Reliability Engineering and System Safety 133 (2015) 184–191

Contents lists available at ScienceDirect

Reliability Engineering and System Safety

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

Modeling and optimizing periodically inspected software rejuvenation policy based on geometric sequences

Haining Meng [a,b,*], Jianjun Liu [c], Xinhong Hei [a]

[a] *School of Computer Science and Engineering, Xi'an Technology University, Xi'an 710048, China*
[b] *Shaanxi Key Laboratory for Network Computing and Security Technology, Xi'an 710048, China*
[c] *Aeronautics Computing Technique Research Institute, Xi'an 710068, China*

ARTICLE INFO

Article history:
Received 18 May 2013
Received in revised form
4 August 2014
Accepted 1 September 2014
Available online 16 September 2014

Keywords:
Software aging
Software rejuvenation
Inspection
Geometric sequence
Cost rate
System availability

ABSTRACT

Software aging is characterized by an increasing failure rate, progressive performance degradation and even a sudden crash in a long-running software system. Software rejuvenation is an effective method to counteract software aging. A periodically inspected rejuvenation policy for software systems is studied. The consecutive inspection intervals are assumed to be a decreasing geometric sequence, and upon the inspection times of software system and its failure features, software rejuvenation or system recovery is performed. The system availability function and cost rate function are obtained, and the optimal inspection time and rejuvenation interval are both derived to maximize system availability and minimize cost rate. Then, boundary conditions of the optimal rejuvenation policy are deduced. Finally, the numeric experiment result shows the effectiveness of the proposed policy. Further compared with the existing software rejuvenation policy, the new policy has higher system availability.

1. Introduction

With the explosive growth in internet technology and the emergence of a number of new and advanced applications, assured stable and reliable operation and availability of software system has become a critical issue. The challenge is to provide the desired availability and performance at a low cost. The reliability studies have reported software aging phenomenon in the long-running software system [1], which leads to the increasing failure rate or performance degradation of a system during execution, and eventually to the system hanging or crashing. Software aging has been observed in many kinds of long-running systems, ranging from business-oriented to highly critical systems, such as telecommunication switching and billing software [2], networked UNIX workstations [3], OLTP DBMS servers [4], Apache web server [5], Sun Hotspot JVM [6], spacecraft flight systems [7], and the cloud computing infrastructure [8]. Software aging could cause great losses in the safety-critical systems, including the loss of human lives [9].

Software aging can be attributed to the activation and propagation of the software faults called aging-related bugs [8], which are transient, non-deterministic and difficult to characterize. These faults do not immediately cause a software failure when triggered, but manifest themselves as memory consumption, unreleased file locks, data corruption or numerical error accumulation after a long period of execution, making the system gradually degrade its performance and eventually fail. Such faults are too subtle or too costly to be removed during software development and testing. Thus, even if software may have been thoroughly tested, it still may have some design faults that are yet to be revealed in practice [10].

Since software aging leads to performance degradation and sudden failures, a lack of proper software maintenance technique will inevitably cause serious economic losses and system downtime. Apart from reactive methods such as system reboot, a proactive and preventive software maintenance technique to counteract software aging is software rejuvenation [1], which involves occasional stopping of software system, removal of error conditions and system restarting in a clean environment. This process removes the accumulated errors and frees up operating system resources, thus improving system availability and reliability, and postponing or preventing the unplanned and expensive future system failures.

Nevertheless, software rejuvenation does not solve the root cause of software aging, consequently software aging will continue since system start-up, so that software rejuvenation has to be executed cyclically at predetermined or scheduled time to

* Corresponding author at: 5 Gold Flower South Road, Xi'an, Shaanxi Province 710048, China.
E-mail addresses: mengning2001ji@gmail.com (H. Meng), jjliu_imu@163.com (J. Liu), heixinhong@xaut.edu.cn (X. Hei).

http://dx.doi.org/10.1016/j.ress.2014.09.007
0951-8320/© 2014 Elsevier Ltd. All rights reserved.

maintain the robustness of software system. Moreover, software system may be unavailable during rejuvenation which will increase system downtime and incur some costs (e.g., costs due to the loss of business). However, unlike the downtime caused by sudden failures, the downtime related to software rejuvenation can be scheduled at the discretion of the user or administrator, typically during the middle of the night or over weekends. It is likely that the costs of downtime will be high if the downtime is unscheduled, so the cost incurred due to software rejuvenation is less than that incurred due to system failure. Therefore, software rejuvenation can avoid or at least postpone software aging, and reduce the overall downtime and related costs. The most important problem is to plan a cost-effective rejuvenation policy to ensure system reliability, reduce maintenance cost and downtime cost, and improve system availability.

One of the significant issues in the software rejuvenation policy is when and how to trigger it, due to its overhead in processing tasks. Two main approaches to determine the timing for software rejuvenation are time-based and inspection-based methods [11]. The time-based approaches determine the optimal rejuvenation timing by analysis of the state relationship of software system and assumption of system failure distribution. The inspection-based policy is usually conducted by continuous monitoring runtime states and failure behaviors and using data statistical analysis method to estimate the software rejuvenation interval. However, the relatively complex computation process generally leads to the increasing resource exhaustion in software system. Thus, the frequent inspections of runtime software system incur some overhead in terms of downtime and cost. This must be traded off with the downtime and cost due to failures to obtain maximum benefits.

The periodical inspection mechanism has been studied to monitor runtime system in which failures are immediately detected and subsequently repaired. For example, Grall et al. [12] studied the inspection-maintenance strategy for a deteriorating system based on the average long-run cost rate, and the degradation is expressed by the Gamma process. The inspection-based maintenance policy in hardware systems was studied [13], and the inspection time is assumed to be exponentially distributed. Vaidyanathan et al. [14] investigated inspection-based preventive maintenance in operational software system which is inspected at regular intervals, and they obtained the optimal inspection interval to minimize downtime and cost. Ning et al. [15] studied a multi-granularity software rejuvenation policy, supposed the inspection interval follows exponential distribution, and obtained the optimal inspection rate and rejuvenation period by maximizing availability and minimizing cost.

Moreover, the periodical system inspection is difficult to discover system failures immediately. Consider that in the software system subjected to software aging, the probability of occurring failures is lower when system begins to execution, accordingly the time interval between two inspections should be determined longer. With the execution of software system, system performance declines gradually and the possibility of failure occurrence increases over time, so the successive inspection intervals need to be shorter and shorter, i.e. the inter-inspection intervals constitute a decreasing sequence.

Nagel and Skrivan [16] stated that geometric sequences are possible in the well-known software reliability model, i.e. Jelinski-Moranda model. The increasing geometric sequence between failure rates of faults was observed in projects of the communication networks department of the Siemens AG [17]. Inspired by Nagel and Skrivan's idea, we resort to a more complex but more realistic paradigm of the decreasing geometric sequence to describe the decreasing inter-inspection intervals, so that the probability that failure occurred at each inter-inspection interval tends to be a constant p. Then through simulation experiments, the sensitivity analysis of the influence of different parameter p on system availability and maintenance cost rate is evaluated. To the best of our knowledge, this is the first work that geometric sequence formulations applied in software rejuvenation policy.

The rest of this paper is organized as follows. In Section 2, the related works are introduced. Then, in Section 3, the definitions of geometric sequences and geometric series are given. In Section 4, a periodically inspected rejuvenation policy is presented and rejuvenation optimization solution and boundary conditions are obtained by maximizing system availability and minimizing cost rate. In Section 5, numerical results are shown and several examples are provided to illustrate optimal rejuvenation scenarios for different system parameters. Section 6 presents the concluding remarks.

2. Related works

2.1. Software aging

Outages in computer systems consist of both hardware and software failures. It has been determined that software failures lead to more outages than hardware failures [18,19]. Software aging is a phenomenon occurring in long-running software systems, which exhibit an increasing failure rate and lead to progressive resource consumption, performance degradation, and eventually to the system failure, typically because of increasing and unbounded resource consumption, data corruption, and numerical error accumulation. Aging in a software system, as in human beings, is an accumulative process. It is important to highlight that a system fails due to the consequences of aging effects accumulated over time. For example, a given aged application fails due to insufficiency of available physical memory caused by the accumulation of memory leaks.

Resource leaking and other aging effects can be due to aging-related bugs in application software [11]. These bugs are hard to reproduce, even when activated, their manifestation takes long time to become evident, and this makes the testing time insufficient to reveal the problem in most of cases. In addition, most of these aging problems are caused by bad software design or faulty code [20]. However, because software is extremely complex and never wholly free of errors, it is almost impossible to fully test and verify that a piece of software is bug-free. This situation is further exacerbated by the fact that software development tends to be extremely time-to-market-driven, which results in applications which meet the short-term market needs, yet do not account very well for long-term ramifications such as reliability. Hence, residual faults have to be tolerated in the operational phase.

So far, software aging phenomenon has been observed and detected in many computing systems through different algorithms or methods. For example, Garg et al. [3] first proposed a measurement-based method to estimate software aging in networked UNIX workstations by designing and implementing a SNMP-based distributed monitoring tool to collect the operating system resource usage and system activity data. Cassidy et al. [4] adopted the statistical pattern recognition method to detect software aging in large OLTP DBMS servers. Grottke et al. [5] applied the non-parametric statistical method to estimate aging trends in an Apache web server, through collecting the data of used swap space, response time, and free physical memory. Cotroneo et al. [6] revealed the presence of software aging in the Sun Hotspot JVM by adopting both parametric and non-parametric statistical techniques to analyze the trend of throughput loss and memory depletion. Alonso et al. [21] presented a monitoring framework based on aspect-oriented programming to monitor the resource usage of