# Lifetime Extension of Software Execution subject to Aging

Fumio Machida, *Member, IEEE*, Jianwen Xiang, *Member, IEEE*,
Kumiko Tadano, *Member, IEEE*, and Yoshiharu Maeno, *Member, IEEE*

***Abstract*—Software aging is a phenomenon of progressive degradation of software execution environment caused by software faults. In this paper, we propose software life-extension as an operational countermeasure against software aging and present the mathematical foundations of software life-extension by means of stochastic modeling. A semi-Markov process is used to capture the behavior of a system with software life-extension and to analyze the system's availability and completion times of jobs running on it. The semi-Markov process can correctly model the time-based life-extension and allows us to derive the optimal trigger for starting life-extension in terms of system availability and mean job completion time. We also present an effective combination of software life-extension and software rejuvenation that can maximize the system availability compared with a system using either rejuvenation or software life-extension.**

***Index Terms*— Availability, job completion time, software aging, software life-extension, software rejuvenation**

## ACRONYMS

| | |
|---|---|
| SMP | semi-Markov process |
| VM | virtual machine |
| CTMC | continuous time Markov chain |
| IFR | increasing failure rate |
| PRT | preemptive-repeat discipline |
| PRS | preemptive-resume discipline |
| LST | Laplace-Stieltjes transform |

## NOTATION

| | |
|---|---|
| $F_f(t), F_e(t)$ | failure time distribution in UP state and life-extended state, respectively |
| $G_{rc}(t), G_{rj}(t)$ | recovery time distribution and rejuvenation time distribution |
| $\mathrm{u}(t)$ | unit step function |
| $\tau_R, \tau_L$ | rejuvenation and life-extension trigger interval, respectively |
| $A_R, A_L, A_{LR}$ | steady-state system availability by rejuvenation, life-extension and hybrid approach respectively |
| $\pi_i$ | steady-state probability in state *i* |
| $h_i$ | mean sojourn time in state *i* |
| $A_R^*, A_L^*, A_{LR}^*$ | maximum steady-state availabilities had by taking rejuvenation, life-extension, and hybrid approaches respectively |
| $\tau_R^*, \tau_L^*$ | optimum rejuvenation and life-extension trigger interval, respectively |
| $r_f(t), r_e(t)$ | failure rate functions with respect to $F_f(t)$ and $F_e(t)$ |
| $\tau_L$ | life-extension trigger interval |
| $U_{LR}, D_{LR}$ | fractions of up-time and downtime in $A_{LR}$ |
| $\mathrm{T}(x)$ | random variable for job completion time for $x$ units of work requirement |
| $\Phi_T^{\sim}(s,x)$, $\Phi_{TL}^{\sim}(s,x)$ | LSTs of completion times of jobs running on aging system and system using life-extension, respectively |
| $G_{rc}^{\sim}(s)$ | LST of recovery time |
| $r$ | decrease in job processing rate |

- *F. Machida is with the NEC Laboratory for Analysis of System Dependability (LASD), Kawasaki, Japan. E-mail: f-machida@ ab.jp.nec.com.*
- *J. Xiang is with the Wuhan University of Technology, Wuhan, Hubei, 430070, China. E-mail: jwxiang@whut.edu.cn.*
- *K. Tadano is with the NEC Laboratory for Analysis of System Dependability (LASD), Kawasaki, Japan. E-mail: k-tadano@ bq.jp.nec.com.*
- *Y. Maeno is with the NEC Laboratory for Analysis of System Dependability (LASD), Kawasaki, Japan. E-mail: y-maeno@ aj.jp.nec.com.*

## I. INTRODUCTION

SOFTWARE faults or bugs are major sources of unreliability affecting software-based IT systems. As software is used extensively in mission-critical infrastructures such as telecommunication networks, industrial plants, banking and transportation systems, the impact of software bugs is more significant. Even if modern testing and debugging techniques are employed, complete removal of latent software bugs is extremely difficult and/or prohibitively expensive. Hence, software systems tend to suffer from residual bugs that lead to unexpected system failures. To mitigate the adverse effects of residual bugs on IT system reliability, maintenance operations such as software updates and data backups are essential in practice.

Aging-related bugs are a category of bug by which the software aging problem is induced. Software aging is the phenomenon of progressive degradation of the software execution environment and it increases the failure rate of software [1]. A typical example of software aging is a progressive increase in memory consumption that causes memory leaks. Detection of aging related-bugs before execution is usually difficult because software aging only manifests itself in specific execution environments.

Software rejuvenation is a well-known operational measure for mitigating the aging problem [2]. It is a preventive maintenance technique that cleans up the internal states of the software execution environment by restarting or resetting before the software faces serious performance degradation and/or failure. Although software rejuvenation can prevent system failures due to software aging, it involves downtime overhead due to restarts. This downtime overhead may not be acceptable for certain applications. For example, software rejuvenation is unsuitable for applications that process long jobs over the course of days or even months, since it erases all the intermediate results by resetting the execution environment. It is vital for such applications to keep running as long as possible during their mission period.

In our previous study [3], we presented an alternative countermeasure to software aging; called *software life-extension*, it is a preventive maintenance technique that prolongs the lifetime of software execution as long as possible in the face of software aging. If failures caused by software aging can be postponed for a while, users or applications may effectively use the extended residual lifetime. As an example, if the content of an application with a memory cache is preserved during the extended lifetime, it might be possible to finish the user sessions within its lifetime or save the cached content in persistent storage before encountering a failure. This approach is particularly useful for long-running mission-critical applications which require continuous up-time.

In this paper, we further investigate the effectiveness of software life-extension against software aging. In contrast to the simple Continuous Time Markov Chain model presented in [3], we reformulate the system's behavior with a semi-Markov process (SMP). The SMP allows us to use a general distribution for the failure time distribution and to correctly represent the behavior of a time-based software life-extension whose interval is deterministic. By analyzing the SMP, we find the optimum life-extension interval that can maximize the system availability. On the basis of the theoretical foundation of SMP, we can theoretically clarify the conditions under which the unique optimum life-extension interval exists that have never been addressed in the previous literature. Moreover, we propose an effective hybrid approach in which software life-extension is followed by rejuvenation. The extended SMP that captures the behavior of the system with the hybrid approach is then used to find the optimum combination of intervals for life-extension and rejuvenation that maximizes the system availability. We also show the impact of the software life-extension on the completion time distribution of jobs running on the software system. Through a numerical study, we show that the hybrid approach is a better option than relying on solely either rejuvenation or life-extension in terms of both system availability and job completion time.

The rest of the paper is organized as follows. Section II introduces the general concept of software life-extension and reviews our feasibility study on it. Section III presents the SMP for software systems using rejuvenation, life-extension, and their combination. By conducting a steady-state analysis of SMP, we find the optimum life-extension interval that maximizes system availability. In Section IV, we analyze the job completion time distributions based on the SMPs in which the preemption type in each state is considered. In Section V, we describe a numerical study showing that the hybrid approach is the best in terms of system availability. The optimum life-extension interval in terms of mean job completion time is also presented. Section VI reviews related work and Section VII provides discussion. Finally, we conclude the paper in Section VIII.

## II. Software Life-extension

We consider software systems that may suffer from software aging. As introduced in the previous section, software rejuvenation is known to be an effective countermeasure to software aging and to improve the system availability. Software rejuvenation has been introduced for the sake of high-availability [2][8][11][13], system performance [18][21][32] or job completion time performance [16]. Our new approach presented in this section is a potential alternative for software rejuvenation and it also aims to improve the availability and the performance of systems suffering from software aging problem. We start from the introduction to the concept of software life-extension.

### A. Concept

The term *software life-extension* comes from a natural extension of the metaphor of software aging. Software aging represents the transient state of the software execution environment, where available resources gradually decrease due to aging-related software faults. The lifetime of software execution reaches its limit when the system depletes its resources as a result of the accumulated aging effects. Software life-extension aims at postponing such failures by impeding the progress of software aging. It is a temporal mitigation to extend the lifetime of the execution environment, but it does not provide a radical solution to the fault causing the software aging. To extend the lifetime of software execution, supplemental resources could be assigned to the execution environment if the application can make use of them. Alternatively, software aging can be impeded by decreasing the workload, provided that the rate of aging depends on the workload. These approaches do not require any changes to the source code and are often easily applicable by means of common maintenance operations, commands, or scripts. Therefore, software life-extension is a non-intrusive countermeasure to software aging.

### B. Means

There are at least two conceptual ways to implement software life-extension: dynamic resource allocation and workload control. The first approach extends the lifetime of aged software through dynamic resource allocation in which the amount of resources is increased dynamically during execution. Recent advances in virtualization technologies make such resource allocations at runtime possible. For example, Xen hypervisor[1] provides a functionality to virtualize hardware resources and allocate them to a Virtual Machine (VM). In this approach, we need standby resources which can be allocated dynamically and may be shared with other software execution

[1] Xen, http://www.xen.org/

environments. The use of standby resources may incur costs, such as higher resource usage costs imposed by the cloud and/or hosting service, and unavailability of other services sharing the standby resource.

The second approach controls the workload so as to decrease the load on the aged software. This approach is limited to applications that work with a workload manager or have a load balancer in the front-end. The workload is reduced by assigning jobs to other instances or dropping job requests at the workload manager or load balancer. Software aging is often associated with the workload of the software [5][6]; therefore, aging can be impeded by reducing the workload. Although this helps to extend the lifetime of the software, resource exhaustion is inevitable as long as the software continues executing. This approach can be considered to be like designing a system that can survive even in the case of a component failure. Unlike typical degradable systems, software life-extension using workload control does not guarantee that the software will continue to execute. Even after a life-extension, the software may eventually encounter a failure due to resource exhaustion because life-extension itself does not remove the root-cause. Similar to the first approach, workload control may also incur additional problems, such as workload reallocations that overload other instances and the workload manager rejecting requests.

Consequently, although both of these approaches are feasible in a real system they require specific system configurations, preparations and resources. The appropriate means should be decided considering the application type and system environment. In section II.D, we show the feasibility of software life-extension by taking the dynamic resource allocation approach.

## C. *Advantages and drawbacks*

Regardless of the above-mentioned means, the primary advantage of software life-extension over software rejuvenation is continuous execution even as the software ages. Although software rejuvenation clears the aging states in a relatively short amount of downtime, it interrupts the software execution and loses potentially valuable data in memory. In contrast, software life-extension can maintain availability without any interruptions as long as possible. When an application requests a job requiring a long execution time and the question is whether or not the job will complete, life-extension is preferable to rejuvenation. Software life-extension is also suitable for applications with predetermined mission times. We can use it to meet the mission time requirement when the software is likely to finish execution before the mission time is up.

Another benefit of software life-extension is its capability of preserving memory content, as mentioned in the Introduction. The persistence of data accumulated in memory is essential to some forms of software. Software rejuvenation completely erases such data, and thus, it may cause a serious degradation in service quality. A typical example of such important memory content is paging data in an operating system. The deletion of paging data during a reboot causes a performance degradation, as reported in [7]. In contrast, software life-extension attempts to preserve memory content as long as possible. While the content of memory is eventually lost at the end of the system's life, the user may wisely use the residual lifetime to make a backup or take a snapshot and save it in persistent storage.

As discussed earlier, software life-extension incurs additional resource usage costs, performance degradations, and degradations to the availability of other services. These are potential drawbacks if they become unacceptably large or unpredictable. The trade-off is the additional lifetime in exchange for these costs. Although rejuvenation imposes an additional downtime cost, it does not require a specific system configuration (e.g., a load balancer) or any standby resources.

Unlike the hot-fix approach that corrects the source code by removing the source of software aging [4], software life-extension does not remove the source. Hence, relying on software life-extension for a long time may hinder the chances of finding and removing the root cause of the aging, which is something that system administrators should be aware of when they consider using life-extension.

## D. *Feasibility study*

In our previous study [3], we implemented software life-extension by using the dynamic memory allocation method provided by Xen hypervisor. This subsection briefly reviews this experimental study of VM-based software life-extension. In the experiment we used memcached[2], which is an in-memory key-value store for caching objects usually used as a cache server for database systems. It simply implements a hash table whose content is read or inserted using the corresponding keys. Memcached has a configuration parameter that specifies the maximum size of memory for cache data. This memory does not include the memcached footprint, and it can be set to a value exceeding the available memory in the system. Thus, setting inadequate limits will cause a memory leak after a gradual increase in memory consumption (i.e., software aging). If such a limit is embedded in software relying on memcached, the problem cannot be easily located and removed.

In order to mitigate the software aging of memory consumption, we used the dynamic memory allocation method provided by Xen hypervisor. When a suspicious aging trend in memory usage was detected, we allocated additional memory to the VM executing the application in order to postpone the potential memory leak. The feasibility of this approach was studied with the experimental test bed shown in Figure 1.

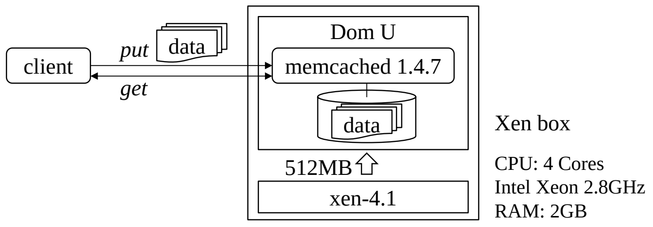

Figure 1. Test bed system for software life-extension

In this test bed, a VM is created on Xen box, and 512MB of memory is initially allocated to the VM. The VM has another

[2] Memcached, http://memcached.org/

512MB of swap space. A single instance of memcached is deployed on the VM, and it starts with a maximum limit of 900MB. Although the limit is within the sum of available memory and swap space (512+512=1024MB), it does count the memcached footprint. A client program generates requests to the memcached for the load test that repeatedly inserts 1MB of unique data and reads it in a subsequent access. During the load generation, we do not insert any delays between consecutive requests. According to the number of insert requests, the memory consumption of memcached gradually increases because the data is cached in memory. When the number of insert events exceeds 500, memory swapping starts. If no counteractions are performed, the VM eventually crashes when it runs out of memory. Figure 2 shows the changes in free memory and swap usage in the VM during this experiment. The VM hanged up when it used up all of the free memory and available swap space.

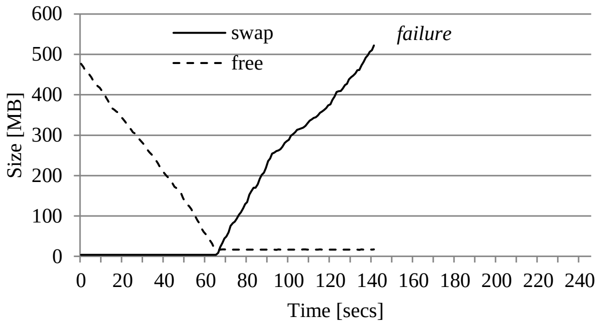

Figure 2. Decrease in free memory and increase in swap spaces

We can postpone a failure by allocating additional memory to the VM, namely by using software life-extension. When the swap usage exceeds 400MB, we allocate 88MB of additional memory to the VM through Xen's command line interface. The total memory is increased to 600MB and the lifetime of the VM is extended. Figure 3 shows the changes in the amount of free memory and swap usage as a result of software life-extension. The steep increase in swap usage pauses when the software life-extension is performed, but it begins again once the free memory is used up. The swap usage reaches 500MB around 135 seconds. The VM does not immediately run out of memory, but the swap usage fluctuates around 500MB with a subtle upward trend. The execution state after software life-extension is not stable as in the beginning of the process, but the application can benefit from the prolonged lifetime.

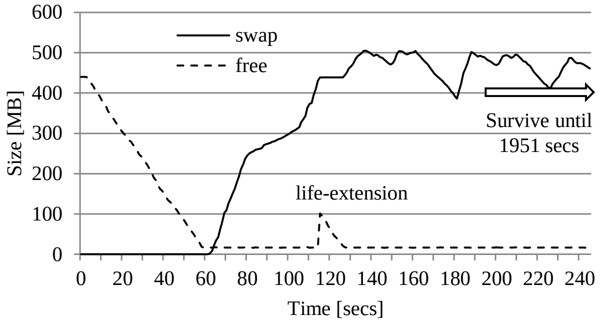

Figure 3. Changes in free memory and swap usage by software life-extension

The effect is partly caused by the memory management mechanism in the operating system which attempts to survive with limited resources. The experimental results are discussed in detail in [3].

## III. System model

In order to study the advantages and drawbacks of software life-extension in relation to software rejuvenation, we presented continuous time Markov chain (CTMC) models in the previous paper [3]. The basic assumption of CTMC is that all the state transitions times are exponentially distributed. In this paper, we relax this assumption and devise a more general model using a semi-Markov process (SMP). The state transitions in SMP can follow any type of distribution. This property allows us to represent deterministic trigger for starting preventive operations (software rejuvenation and software life-extension).

In the following subsections, we review the general SMP model for time-based software rejuvenation and the way to get the optimal software rejuvenation interval in terms of system availability. Next, we propose a SMP model for time-based software life-extension in which software life-extension is applied in a deterministic time interval after the latest restart. Interestingly, under specific conditions, software life-extension also has an optimal trigger interval in terms of system availability. We theoretically clarify this point in Section 0.B. Finally, Section 0.C describes an SMP model for a hybrid approach in which software life-extension is followed by software rejuvenation.

### A. *Time-based rejuvenation model*

In 1995, Garg et al [8] were the first to model time-based software rejuvenation with Markov Regenerative Stochastic Petri Net (MRSPN). Later Suzuki et al. [9] and Chen et al. [10] introduced a three state SMP model that is equivalent to the original MRSPN model.

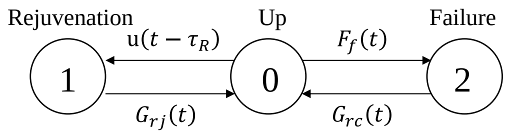

Figure 4. SMP representing the system behavior with rejuvenation

Figure 4 shows the general three-state SMP model for time-based software rejuvenation. In the previous decade, many researchers used this model to analyze the optimum software rejuvenation trigger interval for maximizing system availability or minimizing the downtime cost (e.g., [10][12]). In Figure 4, state 0 is the up state (the software is running). From state 0, the system enters either state 1, i.e., the rejuvenation state, or state 2, i.e., the failure state. The failure time distribution is represented by $F_f(t)$, while the deterministic transition from state 0 to state 1 can be represented by a unit step function $u(t-\tau_R)$, where $\tau_R$ is the rejuvenation trigger interval. The recovery time distributions from state 1 and state 2 are represented by $G_{rc}(t)$ and $G_{rj}(t)$ , respectively. The steady-state availability of the system $A_R$ is computed as the steady-state probability of state 0 ($\pi_0$) [10]:

$$A_R = \pi_0 = \frac{h_0}{h_0 + h_1\left(1 - F_f(\tau_R)\right) + h_2 F_f(\tau_R)} \tag{1}$$

where $h_0$, $h_1$ and $h_2$ are the mean sojourn times for the corresponding states:

$$h_0 = \int_0^{\tau_R}\left(1 - F_f(t)\right)dt\,, h_1 = \int_0^{\infty}\left(1 - G_{rj}(t)\right)dt\,, \quad h_2 = \int_0^{\infty}\left(1 - G_{rc}(t)\right)dt \tag{2}$$

Dohi et al. [11] showed that expression (1) is strictly convex with respect to $\tau_R$ if $F_f(t)$ has the property of increasing failure rate (IFR). Since it is natural to assume that the IFR is caused by software aging, the above condition is likely to hold. If we assume $\frac{dA(0)}{d\tau_R} > 0$ and $\frac{dA(\infty)}{d\tau_R} < 0$, the steady-state availability is maximized at $\tau_R^*$, which satisfies the following equation.

$$\left(1 - F_f(\tau_R^*)\right)\left[h_1\left(1 - F_f(\tau_R^*)\right) + h_2 F_f(\tau_R^*)\right] - \frac{dF_f(\tau_R^*)}{d\tau_R}h_0(h_2 - h_1) = 0 \tag{3}$$

The maximum steady-state availability is

$$A_R^* = \frac{1 - F_f(\tau_R^*)}{1 - F_f(\tau_R^*) + \frac{dF_f(\tau_R^*)}{d\tau_R}(h_2 - h_1)} \tag{4}$$

The optimum rejuvenation interval $\tau_R^*$ is not represented symbolically as it is determined in non-linear equation (3). However, it can be obtained by taking a numerical approach as in [12].

### B. *Time-based life-extension model*

We construct an SMP model for software life-extension in an analogous way to software rejuvenation. The system is assumed to be failed with failure distribution $F_f(t)$, which is IFR due to software aging. If we apply software life-extension before a system failure, the system enters a new state whose failure rate must be smaller than the original state. To represent this state transition, we add a new life-extended state to the SMP. The proposed general SMP model is shown in Figure 5.

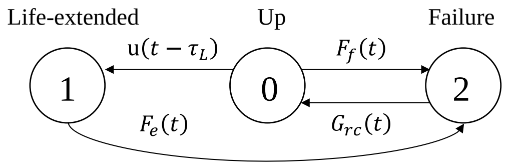

Figure 5. SMP representing the behavior of a system with life-extension

Similar to the rejuvenation model, state 0 and state 2 represent the up and failed states, respectively. State 1 is a life-extended one that has an incoming transition from state 0 and an outgoing transition to state 2. Unlike what happens with software rejuvenation, the system eventually fails regardless of whether life-extension is applied or not. In other words, software execution ends only in state 2. However, we can extend the lifetime of the software execution at an appropriate time. The failure time distribution changes from state 0 to state 1. We denote the failure time distribution in state 1 as $F_e(t)$. If the software life-extension is applied in a deterministic time interval $\tau_L$, the distribution from state 0 to state 1 can be represented as a unit step function u$(t - \tau_L)$. We use the same distribution $G_{rc}(t)$ for the recovery transition from state 2. Our previous study [3] assumed that the failure time distribution is hypo-exponential, but the presented SMP model allows $F_f(t)$ to be a general distribution. The SMP model is a generalization of our previous model for software life-extension. Note that the Markov property in state transition from state 0 to state 1 may impact on the applicability of the model if the failure time distribution $F_e(t)$ depends on the time spent in the state 0. We will discuss such a special case in Section VII.

Using the two-stage method for SMP [14], the steady-state availability of the system can be computed from the sum of steady-state probabilities for state 0 and state 1:

$$A_L = \pi_0 + \pi_1 = \frac{h_0 + h_1\left(1 - F_f(\tau_L)\right)}{h_0 + h_1\left(1 - F_f(\tau_L)\right) + h_2} \tag{5}$$

where

$$h_0 = \int_0^{\tau_L}\left(1 - F_f(t)\right)dt\,, h_1 = \int_0^{\infty}\left(1 - F_e(t)\right)dt\,, \quad h_2 = \int_0^{\infty}\left(1 - G_{rc}(t)\right)dt \tag{6}$$

Steady-state availability $A_L$ can be considered as a function of $\tau_L$. Define the failure rate function $r_f(t)$

$$r_f(t) = \frac{1}{1 - F_f(t)}\frac{dF_f(t)}{dt} \tag{7}$$

Similar to the analysis of optimum rejuvenation interval reviewed in the previous section, it is interesting to clarify the condition where the value of $A_L$ is maximized. Since the life-extension changes the state to prolong the time to failure, the effectiveness of life-extension relies on the relation between the failure time distribution $F_f(t)$ and $F_e(t)$. The following theorem indicates that the optimal trigger of life-extension determines by the relation of the failure rate function $r_f(t)$ and the mean sojourn time $h_1$, which are characterized by $F_f(t)$ and $F_e(t)$, respectively.

**Theorem 1.** *When the failure time distribution $F_f(t)$ is IFR and the mean sojourn time in the life-extended state $h_1$ satisfies the inequality, $r_f(0) < 1/h_1 < r_f(\infty)$, there is a unique value $\tau_L^*$ that maximizes the value of $A_L$.*

**Proof.** In the following proof, we show that $A_L(\tau_L)$ is concave in the range of $\tau_L > 0$ under the given condition. First taking the derivative of $A_L(\tau_L)$ in terms of $\tau_L$, we get

$$\frac{dA_L(\tau_L)}{d\tau_L} = \frac{h_2\left(1 - F_f(\tau_L) - h_1\frac{dF_f(\tau_L)}{d\tau_L}\right)}{\left(h_0 + h_1\left(1 - F_f(\tau_L)\right) + h_2\right)^2} \tag{8}$$

The sign of the derivative depends on the numerator and especially on the following term,

$$1 - \frac{1}{1 - F_f(\tau_L)}h_1\frac{dF_f(\tau_L)}{d\tau_L} = 1 - h_1 r_f(\tau_L) \tag{9}$$

The failure rate function $r_f(t)$ is a strictly monotonically increasing function, as the corresponding distribution $F_f(t)$ is IFR. The sign of (9) changes from negative to positive at a certain value in the range of $\tau_L > 0$ under the given condition $r_f(0) < 1/h_1 < r_f(\infty)$. As a result $A_L(\tau_L)$ is a concave function in $\tau_L > 0$ and the value is maximized at $\tau_L^*$ that satisfies $h_1 = 1/r_f(\tau_L)$. □

The optimum interval $\tau_L^*$ is not represented symbolically, but the value can be numerically obtained in a similar way as the optimum software rejuvenation interval.

Intuitively, the failure rate in state 0 increases over time, and whenever it reaches the mean failure rate in state 1 ($1/h_1$), it is the best timing at which to move to state 1. Instead of a decreased failure rate in state 1, there may be a performance penalty after life-extension; this is addressed in the job completion time analysis presented in Section IV.

## C. *Hybrid approach model*

Software rejuvenation and software life-extension are not exclusive. Rather, they can be combined together in an epoch of the execution lifecycle. We devised such a hybrid approach in which software life-extension is followed by software rejuvenation. The corresponding SMP is drawn in Figure 6.

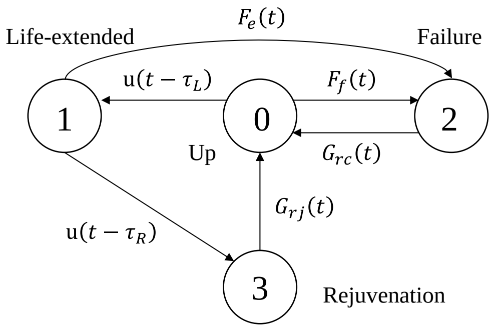

Figure 6. SMP representing the system behavior with both rejuvenation and life-extension

The model has both a life-extended state (state 1) and a rejuvenation state (state 3). Software life-extension is applied at time $\tau_L$ after the system starts in state 0, while software rejuvenation is applied at time $\tau_R$ after the system enters in state 1. The recovery time distributions from state 2 and state 3 are represented by $G_{rc}(t)$ and $G_{rj}(t)$, respectively, and the failure time distribution in state 1 is denoted as $F_e(t)$. In this system, software rejuvenation can be used after a software life-extension so as to minimize the potential downtime. Note that the SMP model asymptotically becomes the time-based life-extension model by taking $\tau_R$ to infinity, and it becomes the time-based rejuvenation model by taking $\tau_R$ to be 0. In the following discussion, we exclude these extreme cases and assume that $0 < \tau_R < \infty$.

The steady-state availability of the system is the sum of the steady-state probabilities of state 0 and state 1, as computed by

$$A_{LR} = \pi_0 + \pi_1 = \frac{U_{LR}}{U_{LR} + D_{LR}} \tag{10}$$

where

$$\begin{aligned} U_{LR} &= h_0 + h_1\left(1 - F_f(\tau_L)\right) \\ D_{LR} &= h_2\left[F_f(\tau_L) + \left(1 - F_f(\tau_L)\right)F_e(\tau_R)\right] \\ &\quad + h_3\left(1 - F_f(\tau_L)\right)\left(1 - F_e(\tau_R)\right) \end{aligned}$$

and

$$\begin{aligned} h_0 &= \int_0^{\tau_L}\left(1 - F_f(t)\right)dt,\ h_1 = \int_0^{\tau_R}\left(1 - F_e(t)\right)dt, \\ h_2 &= \int_0^{\infty}\left(1 - G_{rc}(t)\right)dt,\ h_3 = \int_0^{\infty}\left(1 - G_{rj}(t)\right)dt \end{aligned} \tag{11}$$

We can control both the life-extension interval $\tau_L$ and software rejuvenation interval $\tau_R$, and thus, the steady-state availability $A_{RL}$ can be considered to be a bivariate function of $\tau_L$ and $\tau_R$. Let us define the hazard rate functions $r_f(t)$ and $r_e(t)$ as

$$r_f(t) = \frac{1}{1 - F_f(t)}\frac{dF_f(t)}{dt}, \quad r_e(t) = \frac{1}{1 - F_e(t)}\frac{dF_e(t)}{dt} \tag{12}$$

The following theorem is derived for analyzing the optimum combination of life-extension interval and rejuvenation interval that maximizes $A_{LR}$.

**Theorem 2.** *When both the failure time distribution $F_f(t)$ and the failure time distribution in life-extended state $F_e(t)$ are IFR, and the nonlinear function $Z = h_1h_2 - \left(1 - F_e(\tau_R)\right)(h_2 - h_3)h_0$ is always smaller than 0, and $h_2 > h_3$, the optimum combination of the life-extension interval $\tau_L$ and rejuvenation interval $\tau_R$ that maximizes $A_{RL}$ is given by the solution of the following system of equations.*

$$\begin{cases} D_{LR} - r_e(\tau_R)(h_2 - h_3)U_{LR} = 0 \\ D_{LR} + r_f(\tau_L)Z = 0 \end{cases} \tag{13}$$

**Proof.** First, we prove that $A_{RL}$ is a strictly concave function in terms of $\tau_R$. Then we show the function is also strictly concave on $\tau_L$ under the given condition. The derivative of $A_{RL}$ with respect to $\tau_R$ is

$$\frac{\partial A_{LR}}{\partial \tau_R} = \frac{1}{(U_{LR} + D_{LR})^2}\left[\frac{\partial U_{LR}}{\partial \tau_R}D_{LR} - \frac{\partial D_{LR}}{\partial \tau_R}U_{LR}\right] \tag{14}$$

The derivatives of $U_{RL}$ and $D_{RL}$ with respect to $\tau_R$ are

$$\begin{aligned} \frac{\partial U_{LR}}{\partial \tau_R} &= \left(1 - F_f(\tau_L)\right)\left(1 - F_e(\tau_R)\right) \\ \frac{\partial D_{LR}}{\partial \tau_R} &= \frac{\partial F_e(\tau_R)}{\partial \tau_R}(h_2 - h_3)\left(1 - F_f(\tau_L)\right) \end{aligned} \tag{15}$$

Applying (15) to (14), the sign of (14) is determined by the following term in the numerator.

$$W = D_{LR} - r_e(\tau_R)(h_2 - h_3)U_{LR} \tag{16}$$

Taking the derivative of $W$ with respect to $\tau_R$ and using (15), we get

$$\begin{aligned} \frac{\partial W}{\partial \tau_R} &= \frac{\partial D_{LR}}{\partial \tau_R} - (h_2 - h_3)\left[\frac{\partial r_e(\tau_R)}{\partial \tau_R}U_{LR} + r_e(\tau_R)\frac{\partial U_{LR}}{\partial \tau_R}\right] \\ &= -\frac{\partial r_e(\tau_R)}{\partial \tau_R}(h_2 - h_3)\left(h_0 + \left(1 - F_f(\tau_L)\right)h_1\right) < 0 \end{aligned} \tag{17}$$

Since $W$ is strictly decreasing, $A_{RL}$ is strictly concave in $\tau_R$. The optimum rejuvenation interval is given by $\tau_R^*$, which satisfies $W$=0. The maximum availability is written as

$$A_{LR}^{*} = \frac{1}{1 + r_e(\tau_R^{*})(h_2 - h_3)} \tag{18}$$

With the optimal rejuvenation interval $\tau_R^{*}$, we consider $A_{LR}$ as a function of $\tau_L$ and take the derivative with respect to $\tau_L$:

$$\frac{\partial A_{LR}}{\partial \tau_L} = \frac{1}{(U_{LR} + D_{LR})^2}\left[\frac{\partial U_{LR}}{\partial \tau_L} D_{LR} - \frac{\partial D_{LR}}{\partial \tau_L} U_{LR}\right] \tag{19}$$

From the numerator, the sign of the derivative is determined by

$$V = D_{LR} + r_f(\tau_L)Z \tag{20}$$

Taking the derivative of *V* with respect to $\tau_L$, we get

$$\frac{\partial V}{\partial \tau_L} = \frac{\partial D_{LR}}{\partial \tau_L} + \frac{\partial r_f(\tau_L)}{\partial \tau_L} Z - r_f(\tau_L)\frac{\partial Z}{\partial \tau_L} = \frac{\partial r_f(\tau_L)}{\partial \tau_L} Z < 0 \tag{21}$$

Under the given condition, *V* is strictly decreasing in $\tau_L$ and *A<sub>LR</sub>* is strictly concave in $\tau_L$ as well. The availability $A_{LR}$ is maximized when *V*=0. Consequently, the optimum combination of $\tau_L$ and $\tau_R$ which maximizes $A_{LR}$ is given by the solution of the system of equations *W*=0 and *V*=0. □

The optimum life-extension interval and rejuvenation interval cannot be expressed in a symbolic manner, but they can be computed through a numerical approach like a gradient search method used for analyzing the optimum rejuvenation intervals in a virtualized system [13]. Numerical examples are presented in Section V.

## IV. Job completion time analysis

In this section, we analyze the distribution of completion times of jobs running on the software system based on the SMPs presented in the previous section. As a result of the preventive maintenance operations like software rejuvenation and life-extension, the software execution status changes and the executing job is preempted at the beginning of the new state. Performing software rejuvenation causes a preemptive-repeat (PRT) [15] discipline in which the job restarts from the beginning. In contrast, software life-extension does not lose its execution status; it thus follows a preemptive-resume (PRS) [15] discipline wherein the job resumes at the point it was preempted. The job completion time to process all the requested work is clearly affected by the preemption type as well as the state transitions. Here, we will perform a job completion time analysis on aging software without any preventive operations and one on software with a time-based life-extension.

### A. Aging software

If the software system suffering from aging is not controlled with any preventive operations, its behavior can be captured by a two state model composed of an up state and a down state. Let $F_f(t)$ and $G_{rc}(t)$ denote the failure time distribution and recovery time distribution. Since the job execution is dropped when the system goes down, the down state is considered to be a PRT state. Once the job is interrupted in the down state, it needs to be restarted after recovery.

Define $\mathrm{T}(x)$ to be the amount of time needed to complete a job with a work requirement of *x* units. Suppose that the execution environment processes a work unit in an hour, $\mathrm{T}(x)$ is equal to *x* if the job started in the up state completes before the first failure occurrence. If the job encounters a failure at time *h* (>0), the job needs to be restarted after the system recovers. In this case, the total job execution time becomes the sum of *h*, recovery time, and $\mathrm{T}(x)$. Taking the Laplace-Stieltjes transform (LST) with respect to *t*, the LST of the job completion time $\Phi_T^{\sim}(s,x)$ satisfies the following equation:

$$\Phi_T^{\sim}(s,x)|_h = \begin{cases} e^{-sx}, & h \geq x \\ e^{-sh} G_{rc}^{\sim}(s)\Phi_T^{\sim}(s,x), & h < x \end{cases} \tag{22}$$

where $G_{rc}^{\sim}(s)$ is the LST of $G_{rc}(t)$. Unconditioning on *h*, $\Phi_T^{\sim}(s,x)$ is expressed as

$$\begin{aligned}\Phi_T^{\sim}(s,x) &= e^{-sx}\int_x^{\infty} dF_f(h) \\ &\quad + G_{rc}^{\sim}(s)\Phi_T^{\sim}(s,x)\int_0^x e^{-sh} dF_f(h) \\ &= \frac{e^{-sx}\left(1 - F_f(x)\right)}{1 - G_{rc}^{\sim}(s)\int_0^x e^{-sh} dF_f(h)}\end{aligned} \tag{23}$$

The expected job completion time is computed from the moment generation property of LST [14]:

$$E\big(T(x)\big) = -\left.\frac{\partial \Phi_T^{\sim}(s,x)}{\partial s}\right|_{s=0} \tag{24}$$

Now let us consider a system using software rejuvenation to prevent a failure caused by software aging. The system state transition can be captured by the SMP shown in Section III.A. We assume that the job execution begins immediately after restarting the execution environment (in state 0). In this system, the rejuvenation time interval $\tau_R$ must be larger than the work requirement *x*; otherwise, the job never completes. Under this condition $\tau_R > x$, the job completion time distribution is the same as that without rejuvenation. Therefore, the LST of the job completion time is represented by (23), and the expected job completion time can be computed from (24).

### B. Job completion time in the case of life-extension

Next, we analyze the job completion time on a system using life-extension. The system behavior follows the state transitions specified in Section III.B. Again, we assume that the job execution begins just after restarting the execution environment (in state 0). When the life-extension interval $\tau_L$ is larger than *x*, the job never runs in a life-extended state (State 1), and the job completion time is the same as in the case of the aging system studied in Section IV.A. In the case of $\tau_L \leq x$, there are two scenarios in which the job is dropped as a result of a software failure: the software execution fails 1) before life-extension and 2) after life-extension.

Conditioned by the failure time *h*, the LST of job completion time is represented by

$$\Phi_{TL}^{\sim}(s,x)|_h = \begin{cases} e^{-sx}, & h \geq x \\ e^{-sh} G_{rc}^{\sim}(s)\Phi_{TL}^{\sim}(s,x), & h < x \end{cases} \tag{25}$$

Because the failure time distribution changes to $F_e(t)$ after software life-extension at $t = \tau_L$, the LST of the job completion time $\Phi_{TL}^{\sim}(s,x)$ is

$$\Phi_{TL}^{\sim}(s,x) = e^{-sx}\left(1 - F_f(\tau_L)\right)\int_x^{\infty} dF_e(h-\tau_L) + G_{rc}^{\sim}(s)\Phi_{TL}^{\sim}(s,x)\left[\int_0^{\tau_L} e^{-sh} dF_f(h) + \left(1 - F_f(\tau_L)\right)\int_{\tau_L}^{x} e^{-sh} dF_e(h-\tau_L)\right]$$

$$= \frac{e^{-sx}\left(1 - F_f(\tau_L)\right)\left(1 - F_e(x-\tau_L)\right)}{1 - G_{rc}^{\sim}(s)\Psi_{TL}^{\sim}(s,x)}, \quad (26)$$

where

$$\Psi_{TL}^{\sim}(s,x) = \int_0^{\tau_L} e^{-sh} dF_f(h) + \left(1 - F_f(\tau_L)\right)\int_{\tau_L}^{x} e^{-sh} dF_e(h-\tau_L)$$

The above analysis does not take into account the performance degradation after the software life-extension. If the job processing rate decreases in the life-extended state (i.e., State 1) by $r (0 < r \le 1)$, $\Phi_{TL}^{\sim}(s,x)$ is expressed as

$$\Phi_{TL}^{\sim}(s,x) = \frac{e^{-s\left(\tau_L + \frac{x-\tau_L}{r}\right)}\left(1 - F_f(\tau_L)\right)\left(1 - F_e\left(\frac{x-\tau_L}{r}\right)\right)}{1 - G_{rc}^{\sim}(s)\Psi_{TL}^{\sim}(s,x)} \quad (27)$$

where

$$\Psi_{TL}^{\sim}(s,x) = \int_0^{\tau_L} e^{-sh} dF_f(h) + \left(1 - F_f(\tau_L)\right)\int_{\tau_L}^{\tau_L + \frac{x-\tau_L}{r}} e^{-s\left(\tau_L + \frac{h-\tau_L}{r}\right)} dF_e(h-\tau_L)$$

The above expression is a generalization of (26) and it becomes identical to (26) when $r$=1. The expected job completion time is obtained as

$$E\left(T(x)\right) = -\left.\frac{\partial \Phi_{TL}^{\sim}(s,x)}{\partial s}\right|_{s=0} \quad (28)$$

The job completion time distribution in the hybrid system model in Section III.C is also characterized by (28), provided that the sum of $\tau_L$ and $\tau_R/r$ is larger than or equal to the work requirement $x$. If $\tau_L + \frac{\tau_R}{r} < x$, the rejuvenation always takes place before job completion and hence the job can never complete.

## V. Numerical Examples

Our numerical experiments aim to compare the software rejuvenation, life-extension, and hybrid approaches. For the software system suffering from software aging, we first analyze the optimum intervals for software rejuvenation and life-extension that maximize the steady-state system availability. Next, we study the impact on the job completion time on the basis of the model presented in Section IV.

Since the failure rate affected by software aging increases over time, we assume that the failure time distribution is IFR. The hypo-exponential distribution is known to be an IFR distribution regardless of its parameter values, and it has been widely used for modeling software aging (e.g., [2][8][11]). We thus define the failure time distributions as two-stage hypo-exponential distributions, $F_f(t) = HYPO(\lambda_1, \lambda_2)$ and $F_e(t) = HYPO(\lambda_1, \lambda_3)$ where $\lambda_1$, $\lambda_2$ and $\lambda_3$ are parameters that satisfy $\lambda_2 > \lambda_3$. The recovery time distribution and rejuvenation time distribution are assumed to be exponential with rates $\mu$ and $\alpha$, respectively. The parameter values used in the examples are summarized in TABLE I; most of them are taken from [16].

TABLE I. Default Parameter Values

| Parameters | Values | Descriptions |
|---|---|---|
| $\lambda_1$ | 0.002976190 [1/h] | Parameters for failure time distributions |
| $\lambda_2$ | 0.005952381 [1/h] | |
| $\lambda_3$ | 0.001984127 [1/h] | |
| $\mu$ | 1 [1/h] | Reactive recovery rate |
| $\alpha$ | 12 [1/h] | Rejuvenation rate |
| $x$ | 360 [units] | Amount of work requirements |

### A. *Steady-state availability*

Figure 7 plots the steady-state availability achieved by time-based software rejuvenation and time-based software life-extension for default parameter values and varying the maintenance intervals. There exists an optimum rejuvenation interval, since $F_f(t)$ is IFR, $\mu < \alpha, \frac{dA(0)}{d\tau_R} > 0$, and $\frac{dA(\infty)}{d\tau_R} < 0$ [11]. From Theorem 1, an optimum life-extension interval also exists. TABLE II lists the optimum rejuvenation and life-extension intervals and the corresponding steady-state availabilities.

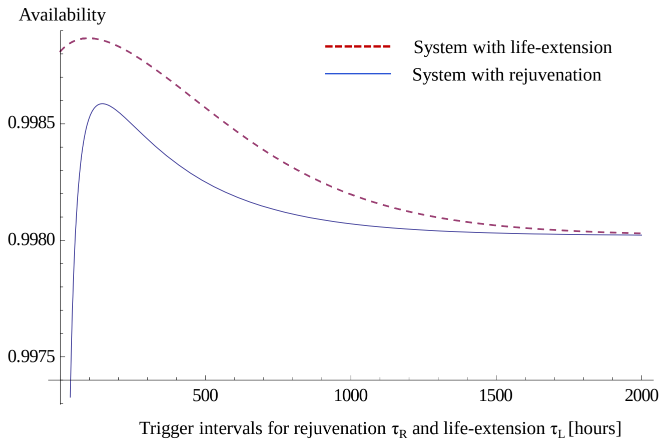

Figure 7. Steady-state availabilities achieved by software rejuvenation or life-extension

TABLE II. Optimum time interval and maximum availability

| Operation | Optimum interval [hours] | Maximum availability |
|---|---|---|
| Rejuvenation | 144.936 | 0.99858628 |
| Life-extension | 99.612 | 0.99886750 |

As can be seen, rejuvenation and life-extension have different optimum intervals that maximize system availability. With the default parameter values, software life-extension achieves higher availability than rejuvenation. However, software rejuvenation potentially achieves higher availability than life-extension, for example when the rejuvenation rate $\alpha$ is high. The impacts of $\alpha$ and $\lambda_3$ on the maximum system availability are analyzed by a sensitivity analysis below.

For a system using only software rejuvenation, the optimum rejuvenation interval depends on the rejuvenation rate $\alpha$. Figure 8 plots the maximum availability versus $\alpha$, where each plot is labeled with the optimum interval $\tau_R^*$.

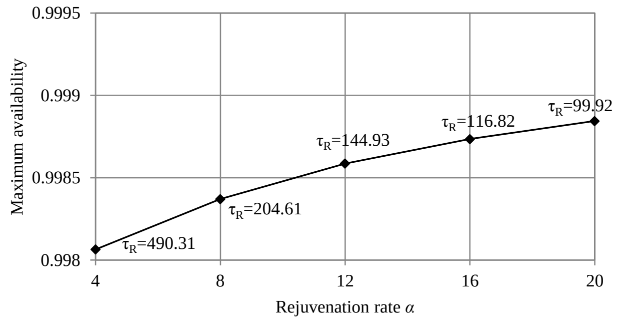

Figure 8. Sensitivity to rejuvenation rate on maximum availability

As $\alpha$ increases, the downtime overhead due to rejuvenation decreases; thus, the optimum rejuvenation interval becomes shorter and the maximum steady-state availability increases. Similarly, the optimum life-extension interval depends on $\lambda_3$, which determines the failure time distribution in the life-extended state. Since the mean time to failure given $HYPO(\lambda_1, \lambda_3)$ is $\frac{1}{\lambda_1} + \frac{1}{\lambda_3}$, a larger $\lambda_3$ shortens the lifetime.

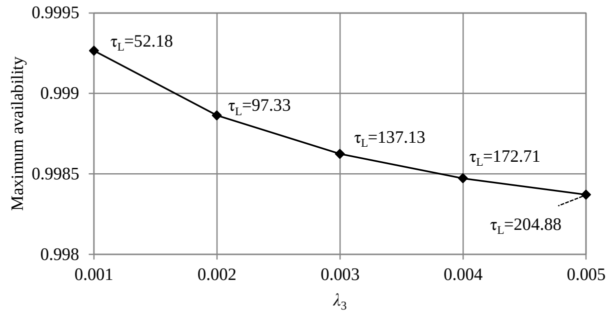

Figure 9. Sensitivity of maximum availability to parameter $\boldsymbol{\lambda_3}$

Figure 9 plots the maximum availability values achieved by the optimum life-extension intervals $\tau_L^*$ versus $\lambda_3$. As $\lambda_3$ increases, the optimum life-extension interval gradually increases and the maximum availability consequently decreases.

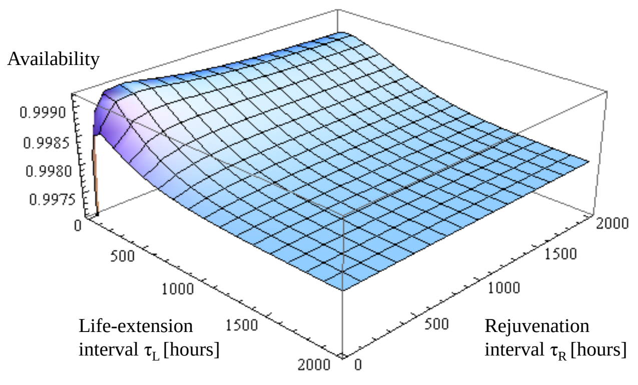

Figure 10. Steady-state system availability with hybrid approach

Next, we consider the hybrid model that contains two control parameters: a life-extension interval $\tau_L$ and rejuvenation interval $\tau_R$. Figure 10 plots the steady-state availability values computed by varying $\tau_L$ and $\tau_R$ from one hour to 2000 hours.

From Theorem 2, there exists a unique optimum combination of $\tau_R$ and $\tau_L$ at which steady-state availability is maximized. In this example, the maximum steady-state availability is 0.99926 which is achieved at $(\tau_L, \tau_R) = (52.7, 207.9)$. Thus, the hybrid approach potentially has higher system availability compared with the individual approaches.

## B. *Job completion time*

Suppose the software system starts processing a job with work requirements $x$ at the beginning of the up state. The job completion time distribution can be characterized by either (23) or (26) depending on the preemption type. Since $\Phi_{\tilde{T}}(s,x)$ and $\Phi_{\widetilde{TL}}(s,x)$ are in LST form, we take numerical inversion using a Mathematica library [17]. We set the performance degradation rate $r$ in the life-extended state to 1.0, 0.8, or 0.5 and compare the results.

Figure 11 shows the resulting job completion time distributions for the aging system without life-extension and the one using software life-extension with different degradation rates. The life-extension interval is set to 99.612, which is the optimum interval obtained in Section 5.1.

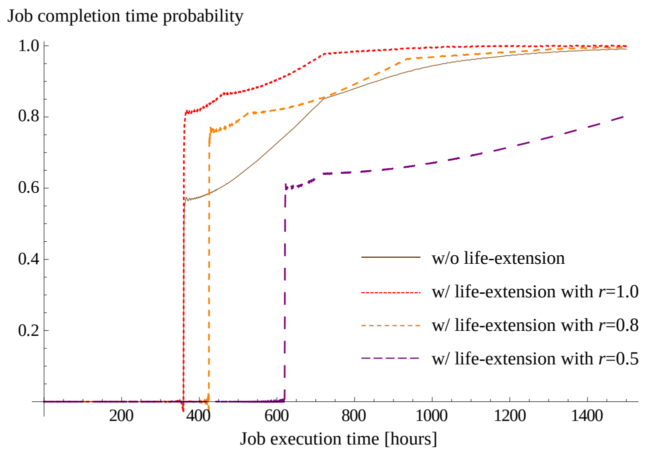

Figure 11. Job completion time distributions for aging system and the system with life-extension

If there is no performance degradation after the life-extension (i.e., $r$=1.0), the system with life-extension can clearly complete the job faster than the system without life-extension. Note that it takes at least 360 hours to complete the job, since the work processing rate is 1 unit/hour in both the cases. When the performance degradation occurs after life-extension, the minimum job completion time is prolonged accordingly. The more significant the degradation in processing rate, the longer the minimum job completion time becomes. Although life-extension has still an advantage when $r$=0.8, it is no more beneficial than the system without life-extension when $r$=0.5. This means re-execution after a failure without life-extension is more effective than extending the software execution by reducing the processing rate.

Therefore, the performance degradation rate $r$ is critical to designing an efficient software life-extension as far as the job completion time is concerned.

Since the completion time performance is influenced by the life-extension interval $\tau_L$, the impact of the interval can be evaluated in terms of the mean job completion time (28). Figure 12 plots the mean job completion times versus $\tau_L$.

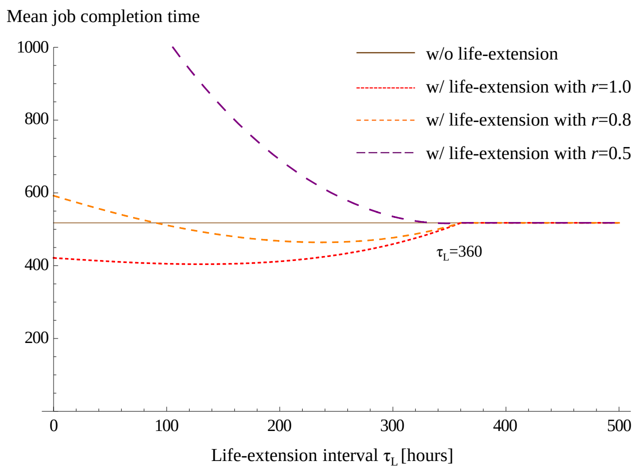

Figure 12. Mean job completion time versus life-extension interval

When $\tau_L$ is larger than $x = 360$, life-extension is never applied during the job execution, and hence, the mean job completion time is the same as the mean of the aging system without life-extension (517.808 hours). Interestingly, if $\tau_L \leq x$, there is an optimum life-extension interval that minimizes the mean job completion time. If there is no performance degradation ($r$=1.0), the mean job completion time is minimized at $\tau_L = 130.25$, at which point the job completion time is 404.2 hours. This is the optimum life-extension interval in terms of the mean job completion time. In the case of the performance degradations with $r$=0.8 and 0.5, the optimum intervals are $\tau_L = 236.49$ and $347.53$, and the corresponding minimum mean job completion times are 464.19 and 516.42 hours.

The results of our numerical experiments show that the optimum intervals for preventive maintenance (rejuvenation, life-extension or both) differ depending on the measure of interest (system availability or mean job completion time) as well as the parameter values. The proposed analytical approaches are useful for designing systems with those maintenance operations.

## VI. Related work

Software aging affects a wide variety of software-based infrastructure systems that tend to operate for long periods of time. The early studies characterized the phenomena in software products such as Apache web server [18], Linux operating system [19], and Java Virtual Machine [20], as well as telecommunication systems [21]. Recently, software aging has been studied in advanced software systems like Xen-based server virtualized systems [7][22], cloud computing systems using Eucalyptus [23], and database systems using MySQL server [24]. It is typically caused by aging-related software faults [1], and researchers have compiled statistics on such faults by making thorough investigations of problem reports. In particular, an investigation of problem reports on NASA's space missions revealed that 4.4% of bugs can be classified as aging-related software faults [25]. Aging-related bugs in open-source software of cloud computing systems have also been investigated [26]. Investigation of bug reports was also used to identify the software complexity metrics that can be used for predicting the location of aging-related faults [27].

The mitigations against software aging can be categorized according to the phases of software development, namely the development phase and the operational phase. Removing the root cause of software aging in the development phase is challenging, because it takes a long time to observe software aging, and one also faces problematic false alarms. Accelerated life-time test [28] and degradation test [29] are capable of reducing the time needed to observe software aging phenomena. However, false alarms of aging detection due to benign trends in resource usage are still an unsolved issue [30]. The impact of software testing on the operational behavior of software systems suffering from aging was studied in [31]. Even with state-of-art testing and debugging techniques, complete removal of aging-related software faults is practically infeasible or unacceptably expensive. Instead of removing the bugs in the development phase, it may be more cost-effective to remove them in the operational phase, as in the case of software rejuvenation. Software rejuvenation was first presented in [2], and subsequently, a number of researchers have come up with analytical models to quantify its effectiveness in terms of system availability [8], performance [31], and job completion time [16]. Software rejuvenation involves additional downtime costs, and variations of it have been presented for cluster systems [33], virtual machine-based systems [7][13][34] and Linux operating systems [35]. Software life-extension [3] is a new countermeasure that is completely different from rejuvenation. In this paper, we proposed a judicious combination of software life-extension and rejuvenation to maximize system availability.

Since software life-extension allows software to continue execution even as it ages, the concept bears some similarity to failure-oblivious computing [36], which neglects errors in favor of continuing the execution. Google admits the effectiveness of failure-oblivious computing and use this concept in their parallel data analysis by Sawzal [37]. Rx is another safe survival technique that uses checkpoints and re-execution in a modified environment [38]. In contrast to these failure survival techniques, we focus on software life-extension as preventive maintenance to postpone the time to failure.

## VII. Discussion

VM-based software life-extension, as explained in Section II.D, is a general approach as it can apply any kind of application running on a VM. However, it should be noted that the application also needs to recognize the added resources and use them for the runtime. If an application does not recognize the dynamically added resource for its process, the life-extension does not help prolong the lifetime. For example, applications running on Java Virtual Machine (JVM) cannot take advantage of dynamically added memory because JVM's maximum heap size is set during initialization and it cannot be modified during execution. This could restrict the application domain of VM-based software life-extension.

As discussed in Section II.B, software life-extension can be achieved through dynamic workload control as well. In this approach, it is important to analyze the relationship between workload and aging rate so that we can determine the level of workload assignment. This can be done by characterizing the workload-aging relationship [6] in the software operational process. The relationship between the workload intensity and the affected lifetime can be estimated by performing accelerated degradation tests [29] and accelerated life tests [28]. Those techniques can complement our technique to determine the optimum life-extension interval.

Our life-extension model introduces a life-extended state that is distinct from the original state and assumes that the failure time distribution changes in the new state. SMP allows us to use any kind of distribution for failure time distribution. However, as mentioned previously, the Markov property assumed on every state transition in SMP may restrict the applications of the proposed model. The amount of accumulated errors in the robust state is not conserved in the life-extended state. If the failure time distribution depends on the time spent in the robust state, one approach we may take is to expand the model by reliability-conservation principle [39]. According to this principle, the reliability at the time of life-extension, $1 - F_f(\tau_L)$, is preserved in the extended state. Let $F_{f2}(t)$ be the failure time distribution in the life-extended state. There exists a time $\widehat{\tau_L}$ that satisfies $1 - F_f(\tau_L) = 1 - F_{f2}(\widehat{\tau_L})$. The failure time distribution when software life-extension is applied at $\tau_L$ is represented as $F_{f2}(t + \widehat{\tau_L})$ that depends on the time spent in the robust state $\tau_L$. Note that the resulting state model is no longer SMP, since the process does not regenerate in the life-extension state (i.e., the transition probabilities in the life-extension state depend on the time spent in the previous states). Alternatively, we may also rely on approximation model instead of explicitly modeling the time-dependent failure distribution. The approximated SMP model can be constructed from empirical data for lifetime in the life-extended state. Once we obtain the approximated SMP model, we can apply the same optimization scheme.

Another important future direction is to extend our model by incorporating the cost of software life-extension. In the current model, the cost factor is indirectly included as performance degradation of the life-extended application. The cost of life-extension depends on the implementation (e.g., additional resource usage costs, performance degradations, and degradations to the availability of other services). Incorporating such factors explicitly in the model can yield more comprehensive optimization problem. For instance, if the server memory is shared by another application on the same server, the life-extension may reduce the memory allocation to other application which results in performance degradation. To incorporate such cost for determining the optimum life-extension interval, we need to model the state transition and performance of the other application in addition to the life-extended application.

## VIII. Conclusion

We have analytically shown the effectiveness of counteracting software aging by extending the lifetime of software execution. On the basis of the semi-Markov process capturing the behavior of the system, we show of the condition where there is an optimum software life-extension interval in which the system availability is maximized. Our numerical experiments reveal that life-extension is comparable to rejuvenation in terms of system availability, while it can significantly shorten the job completion time. Considering both of system-availability and job completion time performance, the hybrid approach in which software life-extension followed by rejuvenation turns out to be the best strategy.

## Acknowledgment

The authors would like to thank Kishor Trivedi of Duke University for his technical advice on modeling and analysis for software aging and rejuvenation.

## References

[1] M. Grottke and K. S. Trivedi, Fighting bugs: Remove, retry, replicate and rejuvenate, IEEE Computer, vol. 40, no. 2, pp. 107-109, 2007.
[2] Y. Huang, C. Kintala, N. Kolettis, N.D. Fulton, Software rejuvenation: analysis, module and applications, In Proc. of 25th Symp. on Fault Tolerant Computing (FTCS-25), pp.381–390, 1995.
[3] F. Machida, J. Xiang, K. Tadano and Y. Maeno, Software life-extension: a new countermeasure to software aging, In Proc. of IEEE 23rd Int'l Symp. on Software Reliability Engineering (ISSRE), pp.131-140, 2012.
[4] K. S. Trivedi, R. K. Mansharamani, D. Kim, M. Grottke, M. Nambinar, Recovery from Failures Due to Mandelbugs in IT Systems, In Proc. of Pacific Rim Int'l Symp. on Dependable Computing (PRDC), pp.224-233, 2011.
[5] K. Vaidyanathan and K. S. Trivedi, A measurement-based model for estimation of resource exhaustion in operational software systems, In Proc. of Int'l Symp. on Software Reliability Engineering (ISSRE), pp. 84-93, 1999.
[6] A. Bovenzi, D. Cotroneo, R. Pietrantuono, S. Russo, Workload characterization for software aging analysis, In Proc. of Int'l Symp. on Software Reliability Engineering (ISSRE), pp. 240-249, 2011.
[7] K. Kourai, Fast and correct performance recovery of operating systems using a virtual machine monitor, In Proc. of Int'l Conf. on Virtual execution environments (VEE11), pp.99-110, 2011.
[8] S. Garg, A. Puliafito, M. Telek, and K. S. Trivedi, Analysis of software rejuvenation using Markov regenerative stochastic Petri nets, In Proc. of Int'l Symp. on Software Reliability Engineering (ISSRE), pp. 180-187, 1995.
[9] H. Suzuki, T. Dohi, K. Goseva-Popstojanova, and K. S. Trivedi, Analysis of multi-step failure models with periodic software rejuvenation, Advances in Stochastic Modelling (J. R. Artalejo and A. Krishnamoorthy, eds.), Notable Publications, Inc., pp. 85-108, 2002.
[10] D. Chen and K. S. Trivedi, Analysis of periodic preventive maintenance with general system failure distribution, in Proc. of Pacific Rim Dependable Computing (PRDC), pp. 103-107, 2001.
[11] T. Dohi, K. Goseva-Popstojanova, and K. S. Trivedi, Estimating software rejuvenation schedules in high assurance systems, Computer Journal, vol. 44, no. 6, pp. 473-482, 2001.
[12] J. Zhao, Y. Wang, G. Ning, K. S. Trivedi, R. Matias Jr., and K. Y. Cai, A comprehensive approach to optimal software rejuvenation, Performance Evaluation, vol. 70, no. 11, pp. 917-933, 2013.

[13] F. Machida, D. Kim, and K. S. Trivedi, Modeling and analysis of software rejuvenation in a server virtualized system with live VM migration, Performance Evaluation, vol. 70, no. 3, pp. 212-230, 2012.

[14] K. S. Trivedi, Probability and Statistics with Reliability, Queuing, and Computer Science Applications, John Wiley & Sons, 2nd edition, 2001.

[15] V. G. Kulkarni, V. V. Nicola and K.S. Trivedi, The completion time of a job on multimode systems, Advances in Applied Probability, vol. 19, pp. 932-954, 1987.

[16] F. Machida, V. F. Nicola, and K. S. Trivedi, Job completion time on a virtualized server with software rejuvenation, ACM J. on Emerging Technologies in Computing Systems, vol. 10, no. 1, 10, 2014.

[17] A. Mallet, Numerical Inversion of Laplace Transform, in Wolfram Library Archive, http://library.wolfram.com/infocenter/MathSource/2691/, 2000.

[18] M. Grottke, L. Lie, K. Vaidyanathan, and K. S. trivedi, Analysis of software aging in a web server, IEEE Trans. Reliability, vol. 55, no. 3, pp.411-420, 2006.

[19] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, Software aging analysis of the Linux operating system, In Proc. of Int'l Symp. Software Reliability Engineering (ISSRE), pp. 71-80, 2010.

[20] D. Cotroneo, S. Orlando, R. Pietrantuono, and S. Russo, A measurement-based ageing analysis of the JVM, Software Testing, Verification and Reliability, vol. 23, no. 3, pp. 199-239, 2013.

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

[22] F. Machida, J. Xiang, K.Tadano and Y. Maeno, Combined Server Rejuvenation in a Virtualized Data Center, In Proc. of 9th Int'l Conf. on Autonomic and Trusted Computing (ATC), pp.486-493, 2012.

[23] J. Araujo, R. S. Matos, V. Alves, P. R. M. Maciel, F. V. Souza, R. Matias Jr., and K. S. Trivedi, Software aging in the eucalyptus cloud computing infrastructure: Characterization and rejuvenation, ACM J. Emerging Technologies in Computing Systems, vol. 10, no. 1, 11, 2014.

[24] A. Bovenzi, D. Cotroneo, R. Pietrantuono, S. Russo, On the aging effects due to concurrency bugs: a case study on MySQL, In Proc. of IEEE 23rd Int'l Symp. on Software Reliability Engineering (ISSRE), pp.211-220, 2012.

[25] M. Grottke, A. P. Nikora, K. S. Trivedi, An empirical investigation of fault types in space mission system software, In Proc. of 40th IEEE/IFIP Int'l Conf. on Dependable Systems and Networks (DSN), pp. 447-456, 2010.

[26] F. Machida, J. Xiang, K. Tadano and Y. Maeno, Aging-related bugs in cloud computing software, In Proc. of 4th Int'l Workshop on Software Aging and Rejuvenation (WoSAR), 2012.

[27] D. Cotroneo, R. Natella, R. Pietrantuono, Predicting aging-related bugs using software complexity metrics, Performance Evaluation, vol. 70, no. 3, pp. 163-178, 2013.

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

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

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

[31] M. Grottke and B. Schleich, How does testing affect the availability of aging software systems?, Performance Evaluation vol. 70, no.3, pp. 179-196, 2013.

[32] D. Wang, W. Xie, and K. S. Trivedi, Performability analysis of clustered systems with rejuvenation under varying workload, Performance Evaluation, vol. 64, no. 3, pp. 247-265, 2007.

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

[34] L. Silva, H. Madeira and J. G. Silva, Software aging and rejuvenation in a soap-based server, In Proc. of the Fifth IEEE Int'l Symp. on Network Computing and Applications (NCA 2006), pp. 56-65, 2006.

[35] K. Yamakita, H. Yamada, and K. Kono, Phase-based reboot: Reusing operating system execution phases for cheap reboot-based recovery, In Proc. of 41st Int'l Conf. on Dependable Systems Networks (DSN), pp. 169-180, 2011.

[36] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T, Leu, and W. S. Beebee Jr., Enhancing server availability and security through failure-oblivious computing, In. Proc. of USENIX Symp. on Operating Systems Design and Implementation (OSDI), 2004.

[37] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, Interpreting the data: Parallel analysis with Sawzall, Scientific Programming, Vol. 13, No. 4, pp.277-298, 2005.

[38] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, Rx: Treating bugs as allergies: A safe method to survive software failures, In Proc. of Symp. on Operating Systems Principles (SOSP), pp. 235-248, 2005.

[39] D. Kececioglu, Reliability Engineering Handbook (Vol. 1 and 2). Prentice-Hall, Inc., 1991.

**Fumio Machida** received the B.S. and M.S. degrees from Tokyo Institute of Technology in 2001 and 2003, respectively. He is a researcher at NEC Corporation. He was a visiting scholar in the Department of Electrical and Computer Engineering at Duke University in 2010. He was a recipient of the young scientists' prize of Japan in 2014. His research interests include modeling and analysis for system dependability, software aging and rejuvenation, and virtualization of systems and networks. He is a member of the IEEE and the IEEE Computer Society.

**Jianwen Xiang** received PhD degrees from Wuhan University, and Japan Advanced Institute of Science and Technology (JAIST) in 2004, and 2005, respectively. He is a currently a Professor of the School of Computer Science and Technology of Wuhan University of Technology and he was a researcher at NEC Corporation. His research interests include dependable computing, formal methods, and software engineering.

**Kumiko Tadano** received the B.S., and M.S. degrees in physics and systems and information engineering from University of Tsukuba in 2005, and 2007, respectively. She is currently a researcher at NEC Corporation. Her research interests include modeling and analysis for disaster resilience of critical infrastructures.

**Yoshiharu Maeno** received the B.S., and M.S. degrees in physics from Tokyo University, Japan, and the Ph.D. degree in business science from Tsukuba University. He is a principal researcher at NEC Corporation. He is interested in stochastic processes, network theory, statistical analysis, complexity sciences, and their application to socio-economic design problems. He is a member of the IEEE, APS, and INSNA. He received the Young Researchers' Award from the IEICE..