$$

where $-\infty < \mu < \infty$ is a location parameter, $\sigma > 0$ is a scale parameter, and Φ does not depend on any unknown parameters. Appropriate substitution [Meeker and Escobar 1998] shows that Φ is the cdf of $(Y-\mu)/\sigma$ when $\mu = 0$ and $\sigma = 1$. The importance of this family of distributions for ALT is due to the assumption that the location parameter, in (3), depends on the stress variable, s, that is $\mu(s)$, and the scale parameter, σ, is independent of s. This relationship is shown in (4).

$$Y = \log(t) = \mu(s) + \sigma\varepsilon, \tag{4}$$

where t is the time to failure, and ε is a probabilistic component modeling the time to failure sample variability. Essentially, we have a location-scale regression model to describe the effect that the explanatory variable, s, has on the time to failure. In this work, we evaluate density functions from location-scale family of distributions, and assume their scale parameter approximately constant (within the same CI) across the stress levels. Based on Matias et al. [2010a], we test the probability distributions Lognormal, Weibull and Exponential to identify the best fit. The criterion used to determine the quality of fit is the log-likelihood function (Lk) [Trivedi 2002].

We calculate n_{ALT} to be 25 using the method given in Nelson [2004], which is the minimum number of samples needed for our ALT test plan. Since this number is smaller than the one we used in our experiments, we satisfy the ALT assumptions for the sample size.

The fitting results for Lognormal, Weibull, and Exponential are shown in Table III.

We can see from Table III that both Weibull and Lognormal fitting results are very close. Equations for the SMP model in Section 5 have been worked out for the Weibull case, but not yet for the Lognormal. And since Weibull fitting is nearly as good as Lognormal fitting, we chose Weibull combined with IPL model to create our life-stress relationship. The probability density function (pdf) of Weibull is shown in Eq. (5)

$$f(t) = \frac{\beta}{\eta}\left(\frac{t}{\eta}\right)^{\beta-1} e^{-(\frac{t}{\eta})^{\beta}}, \tag{5}$$

where, $f(t) \geq 0, t \geq 0, \beta \geq 0, \eta \geq 0$, $\eta =$ scale parameter, $\beta =$ shape parameter (or slope).

Table IV. Parameter Estimation of Weibull Model

Accelerated level	Parameter	ML estimate	90% CI Lower	90% CI Upper
S1	η_1	787.015	753.944	821.536
	β_1	15.464	10.071	23.743
S2	η_2	1409.691	1376.925	1443.237
	β_2	28.116	17.881	44.209
S3	η_3	1992.147	1928.448	2057.949
	β_3	20.146	12.138	33.436
S4	η_4	2423.091	2308.777	2543.065
	β_4	13.631	8.368	22.202

Table V. IPL-Weibull Parameter

Parameter	ML Estimate	90% CI Lower	90% CI Upper
β	17.3682	13.7767	21.8959
k	9E-6	8E-6	1.1E-5
w	0.9796	0.9397	1.0195

In ALT, the life-stress relationship is usually based on traditional models such as Arrhenius, Eyiring, Inverse Power, Coffin Manson, etc. [Nelson 2004], which are appropriate to physical or chemical phenomena. However, due to the lack of equivalent models established for software experiments, Matias et al. [2010a] investigated several models and recommended the Inverse Power Law (IPL) for software aging experiments. IPL is a general model applicable to any type of positive stress scenario such as the one we have in our experiment.

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

$$f(t,s) = \beta k s^w (k s^w t)^{\beta-1} e^{-(k s^w t)^\beta} \tag{6}$$

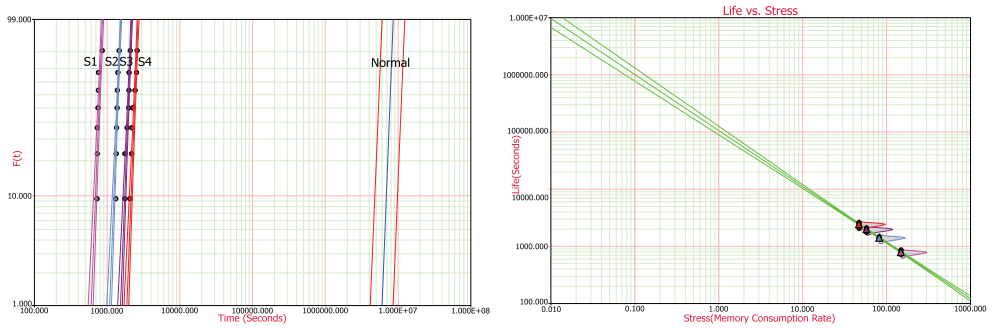
This is a three-parameter model. Thus, (7) can be directly derived from (6) and used to estimate the mean time to failure, MTTF, of the SUT at a specific use rate.

$$MTTF(s) = \frac{1}{k s^w} \cdot \Gamma\left(\frac{1}{\beta} + 1\right) \tag{7}$$

where, Γ is the Gamma function [Trivedi 2002].

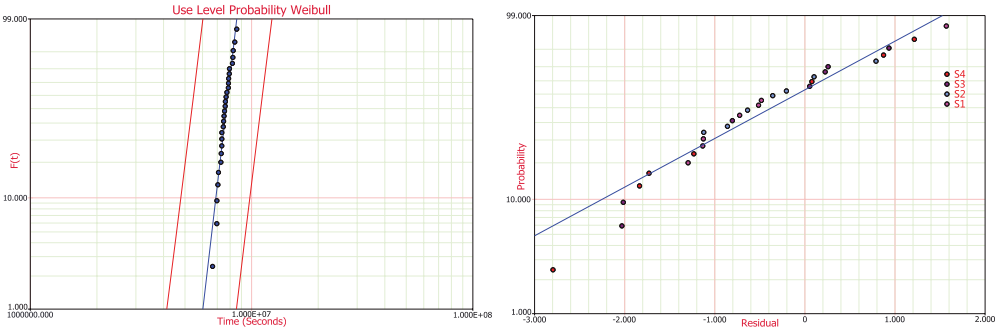
According to Meeker and Escobar [1998], the Weibull distribution may adopt the same parametrization structure shown in (3), where $\sigma = 1/\beta$ is the scale parameter, and $\mu = \log(\eta)$ is the location parameter. Hence, the assumption of same scale parameter across the stress levels must be evaluated on the estimated values of β after fitting the Weibull model to the four samples of failure times. We verified that the four beta values are inside the CI calculated for each sample, and their intervals satisfy the assumption of scale invariance. Table IV presents the estimates for Weibull parameters, obtained through the maximum likelihood (ML) parameter estimation method [Nelson 2004].

The estimated IPL-Weibull parameters are listed in Table V. We use Eq. (7) to estimate the MTTF and pdf for the normal condition. Figure 9(a) presents the fit of the Weibull model estimated from the four stress levels, and for the normal condition. Figure 9(b) shows the failure and stress relationships. The calculated estimate for time to failure (x-axis) at the normal level starts at 0.0124. We obtain the MTTF at normal level of 7.6115E+6 seconds, or 126858 minutes, with its 90% CI being (5.3730E+6, 1.0782E+7) seconds, that is, (89550, 179700) minutes.



(a) The IPL-Weibull model's ML estimates and 90% CIs for $F(t)$ at normal level (b) Failure and stress levels by the IPL-Weibull model estimation

Fig. 9. IPL-Weibull model estimation.



(a) Probability plot for the normal condition level (b) Standardized residuals

Fig. 10. Goodness of fit of the estimated IPL-Weibull model.

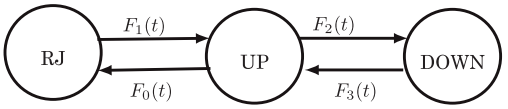


Fig. 11. Rejuvenation model.

Figure 10(a) presents the probability plot for the normal condition level, and the standardized residuals plot is shown in Figure 10(b), which confirms the good fit of the estimated model.

From the analyses of these results, we can see the experimental costs to obtain TTFs and the related metrics (e.g., MTTF, pdf, etc.) are greatly reduced due to the ALT approach. 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.

5. OPTIMAL SOFTWARE REJUVENATION

Based on the results discussed in Section 4, we employ the preventive maintenance model presented in Chen and Trivedi [2001], 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 11 shows this model.

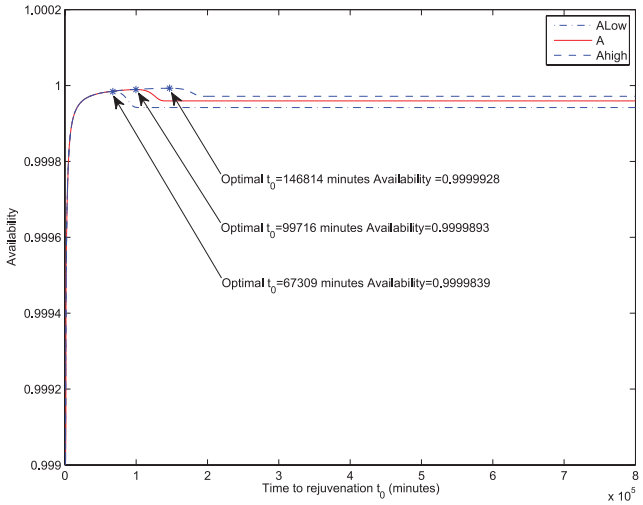


Fig. 12. Steady-state availability *vs.* time to rejuvenation t_0.

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 [Trivedi 2002].

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 CDF (Cumulative distribution function) of the time to failure distribution is assumed to be Weibull and given by:

$$F_2(t) = 1 - e^{-(\frac{t}{\eta})^{\beta}}. \tag{8}$$

The sojourn time in UP state is then given by:

$$h_0 = \int_0^{t_0} (1 - F_2(t))\, dt = \frac{\eta}{\beta}\Gamma\left(\frac{1}{\beta}\right) G\left(\frac{1}{\eta^{\beta}} t_0^{\beta}, \frac{1}{\beta}\right), \tag{9}$$

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:

$$A_{weib} = \frac{h_0}{h_0 + (1 - F_2(t_0))t_1 + F_2(t_0)t_2}. \tag{10}$$

Therefore, from (10), we derive the steady state availability $A = A_{weib}(\eta, \beta)$ is computed using the point estimates of the Weibull parameters from the experiments. Similarly, the CI $A_{low} = A_{weib}(\eta_{low}, \beta_{low})$, $A_{high} = A_{weib}(\eta_{high}, \beta_{high})$ is computed from the experimental CIs of Weibull parameters. 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 12. The optimal time to

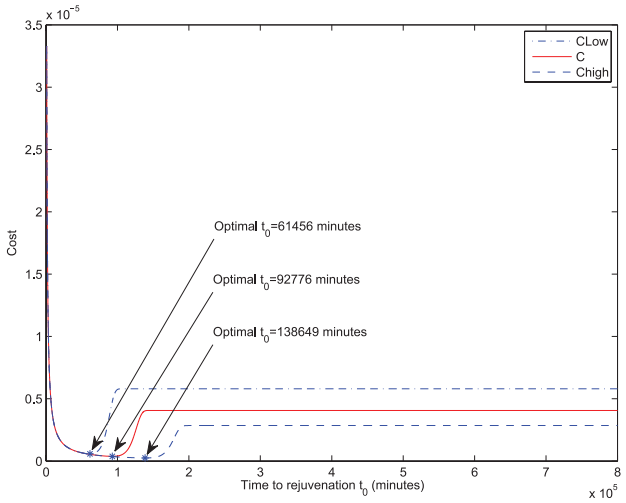


Fig. 13. Average cost *vs* time to rejuvenation t_0.

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 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

$$C = C_f\pi_2 + C'_f\pi_2/t_2 + C_p\pi_1 + C'_p\pi_1/t_1, \tag{11}$$

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 13. The optimal intervals for the cost models are as short as 61456 to 138649 minutes, while the optimal intervals for the availability models are 67309 to 146814 minutes.

6. SIMULATION APPROACH IN ALT

We develop a discrete-event simulation program using C++ to simulate the TPC-W experiment in Section 3, so as to cross validate the statistical results derived from the aforementioned IPL-Weibull models in Section 4. And we also estimate the steady-state availability from the simulation model with rejuvenation to cross-validate the results of the semi-Markov availability model of the previous section. Figure 14 shows the queueing model of the system. In this simulation model, each client represents one EB in our experiment, and simulates the EB's actions of sending requests and receiving responses. Following the experimental setup presented in Section 3, we select the workload to be 100 clients. The arriving requests while the server is busy are placed into the queue. Considering that the server in our experiment has only one CPU, we configure the server in the simulation program to work at a FCFS (First-come, First-served) mode, and the capacity of the queue is set to the sum of Tomcat's buffer length (acceptCount = 100) and max number of threads (maxThreads = 200) [Apache 2011c].

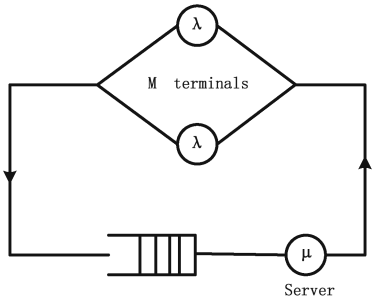


Fig. 14. Queueing model of the simulation program.

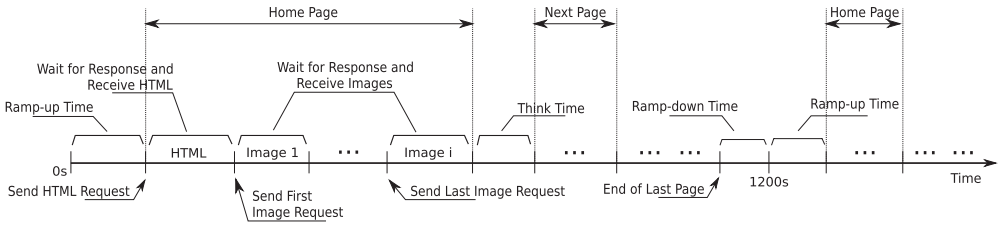


Fig. 15. Work flow of one client in the simulation program.

The TPC-W benchmark defines three distinct mixes of web interactions: Browsing, Shopping and Ordering. The term, *web interaction*, refers to a complete process of requesting for one of the 14 different pages in the e-commerce web site. It includes one or more HTTP requests for HTML documents, image files or other web objects. Each EB or client starts requesting the Home Page, and then randomly selects the next navigation option according to the current mix of web interactions [TPC 2002]. We select the Shopping mix in our experiment, so that is used in our simulation. We configure each client to request one or more web interactions, starting with the Home web interaction and ending as defined in TPC-W specification [TPC 2002], and each EB's working cycle is preceded by a Ramp-up Time, and ended by a Ramp-down Time of 1200 seconds, as shown in Figure 15. A web interaction in the TPC-W experiment can be divided into two phases. In the first phase, EB sends an HTTP request to the server asking for the HTML document, and then parses the HTML code to get the URLs of other web objects of image files. In the second phase, EB sends HTTP requests for these image files. A web interaction completes when all of the image files are received or a timeout occurs. The quantities of image files on different pages may be different, so are the number of HTTP requests the client sends for different pages. When a web interaction finishes, EB enters a "think" phase. "Think Time" is taken from a negative exponential distribution, which is generated by Eq. (12), where r is a random number from a uniform distribution such that $0 < r < 1$, and t is the mean think time [TPC 2002]. We have used t as 7.0 seconds in both the simulation and experimental settings

$$T = -\ln(r) \cdot t. \tag{12}$$

The simulation model is detailed with individual client request generation and processing, memory leak injection, ALT and rejuvenation. Our simulation model has the following requirements.

(1) It must simulate each EB request generation and processing;
(2) It must simulate each EB's visiting 14 pages following the TPC-W Specification;

(3) It must simulate the memory leak injection at different accelerated level as well as normal use level;
(4) It must simulate the full GC represented in the experimental settings;
(5) It must simulate the rejuvenation in the SMP model.

We measured the response time of all the HTML documents and image files in the experiment under a non-queuing condition, in which there is only one EB so that there exists at most one HTTP request being processed by the server. For each HTTP request, the response time is the time measured when EB received the last byte of the response minus the time when EB sent this request. In our simulation program, we use the measured mean response time of this non-queuing case as the mean service time of each request. We test the EB requesting HTML file and image files of each page, and 14 pages in total, to get the service time distribution of each request in a page. We made three test runs under the same experimental setup as well as the same interval of EB's running one session, and we obtain 2 to 13 service time samples for different pages caused by different visiting probability. For example, the probability of EB visiting the Search Request Page is higher than probabilities of visiting other pages according to TPC-W specification, so we may get more samples for visiting Search Request Page. We did not obtain the service time of the Admin Confirm Page due to its extremely low visiting probability, and the Admin Confirm Page's service time is calculated by a similar attribute page of Admin Request Page. In our simulation model, the service time distribution of different requests for a page is deterministic represented as the sample mean of each request type.

In order to characterize the memory leak behavior, we need to test the parameters related to memory. Four parameters are used to describe the heap memory status of the server: the memory capacity, the system reserved memory, the current memory usage, and the amount of memory that can be reclaimed by the garbage collector. In the current simulation settings, the first two parameters are fixed values, while the last two are variables, which change as the simulation goes forward. The memory capacity is 127729664 bytes, which is obtained from the Young plus Old heap memory of Tomcat in our experiment. The memory capacity does not consist of the Permanent zone because the usage and the capacity of this zone is substantially unchanged during our experiment, thus it does not affect the injection behavior in the simulation model. The system reserved memory is used to keep the Tomcat critical components running, such as the connector and the servlet container systems, and it is not allowed to be injected as well as collected by garbage collector. We observe 93 test runs of ALT of our experiments to obtain 93 injection times samples, using the test bed in Section 3.1, with different values of N. We obtain the sample mean of the memory injections of 101 when the server failed. So we calculate the reserved memory, which equals the memory capacity minus the injected memory, that is, 21823488 bytes.

Figure 16 shows the memory usage and GCs in the first 30,000 seconds of our nonaccelerated experiment, in which 100 EBs are used. We observe that the full GC runs approximately every one hour in our experiment. However, the Young GCs show no obvious regularity. Moreover, we measure the memory usage by combining the Young and old heap memory together, as a result we omit the Young GC in the simulation. When processing requests, the server need to consume heap memory to create temporary objects, which will become recyclable if there exist no references to them, thus the temporary objects will be collected by one of the above GCs. We measured the memory consumption rate of Young plus Old heap memory (denoted as v) between two full GCs, such as the time intervals a, b and c in Figure 16. In the simulation, which is calculated by Sen's slope as $v = 2462.9$ bytes/s. Accordingly, the memory consumed by processing one request is defined as $(t_j - t_i) \cdot v$, where t_j is the time when the current request

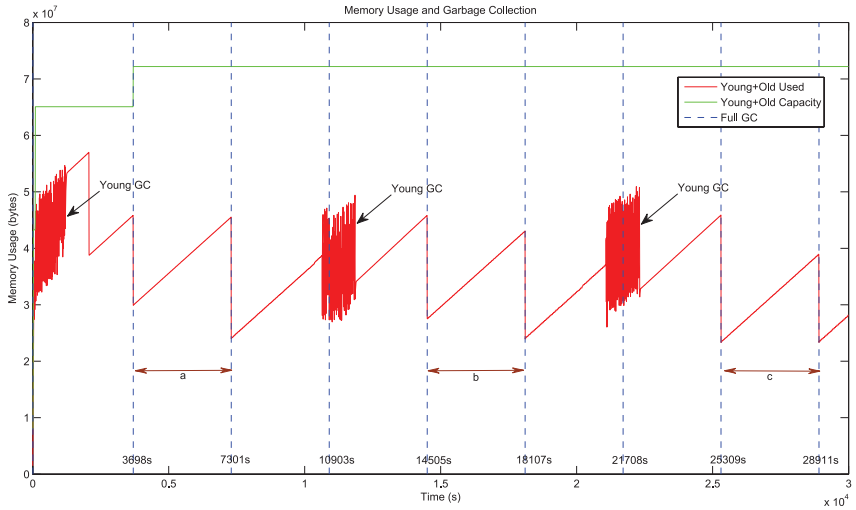


Fig. 16. Memory usage and GC.

Table VI. TTFs of Different Accelerated Levels in Simulation (seconds)

TTF(S1)	TTF(S2)	TTF(S3)	TTF(S4)
639.25	1009.68	1498.46	1976.61
597.71	971.06	1512.30	1822.65
664.98	974.30	1440.09	2010.60
609.22	1105.74	1435.68	1914.24
590.30	948.04	1385.46	2072.92
585.29	966.59	1456.35	2096.28
588.89	993.46	1356.51	1984.33

finished, and t_i is the time when the last request finished. Therefore, the amount of required memory due to process request as well as the recyclable actions after the request is processed by full GC can be simulated. The memory injection method in the simulation is the same as the one we described in Section 3.2. Depending on a *randomNumber* between 0 and N, we randomly inject about 1-megabyte of memory leak to the server when the Search Request Page is processed. The injected memory will not be added to the recyclable memory, so it will never be collected by the garbage collector. When the unrecyclable memory reaches the capacity of the server, the server fails, and it must be restarted to get back to work again.

Our simulation program first cross-validate against the ALT experimental results. We carried out seven repeated simulation experiments at acceleration levels from S1 to S4, where N equals to 4, 8, 12, and 16, respectively. From the simulation, we obtained the TTF samples and the memory consumption rate in each replication. The TTFs are shown in Table VI, and the memory consumption rates are shown in Table VII. Our simulation employs the different N values to simulate the ALT experiment. Since the N value can not control the memory consumption rate obtained from the experiment precisely, the TTFs of the ALT experiment as well as the simulated are different. In both the ALT experiment and the simulation, and at each stress level, the production of the MTTF and the mean memory consumption rate equals the system memory usage. We calculated the production of MTTF and the mean memory consumption rate at each stress level in the experiment as well as in the simulation, respectively.

Table VII. Memory Consumption Rate and N in Simulation

Memory consumption rate (kB/s)	N	Memory consumption rate per replication
191.10	4	186.42, 190.46, 175.86, 190.67, 189.26, 206.46, 198.54
111.92	8	115.49, 113.10, 116.79, 96.23, 113.97, 117.53, 110.32
76.09	12	74.01, 71.03, 73.67, 80.51, 81.94, 74.47, 77.05
55.04	16	55.13, 59.59, 54.61, 55.13, 54.93, 52.21, 53.70

Table VIII. Simulation Results in User Level ($N = 69000$)

Memory consumption rate (kB/s)	Time to failure (s)
0.0122509	8051540
0.0123884	7648960
0.0153444	7081620
0.0138045	8350660
0.0146280	8146790
0.0146254	7334810
0.0121648	8391360
0.0155159	7061190
0.0135798	7485940
0.0129997	6714810
0.0145606	7208990
0.0123750	7891720
0.0124154	6904560
0.0133905	7480030

In the experiment, the productions are 114475.374 kB at S1, 114331.424 kB at S2, 112653.679 kB at S3 and 110457.990 kB at S4. In the simulation, they are 116724.699 kB at S1, 111421.956 kB at S2, 109622.320 kB at S3 and 109117.822 kB at S4. It can be seen that the system memory usages in accelerated levels from simulation and ALT are all close.

Accordingly, we carried out 14 repeated simulation when $N = 69000$, while the memory consumption rate is close to that obtained from the normal level in our experimental testbed (Section 4). From the simulation results in Table VIII, we obtain the sample mean of memory consumption rate of 0.013575 kB/s, with its 90% CI being (0.013031, 0.014118) kB/s, and also we obtain the MTTF of 7.553784E+6 seconds, with its 90% CI being (7.306314E+6, 7.801254E+6) seconds, or (1.217719E+5, 1.300209E+5) minutes. This results of memory consumption rate are all calculated with the Sen's slope estimation method. We define the relative error RE, at normal level, between the MTTF from the simulation, $MTTF_{sim}$, and IPL-Weibull in Section 4, $MTTF_{ALT}$, as $RE = |(MTTF_{sim} - MTTF_{ALT})|/MTTF_{sim}$, which equals 0.764%. It can be seen that the 90% CI of MTTF at normal level obtained from the simulation results falls into that of obtained from the experimental results, thus we consider that the simulation model represents well the experimental testbed.

Table IX. Availability Results of the Simulation Model

Rejuvenation Trigger (Seconds)	Availability	90% CI	
		Lower	Upper
4038540	0.99998175	0.99997937	0.99998386
5982960	0.99998669	0.99998464	0.99998847
7306314	0.99998536	0.99998236	0.99998786
7553784	0.99997974	0.99997559	0.99998319
7801254	0.99997638	0.99997151	0.99998043
8400000	0.99996446	0.99995690	0.99997069
8808840	0.99999026	0.99998848	0.99999177
9600000	0.99996115	0.99995295	0.99996792
10800000	0.99996105	0.99995286	0.99996782

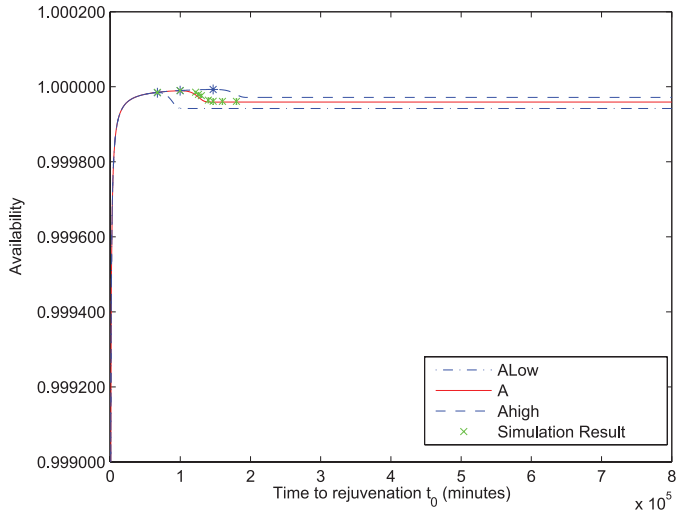


Fig. 17. Availability comparisons between semi-Markov model and simulation model.

Next, we introduce both reactive repair and rejuvenation in our simulation program to cross-validate the availability computed from the semi-Markov model. We take nine different rejuvenation trigger intervals in our simulation model with reactive repair at non-accelerated level, and use 12 replications at each rejuvenation trigger interval of 4038540, 5982960, 7306314, 7553784, 7801254, 8400000, 8808840, 9600000, and 10800000 seconds, among which 5982960 is the optimal trigger interval obtained from the semi-Markov model.

Then we estimate the system availability at each rejuvenation trigger and the 90% CI applying the F-distribution from the simulation model with reactive repair, as shown in Table IX. We plot the simulation estimated availability results against the semi-Markov model, in Figure 17. Furthermore, we show nine discrete availability points as well as their 90% CIs and a zoom-in in Figure 18. From these figures we can see that the simulation availability results have a reasonably good match with those from the semi-Markov model.

7. CONCLUSION

In this article, we obtain the accelerated life test results by injecting memory leaks in an ALT experiment for an on-line bookstore subject to aging due to memory leaks. We develop a detailed simulation model to cross validate the MTTF at nonaccelerated

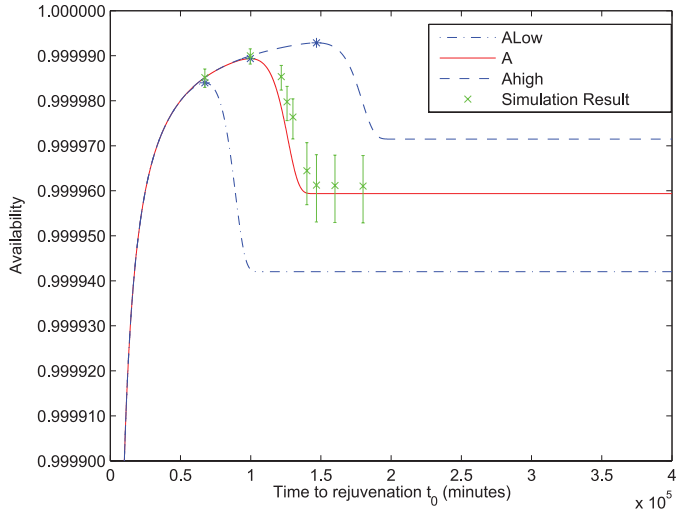


Fig. 18. Zoomed: Availability comparisons between semi-Markov model and simulation model.

level. Given that the measurements in experimental testbed with rejuvenation would take inordinate time to estimate system availability, the simulation model is used to estimate availability. In a semi-Markov process, we use Weibull time to failure distribution to compute availability and cross-validate with the results obtained from the simulation model. The semi-Markov model is then used to optimize the software rejuvenation trigger interval so as to maximize the availability or minimize the operational cost.

In a future paper, we will construct the measurement experiment with rejuvenation to obtain the availability vs optimal rejuvenation trigger interval, to cross-validate against the SMP model. Second, other than the memory leak injections in ALT, we will construct the experiment of injecting the memory fragmentation using ALT to study the application failure and optimal rejuvenation trigger interval. Finally, we will use importance sampling technique for efficient simulation.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees for their helpful comments.

REFERENCES

Alonso, J., Silva, L., Andrzejak, A., Silva, P., and Torres, J. 2007. High-available grid services through the use of virtualized clustering. In *Proceedings of the 8th IEEE/ACM International Conference on Grid Computing*. IEEE, 34–41.

Alonso, J., Torres, J., Berral, J. Ll., and Gavaldà, R. 2010. Adaptive on-line software aging prediction based on machine learning. In *Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*. IEEE, 507–516.

Apache. 2011a. The Apache HTTP Server Project. (2011). http://httpd.apache.org.

Apache. 2011b. Apache tomcat. (2011). http://tomcat.apache.org/.

Apache. 2011c. Apache tomcat configuration reference - the HTTP connector. (2011). http://tomcat.apache.org/tomcat-6.0-doc/config/http.html.

Bao, Y., Sun, X., and Trivedi, K. S. 2005. A workload-based analysis of software aging, and rejuvenation. *IEEE Trans. Reliab. 54*, 3, 541–548.

Bezenek, T., Cain, T., Dickson, R., Heil, T., Martin, M., McCurdy, C., Rajwar, R., Weglarz, E., Zilles, C., and Lipasti, M. 2011. TPC-W benchmark Java version. http://pharm.ece.wisc.edu/tpcw.shtml.

Bobbio, A., Sereno, M., and Anglano, C. 2001. Fine grained software degradation models for optimal rejuvenation policies. *Perform. Eval. 46*, 1, 45–62.

Cai, K.-Y. 2006. Software reliability experimentation and control. *J. Comput. Sci. Tech. 21*, 5, 697–707.

Cassidy, K. J., Gross, K. C., and Malekpour, A. 2002. Advanced pattern recognition for detection of complex software aging phenomena in online transaction processing servers. In *Proceedings of the International Conference on Dependable Systems and Networks*. IEEE, 478–482.

Castelli, V., Harper, R. E., Heidelberger, P., Hunter, S. W., Trivedi, K. S., Vaidyanathan, K., and Zeggert, W. P. 2001. Proactive management of software aging. *IBM J. Res. Development 45*, 2, 311–332.

Chen, D. and Trivedi, K. S. 2001. Analysis of periodic preventive maintenance with general system failure distribution. In *Proceedings of the 2001 Pacific Rim International Symposium on Dependable Computing*. IEEE, 103–107.

Dohi, T., Goseva-Popstojanova, K., and Trivedi, K. S. 2000. Statistical non-parametric algorithms to estimate the optimal software rejuvenation schedule. In *Proceedings of the 2000 Pacific Rim International Symposium on Dependable Computing*. IEEE, 77–84.

Garg, S., Puliafito, A., Telek, M., and Trivedi, K. S. 1998a. Analysis of preventive maintenance in transactions based software systems. *IEEE Trans. Comput. 47*, 1, 96–107.

Garg, S., van Moorsel, A., Vaidyanathan, K., and Trivedi, K. S. 1998b. A methodology for detection and estimation of software aging. In *Proceedings of the 9th International Symposium on Software Reliability Engineering*. IEEE, 283–292.

Grottke, M., Li, L., Vaidyanathan, K., and Trivedi, K. S. 2006. Analysis of software aging in a web server. *IEEE Trans. Reliab. 55*, 3, 411–420.

Grottke, M., Matias, R., and Trivedi, K. S. 2008. The fundamentals of software aging. In *Proceedings of the 2008 IEEE First International Workshop on Software Aging and Rejuvenation (WoSAR)*. IEEE, 1–6.

Grottke, M., Nikora, A. P., and Trivedi, K. S. 2010. An empirical investigation of fault types in space mission system software. In *Proceedings of the 2010 IEEE / IFIP International Conference on Dependable Systems & Networks (DSN)*. IEEE, 447–456.

Huang, Y., Kintala, C., Kolettis, N., and Fulton, N. D. 1995. Software rejuvenation: Analysis, module and applications. In *Proceedings of the 25th Symposium on Fault Tolerant Computing*. IEEE, 381–390.

Jia, Y.-F., Zhao, L., and Cai, K.-Y. 2008. A nonlinear approach to modeling of software aging in a web server. In *Proceedings of the 15th Asia-Pacific Software Engineering Conference*. IEEE, 77–84.

Macêdo, A., Ferreira, T., and Matias, R. 2010. The mechanics of memory-related software aging. In *Proceedings of the 2010 IEEE Second International Workshop on Software Aging and Rejuvenation (WoSAR)*. IEEE, 1–5.

Marshall, E. 1992. Fatal error: How patriot overlooked a scud. *Science (New York, NY) 255*, 5050, 1347.

Matias, R., Barbetta, P. A., Trivedi, K. S., and Filho, P. J. F. 2010a. Accelerated degradation tests applied to software aging experiments. *IEEE Trans. Reliab. 59*, 1, 102–114.

Matias, R., Beicker, I., Leitão, B., and Maciel, P. R. M. 2010b. Measuring software aging effects through OS kernel instrumentation. In *Proceedings of the 2010 IEEE Second International Workshop on Software Aging and Rejuvenation (WoSAR)*. IEEE, 1–6.

Matias, R., Trivedi, K. S., and Maciel, P. R. M. 2010c. Using accelerated life tests to estimate time to software aging failure. In *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 211–219.

Meeker, W. Q. and Escobar, L. A. 1998. *Statistical Methods for Reliability Data*. Vol. 78. Wiley, New York.

Nelson, W. B. 2004. *Accelerated Testing: Statistical Models, Test Plans and Data Analyses*. Wiley-Interscience.

Oracle. 2010. JSTAT - Java virtual machine statistics monitoring tool. (2010). http://download.oracle.com/javase/1.5.0/docs/tooldocs/share/jstat.html.

Sen, P. K. 1968. Estimates of the regression coefficient based on Kendall's tau. *J. Amer. Statist. Assoc. 63*, 324, 1379–1389.

Silva, L., Madeira, H., and Silva, J. G. 2006. Software aging and rejuvenation in a SOAP-based server. In *Proceedings of the 5th IEEE International Symposium on Network Computing and Applications*. IEEE, 56–65.

TPC. 2002. TPC Benchmark™ W specification version 1.8. (2002). http://www.tpc.org/tpcw/spec/tpcw_V1.8.pdf.

Trivedi, K. S. 2002. *Probability and statistics with reliability, queuing, and computer science applications* (2nd ed.), Wiley, New York.

Vaidyanathan, K. and Trivedi, K. S. 2005. A comprehensive model for software rejuvenation. *IEEE Trans. Depend. Secure Comput. 2*, 2, 124–137.

ZHANG, X. AND PHAM, H. 2002. Predicting operational software availability and its applications to telecommunication systems. *Int. J. Syst. Sci. 33*, 11, 923–930.

ZHAO, J., JIN, Y., TRIVEDI, K. S., AND MATIAS, R. 2011. Injecting memory leak to accelerate software failures. In *Proceedings of the 2011 IEEE 22st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 260–269.

Received April 2012; revised September 2012 and December 2012; accepted March 2013