An Adaptive Reconfiguration Mechanism for Periodic Software Rejuvenation based on Transient Reliability Analysis

Pan He[1,a], Gang Liu[1], Yue Yuan[1]

[1]Chongqing Institute of Green and Intelligent Technology, Chinese Academy of Sciences, Chongqing, China

Abstract. While software rejuvenation is used to prevent severe software failures, existing researches generally choose the constant-value periodic policy through steady-state reliability optimization. Since the software reliability declines with the execution time, a steady policy either introduces extra overhead or could not guarantee the reliability constraints. So an adaptive mechanism is proposed to reconfigure the software rejuvenation in the runtime. The transient reliability analysis is used to choose an optimal rejuvenation policy which maintains the software reliability for a certain period of time. A dynamic time series is generated for the reconfiguration process and the optimal rejuvenation policy is re-calculated according to the reconfiguration intervals during the software execution. Experimental studies results show that as the execution time increases, the software reliability drops continuously and the optimal rejuvenation interval should be decreased to maintain the same reliability constraints. This mechanism guarantees the software reliability constraint by resetting the optimal rejuvenation policy dynamically according to a reconfiguration interval time series.

1 Introduction

Although the software is debugged and tested before deployment, unpredicted errors also occurs at runtime due to the software aging, unnoticed faults and etc[1,2]. Software rejuvenation, involving releasing unused memories, restarting the software, or rebooting the operating system, has been considered as the main solution to avoid severe or aging-related failures[3].After rejuvenation, the software is generally reset to the initial working state. But the whole process interrupts the normal software operation and introduces performance overhead. Considering the regained performance and the introduced overhead, much attention should be devoted to choosing the appropriate rejuvenation policy[4].

A few researches formulated the optimal rejuvenation problem into a Markov decision process and analyzed the optimality of software rejuvenation based on reliability analysis [5-7]. Garg et al.[8] proposed the periodic rejuvenation for the steady-state system availability. Bobbio et al. [9] and Okamura et al. [10] presented risk-level rejuvenation policy, alert threshold rejuvenation policy and workload-based policies[11]. Xie and et al. considered an inspection-based preventive maintenance, introduced a two-level software rejuvenation policy and studied the optimal rejuvenation schedule maximizing the system availability [12,13]. Although a lot of research has been conducted on this area, the optimal rejuvenation policy is often obtained through the steady state analysis and remains unchanged during the long-time software execution. Since the software reliability degrades through the time, one stable rejuvenation policy may not be suitable for degrading software. The transient reliability analysis methods should be used to calculate optimal rejuvenation policy in the runtime. Dohi et al.[14] proposed to calculate optimal periodic software rejuvenation policies based on interval reliability criteria. They demonstrated the change of the rejuvenation interval according to the execution time increase, but did not explain how to update these policies dynamically.

Based on the above researches, this paper aims to propose an adaptive mechanism to reconfigure the periodic software rejuvenation policy dynamically. Based on the Markov process model, the transient reliability of the software with periodic software rejuvenation policy is analyzed. After pre-estimating the software failure-free execution time, a time interval series to reconfigure the rejuvenation policy is first calculated in the runtime, then the optimal rejuvenation policy is calculated at each reconfiguration interval to maintain the software reliability for a certain amount of time. The whole process repeats until software failure occurs. The main difference between this work and the previous work is that the optimal rejuvenation policy does not stay unchanged during the software execution.

The rest of this paper is organized as follows: Section 2 illustrates the periodic software rejuvenation model;

[a]Corresponding author: hepan@cigit.ac.cn

© The Authors, published by EDP Sciences. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (http://creativecommons.org/licenses/by/4.0/).

Section 3 presents the whole adaptive mechanism to reconfigure the periodic policy at specific time; the numerical examples are presented in Section 4 while Section 5 conducts the overall conclusion.

2 Periodic software rejuvenation model

Following the assumptions in [14,15], the software rejuvenation could be modelled as a semi-Markov process, shown in Figure 1.

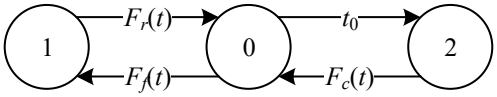

Figure 1. Markov transition diagram for the periodic software rejuvenation.

1. There are three states 0, 1, 2 in this model. State 0 represents the normal operation state, state 1 represents the failure state and state 2 stands for the software rejuvenation state.

2. The transition process from state 0 to state 1 represents the random failure process. The probability distribution function of this process is $F_f(t)$ with finite mean λ.

3. The transition process from state 0 to state 2 stands for the periodic process to trigger rejuvenation. The rejuvenation trigger interval is a constant t_0. The transition process follows the uniform distribution

$$F_a(t) = U(t - t_0) = \begin{cases} 1, & \text{if } t \geq t_0 \\ 0, & \text{else} \end{cases}. \tag{1}$$

4. The transition process from state 2 to state 1 stands for the time to complete the software rejuvenation, the probability distribution function of which is $F_c(t)$ with finite mean μ_c.

5. The transition process from state 1 to state 0 represents the failure recovery process. The probability distribution function of the failure recovery is $F_r(t)$ with finite mean μ.

3 Adaptive reconfiguration mechanism for rejuvenation policy

3.1 Transient reliability analysis

Software reliability measures the probability of failure-free software operation for a specified period of time t. Using the Markov model defined in Section 2, let $\pi_i(t)$ represents the probability of the software in state i(i=0,1,2) at time t. Let $q_{ij}(t)$ denotes the transition rate from state i to state j at time t. The reliability could be calculated as the probability of the software in non-failure states. Although the semi-Markov model could be solved to get the state probability $\pi_i(t)$ using the Laplace transom, it is often hard to take the inversion of the Laplace transforms for general distributions. Thus, following the existing researches[14], it is assumed that the software failure, recovery and rejuvenation processes obey the exponential distributions: $F_f(t)$~EXP(λ), $F_r(t)$~EXP(μ), $F_c(t)$~EXP(μ_c). For an exponential distributed transition process, the transition rate $q_{ij}(t)$ is a constant q_{ij}, which equals to the rate value of the distribution. To simplify the calculation process, one more assumptions is made as follows.

• The rejuvenation trigger process originally follows the uniform distribution, and it could be approximated as an exponential distribution with mean λ_0=2/t_0 as time t increases. Thus, we make the assumption that $F_a(t)$ ~EXP(λ_0)

After that, the whole process is modelled as a continuous-time Markov chain(CTMC), shown in Figure 2.

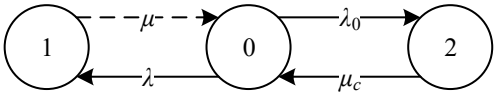

Figure 2. Simplified Markov transition diagram for the periodic software rejuvenation.

According the Markov-chain theory, $\pi_i(t)$ and $q_{ij}(t)$ could be solved through

$$\frac{\partial \pi_i(t)}{\partial t} = (\sum_{j \neq i} q_{ji}(t)\pi_j(t)) - q_{ii}(t)\pi_i(t). \tag{2}$$

The Laplace transform is carried out on (2) as

$$\begin{cases} s\overline{\pi}_0(s) - 1 = -(\lambda + \lambda_0)\overline{\pi}_0(s) + \mu\overline{\pi}_1(s) + \mu_c\overline{\pi}_2(s) \\ s\overline{\pi}_1(s) = \lambda\overline{\pi}_0(s) - \mu\overline{\pi}_1(s) \\ s\overline{\pi}_2(s) = \lambda_0\overline{\pi}_0(s) - \mu_c\overline{\pi}_2(s) \end{cases}, \tag{3}$$

where $\overline{\pi}_i(s) = \int_0^\infty \pi_i(t)e^{-st}dt$.

The transformed vector $\overline{\boldsymbol{\pi}}(s)$ could be calculated as $\overline{\boldsymbol{\pi}}(s) = \boldsymbol{\pi}(0)(s\mathbf{I} - \mathbf{Q})^{-1}$ and the value of $\pi_i(t)$ is obtained by taking an inversion of the Laplace transform as

$$\pi_i(t) = \frac{1}{2\pi k}\int_{c-k\infty}^{c+k\infty} \overline{\pi}_i(s)e^{st}ds. \tag{4}$$

The transient reliability is defined as the probability that, at a specified operation time t, the software system is operating without a failure and the repair process. Without considering the recovery process, the model in Figure 2 could be reformulated into a Markov chain with an absorbing state:

$$R(t) = \pi_0(t) + \pi_2(t), \text{where } \overline{\pi}_0(s) = \frac{1}{s + \lambda + \lambda_0 - \dfrac{\lambda_0\mu_c}{s + \mu_c}}. \tag{5}$$

3.2 Dynamic reconfiguration algorithm

Based on the reliability analysis, all the variables in the reliability model are already known, exception for the software rejuvenation interval t_0 and the execution time t. Thus, this model is essentially the functions of t_0 and t: R=R(t, t_0). Given a certain time period t, during which no

software failure occurs, the value t_0 of could be calculated through optimizing the software reliability

$$\begin{aligned} &\text{Max } t_0 \\ &\text{s.t. } R(t,t_0) \geq R_0 \end{aligned}, \tag{6}$$

where R_0 denotes the reliability constraint.

That is to say, the value of optimal software rejuvenation interval t_0 varies according to the pre-chosen value of t. According to the research of Dohi et al.[14], as the operation time t elapses, the optimal software rejuvenation interval first increases and then decreases to a relatively stable value, shown by Figure 3.

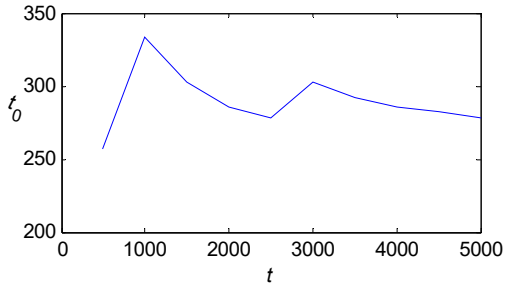

Figure 3. Dependence of the operation time t on the optimal rejuvenation policy.

However, during the long-time software execution process, the failure-free execution time t between two failures varies and it is difficult to identify it in advance. Using a large expected execution time or the steady state availability may cause a more frequent rejuvenation rate, which is unnecessary in the very beginning of the operation process. On the other hand, employing a small expected execution time may get an inappropriate rejuvenation rate, which could not guarantee the reliability after long-time execution.

Thus, an adaptive algorithm is used to select and reconfigure the software rejuvenation interval in the runtime. The core idea of this algorithm is to generate a time series of reconfiguration interval to recalculate the rejuvenation policy. Let Δt_i denotes the ith the reconfiguration interval and Δt_p is the time series of Δt_i: Δt_p=(Δt_1,Δt_2,Δt_3,Δt_4……). The time series is selected according to the following manners:

$$\begin{aligned} &\text{Max } \Delta t_{i+1} \\ &\text{s.t. } e^{-\lambda t_i} - e^{-\lambda t_{i+1}} \leq \varepsilon, t_i = \sum_{j=1}^{i} \Delta t_j \end{aligned}, \tag{7}$$

where $ER(t_i) = e^{-\lambda t_i}$ denotes the original software reliability without rejuvenation policies at time t_i and ε denotes the reliability degradation threshold. The original software reliability decreases obeying an exponential distribution, so when the software initially begins to execute, the trend of reliability change is relatively large. Then the software rejuvenation interval should be changed more often, which leads to a more frequent reconfiguration. After certain time of execution, the reliability decrease trend will be largely reduced and the rejuvenation interval will converge to a constant value. So, as the execution time increases, the reconfigure process should be slowed down, which leads to an increasing reconfiguration interval. Following (7), the value of reconfiguration interval increases gradually. The optimal value of Δt_i should be the largest value satisfying $e^{-\lambda t_i} - e^{-\lambda t_{i+1}} = \varepsilon$. Since the exponential function $ER(t_i) = e^{-\lambda t_i}$ decreases with the increase of t_i. A larger value of Δt_i would lead to a smaller $ER(t_i)$, which violates the reliability degradation constraint. The value of Δt_i is given by

$$\Delta t_i = -\frac{\ln(e^{-\lambda t_{i-1}} - \varepsilon)}{\lambda} - t_{i-1}, \Delta t_0 = 0, \tag{8}$$

where $1/\lambda$ stands for the mean time to failure (MTTF).

3.3 Adaptive reconfiguration mechanism

Based on the analysis of the dynamic reconfiguration interval and the optimal rejuvenation interval, the adaptive reconfiguration mechanism is built as follows.

Step I. Given an initial reconfiguration interval Δt_i(i=1), the corresponding t_0 is calculated through (6). Since it is too complex to solve the analytical model of $R(t)$ in a symbolic form, a numerical method is used to search the optimal value of t_0 using Algorithm 1. Then the periodic software rejuvenation policy is set using the value of t_0.

Algorithm1 Cal_opt_reju_inte(Δt_i, λ, μ, μ_c): Output t_0

t_0=Δt_i, $t_i = \sum_{j=1}^{i} \Delta t_j$

While t_0>0, solve (4) using numerical methods and calculate R according to (5) using the value of t_i:

If $R \geq R_0$

output t_0 and exit.

Else

decrease t_0 by a small predefined difference value t_0= t_0−δ and continue the loop.

Step II. During the specific execution time Δt_i, if no failures occur to the software, the new reconfiguration interval Δt_{i+1} is calculated at t_i through (8) and the procedure goes back to Step I with i=i+1, in which a new rejuvenation interval is optimized according to the new execution time t_{i+1}.

Step III. Otherwise, if failures occur during Δt_i, the software recovery operation is carried out. The whole software is treated as the initial state. Set i=0 and the procedure is directed back to Step I.

The algorithm of the whole procedure is show as follows.

Algorithm 2 Adp_reconfi(λ, μ, μ_c)

i=1

While true

If i ==1

Get the initial value of Δt_1.

Else

calculate t_{i-1}, then calculate Δt_i though (8) according to t_{i-1}.

Calculate optimal policy t_0 using the value of Δt_i by invoking Cal_opt_reju_inte(Δt_i, λ, μ, μ_c).

The optimal periodic rejuvenation policy is set using the interval value t_0.

For the period of Δt_i, monitor the software execution:

If software failures occur, recover the system, set i=1 and reset the loop from the initial state.

Else no failures occur until the time Δt_i elapses, set i=i+1 and continue the loop.

4 Experimental studies

4.1 Optimal rejuvenation interval based on the transient reliability

The parameters of the software is set as λ=0.0001(h^{-1}), μ=1/2(h^{-1}), μ_c=1/6(h^{-1}), the reliability degradation threshold is set as ε=0.001 and the reconfiguration interval decrement value is δ=0.1(h). The transient software reliability change with or without certain rejuvenation polices is shown in Figure 4.

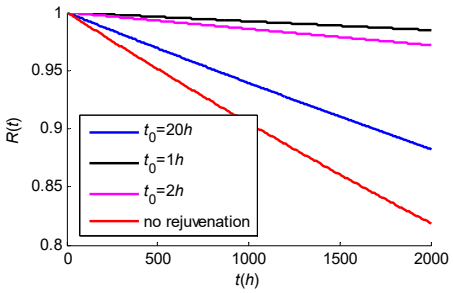

Figure 4. The transient software reliability change with t.

It is observed that as the time t increases, the software reliability drops continuously, even with a rejuvenation policy. The smaller rejuvenation interval, or more frequent software rejuvenation, will prevent the software reliability from dropping rapidly. The similar conclusion could be drawn from Figure 5. For a certain amount of execution time, the reliability decreases with the increase of rejuvenation interval. Moreover, to achieve the same reliability constraint, e.g. 0.8, a smaller interval or a more frequent rate should be set for a longer execution time.

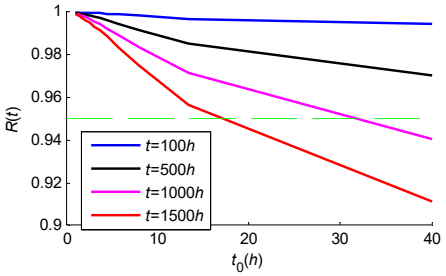

Figure 5. The transient software reliability change with t_0.

Following the Algorithm 1, the optimal rejuvenation policy which satisfies the reliability constraint is calculated for different execution time period t with initial $\Delta t_1 = \infty$. The results are shown in Figure 6 using different reliability constraints.

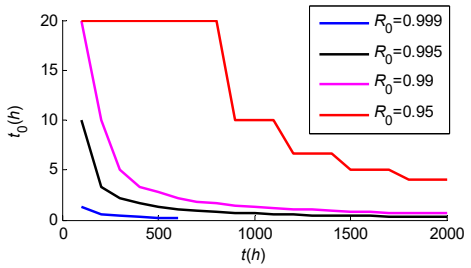

Figure 6. The optimal software rejuvenation interval t_0 varying with execution time and reliability constraints.

Figure 6 shows that as the execution time increases, the optimal rejuvenation interval should be decreased to maintain the same reliability constraints. Similarly, higher reliability constraints calls for more frequent software rejuvenation.

4.2 The adaptive rejuvenation interval reconfiguration

According to the experiments in the last section, it is unnecessary to set a high rejuvenation rate in the beginning of the software execution process. During the execution, the rejuvenation interval should be adjusted to maintain the reliability constraints. So, a reconfiguration interval time series is first calculated using different software failure rate λ and the results is show in Figure 7.

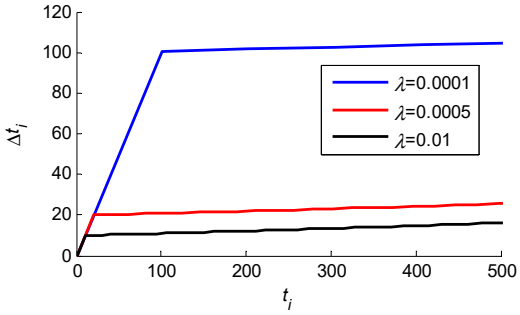

Figure 7. The time series for the reconfiguration interval and the expected execution time under different failure rate.

Figure 7 shows that the reconfiguration interval increases with the execution time, which indicates that a more frequent reconfiguration process should be carried out when there are more dramatic changes in the software reliability. After long-time execution, the reconfiguration process should be slowed down since the change in the reliability decreases. So the reconfiguration interval changes fast at first and then the increasing trend declines. According to these features, an adaptive reconfiguration mechanism is used to select the software rejuvenation interval dynamically and the results are shown in Figure 8. The results in Figure 8 show that after a certain amount of time, a new rejuvenation interval is calculated. The new rejuvenation policy aims to maintain the software reliability until the next reconfiguration interval.

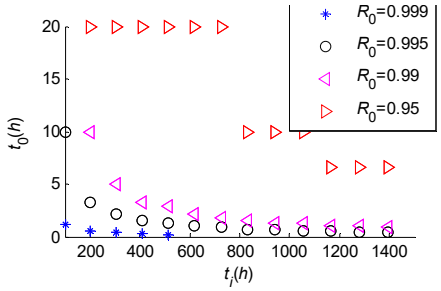

Figure 8. The adaptive rejuvenation policy changing dynamically with execution time *t*

5 Conclusions and future work

Software rejuvenation, as a preventive and proactive maintenance policy, has been considered as the main solution to avoid severe or aging-related failures. As it introduces extra overhead, some researches have paid attention to the problem of choosing optimal software rejuvenation intervals based on reliability analysis. However, these studies generally borrow the steady-state analysis methods to calculate a constant rejuvenation policy. They ignore the potential change of the policy along with the execution time. So this paper aims to propose an adaptive mechanism to reconfigure the periodic software rejuvenation policy dynamically. The transient reliability analysis is used to choose an optimal interval value to maintain the software reliability for a certain amount of time. A time series is generated for the reconfiguration process and the optimal rejuvenation policy is re-calculated according to the reconfiguration intervals dynamically. Experimental studies results showed that the software reliability drops continuously, even with a rejuvenation policy. As the execution time increases, the optimal rejuvenation interval should be decreased to maintain the same reliability constraints and our mechanism could reset the interval values according to a reconfiguration interval time series. In the next step of work, the analytical methods and the numerical methods should be combined to improve the calculation accuracy.

Acknowledgement

This work is supported by the Key Technology Innovation Projects for Key Industries in Chongqing (No.cstc2015zdcy-ztzx40008)

References

1. H. Wang, L. Wang, Q. Yu, Z. Zheng, A. Bouguettaya, M.R. Lyu, Online reliability prediction via motifs-based dynamic Bayesian networks for service-oriented systems, IEEE Trans Softw Eng, **43**, 556-579 (2017)
2. M. Grottke, K.S. Trivedi, Fighting bugs: Remove, retry, replicate, and rejuvenate, Computer, **40**, 107–109 (2007)
3. V.P. Koutras, A.N. Platis, VoIP availability and service reliability through software rejuvenation policies, *Proc. of 2nd Int Conf Dependability of Computer Systems*, 262–269 (2007)
4. J. Alonso, R. Matias, E. Vicente, A. Maria, K.S. Trivedi, A comparative experimental study of software rejuvenation overhead, Perform Eval, **70**, 231–250 (2013)
5. J. Zheng, H. Okamura, L. Li, T. Dohi, A comprehensive evaluation of software rejuvenation policies for transaction systems with Markovian arrivals, IEEE Trans Rel, **66**, 1157–1177 (2017)
6. H. Okamura, T. Dohi, Dynamic software rejuvenation policies in a transaction-based system under Markovian arrival processes, Perform Eval, **70**, 197–211 (2013)
7. H. Okamura, J. Zheng, T. Dohi, A statistical framework on software aging modelling with continuous-time hidden Markov model, *Proc. of 36th Int Symp Rel Dist Systems*, 114–123 (2017)
8. S. Garg, S. Pfening, A. Puliafito, M. Telek, K.S. Trivedi, Analysis of preventive maintenance in transactions based software systems, IEEE Trans Computers, **47**, 96–107 (1998)
9. A. Bobbio, M. Sereno, C. Anglano, Fine grained software degradation models for optimal rejuvenation policies, Perform Eval, **46**, 45–62 (2001)
10. H. Okamura, H. Fujio, T. Dohi, Fine-grained shock models to rejuvenate software systems, IEICE Trans Info Syst, **E86-D**, 2165–2171 (2003)
11. H. Okamura, S. Miyahara, T. Dohi, S. Osaki, Performance evaluation of workload-based software rejuvenation scheme, IEICE Trans Info Syst, **E84-D**, 1368–1375 (2001)
12. W. Xie, Y. Hong, K.S. Trivedi, Analysis of a two-level software rejuvenation policy, Rel Eng Syst Saf, **87**, 13–22 (2005)
13. G. Ning, J. Zhao, Y. Lou, J. Alonso, R. Matias, K.S. Trivedi, B.B. Yin, K.Y. Cai, Optimization of two-granularity software rejuvenation policy based on the Markov regenerative process, IEEE Trans Rel, **65**, 1630–1646 (2016)
14. T. Dohi, J. Zheng, H. Okamura, K.S. Trivedi, Optimal periodic software rejuvenation policies based on interval reliability criteria, Reliab Eng Syst Saf (2018, in publication process)
15. H. Suzuki, T. Dohi, K. Goseva-Popstojanova, K.S. Trivedi, Analysis of multistep failure models with periodic software rejuvenation, *Advances in Stochastic Modelling*, 85–108 (Notable Publications Inc., 2002)