Preventive Maintenance for Cloud-Based Software Systems Subject to Non-Constant Failure Rates

Jean Rahme and Haiping Xu
Computer and Information Science Department
University of Massachusetts Dartmouth, Dartmouth, MA 02747, USA
E-mail: {jrahme, hxu}@umassd.edu

Abstract—**Online applications, such as e-commerce, have made a huge impact in our daily life. With the rapid shift of online applications to cloud-based platforms in recent years, it becomes very important to maintain the high Quality of Service (QoS) for cloud-based software systems in order to support successful businesses. Since hardware reliability has been well understood and is typically guaranteed by the cloud providers, software failures have now become the major factor of system failures in cloud-based software systems. Correctly measuring the reliability and availability of a cloud-based software system is critical for making preventive maintenance schedules. In this paper, we address software-aging related bugs or faults that may lead to performance degradation or increased failure rates of system components. Based on our previous work, we study how to derive preventive maintenance schedules for cloud-based software systems subject to non-constant failure rates. We adopt the Weibull distribution to model an increasing failure rate for software components with software-aging issues. Finally, we use a case study to show that our analytical approach can effectively support development of software rejuvenation schedules for preventive maintenance of cloud-based software systems.**

Keywords-Software reliability engineering; software aging; reliability analysis; preventive maintenance; software rejuvenation schedule; non-constant failure rate

I. INTRODUCTION

Over the past decades, online applications have made a huge impact in our daily life. Cloud computing, the idea of using computing resources as a utility, has become an attractive paradigm for developers to deploy their services and get their services started, without the need to spend large capital in hardware resources. As the shift to cloud computing is rapidly increasing, there is a pressing need to maintain a high Quality of Service (QoS) for cloud-based systems to support successful online businesses. As hardware reliability is well understood and is typically guaranteed by the cloud providers [1], software faults in cloud services have become a major factor leading to system failures in cloud-based systems. Various strategies in Software Reliability Engineering (SRE) can be used to combat against software faults to achieve highly reliable software. Before the concept of software aging was introduced, SRE supported analysis of software defects and related to Heisenbugs or BohrBugs [2]. Bohrbugs are easier to deal with since they are deterministic, and can be eliminated at the design level by debugging or adopting design diversity. On the other hand, Heisenbugs are non-deterministic errors, which appear at the operational level, and can be dealt with by retrying the operation or restarting the associated application. However, neither of these two types of bugs would lead to an increasing failure rate; therefore, we typically assume constant failure rates for software components that are subject to these types of bugs [2, 3]. Software-aging related bugs are non-deterministic like Heisenbugs, thus, both of them are classified under Mandlebugs [3]. However, software-aging related bugs may result in an increased failure rate since the error conditions, such as unreleased memory due to memory leaks, can accumulate in a running application or within its environment (e.g., the operating system).

To deal with software aging and assure software fault tolerance, software rejuvenation process has been introduced as a proactive approach to counteracting software aging and maintaining a reliable software system [4]. In this work, we take advantage of cloud-based software design to perform rejuvenation in its simple form, namely to restart the application or its software components subject to software aging with increasing failure rates that would lead to the degradation of system performance. Fault tolerance and fault or failure forecasting are two major techniques that can be adopted side by side to improve the system reliability for an operational software system [5]. Fault tolerance in this work is achieved by employing standby Hot Software Spares (HSS); while failure forecasting is to estimate the failure-time probability density function (*pdf*) based on empirical data collected for the designed fault-tolerant system. In the context of cloud computing, HSS is a Virtual Machine (VM) instance that is available instantly when a primary component fails. The reliability of a cloud-based system can be computed by plugging the *pdf*s of its system components into a previously proposed analytical approach [6, 9], and then derive a software rejuvenation schedule for preventive maintenance. In this paper, we assume the time-to-failure *pdf* follows the Weibull distribution. By selecting appropriate parameters, we can model an increasing failure rate function due to software-aging related bugs. We show in a case study the ability of our analytical technique to evaluate the reliability of cloud-based systems with non-constant failure rates as well as fault-tolerant designs supported by either one HSS or two HSSs.

978-1-5386-0435-9/17/$31.00 ©2017 IEEE

II. Reliability Modeling and Analysis

Dynamic Fault Tree (DFT) has modeling capabilities for dynamic features of a computer-based system, such as spare components, functional dependency, and failure sequence dependency. In this paper, we adopt an extended DFT for modeling software spare components in cloud-based software systems [6]. The approach supports a two-phased software rejuvenation process, where Phase 1 is a pre-rejuvenation stage, and in Phase 2, system components in low performance are replaced by newly deployed ones. In particular, a Software SPare (SSP) gate is used to model the fault-tolerant aspect of a system design that employs one or multiple HSSs. It is important to mention that a DFT can be decomposed into independent sub-modules (sub-trees), so their reliabilities can be calculated independently, and then joined to derive the reliability of the whole system [7]. In the following two subsections, we show the modeling and analysis approach for spare components with either 1-HSS or 2-HSSs that follow the Weibull distribution to simulate their non-constant failure rates. A 2-parameter Weibull distribution has the following two parameters: the shape parameter "p" and the scale parameter "λ" [8], as given in Eq. (1) for its *pdf*:

$$f(t) = p\lambda^{p} t^{(p-1)} e^{-(\lambda t)^{p}} \tag{1}$$

The reliability function $R(t)$ based on $f(t)$ can be derived as in Eq. (2). Consequently, we can derive the failure rate function $h(t)$ as in Eq. (3).

$$R(t) = \int_{t}^{\infty} f(t)dt = e^{-(\lambda t)^{p}} \tag{2}$$

$$h(t) = \frac{f(t)}{R(t)} = p\lambda^{p} t^{(p-1)} \tag{3}$$

Note that in a special case, when $p = 1$, $h(t) = \lambda$. This means the probability density function $f(t)$ becomes an exponential distribution, where λ is a constant failure rate.

A. An SSP Gate for Cloud-Based Systems with 1-HSS

Following the same model construct defined in previous work [9], a SSP gate with one primary component P and one HSS component H is illustrated in Fig. 1. A SSP gate fails when P and all other alternate spares (the only spare part in Fig. 1 is H) fail. When P fails, H takes over P's workload, and then behaves as H^* with $\lambda_{H^*} \geq \lambda_H$. This is due to the software-aging phenomenon when an HSS takes a full workload after it replaces the primary one. Based on Fig. 1, we now consider two disjoint paths that lead to the failure of the SSP gate, which are P fails before H (called path event p_1) and H fails before P (called path event p_2).

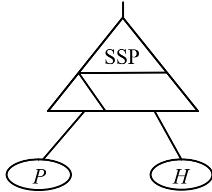

Fig. 1. An SSP gate with a primary component and a HSS

Path 1: P fails before H fails, denoted as $P \prec H$. Let τ_1 and τ_2 be the failure times of P and H, respectively. In this case, it is impossible for H to fail during $(0, \tau_1]$. Hence, the probability of P failing before H fails, i.e., $\Pr(p_1)$, can be calculated using double integrations as in Eq. (4).

$$\begin{aligned}\Pr(p_1) &= \int_{0}^{t} \int_{\tau_{H_1*}}^{(t-\tau_1+\tau_{H_1*})} f_P(\tau_1).f_{H*}(\tau_2) d\tau_2 d\tau_1 \\ &= \int_{0}^{t} \int_{\tau_{H_1*}}^{(t-\tau_1+\tau_{H_1*})} p^2 \lambda_P^p \lambda_{\mathrm{H}*}^p (\tau_1^{(p-1)} \tau_2^{(p-1)})(\lambda_{\mathrm{P}} e^{-(\lambda_{\mathrm{p}}\tau_1)^p})(\lambda_{\mathrm{H}*} e^{-(\lambda_{\mathrm{H}*}\tau_2)^p}) d\tau_2 d\tau_1\end{aligned} \tag{4}$$

where $\tau_{H_1*} = [(h_H(\tau_2))/(h_{H*}(\tau_2))]\tau_1 = (\lambda_H/\lambda_{H*})^p \tau_1$.

Path 2: H fails before P fails, denoted as $H \prec P$. In this case, it is impossible for P to fail during $(0, \tau_2]$, where τ_2 is the failure time of H. Hence the probability of H failing before P fails, i.e., $\Pr(p_2)$, can be calculated as in Eq. (5).

$$\Pr(p_2) = \int_{0}^{t}\int_{\tau_2}^{t} p^2 \lambda_P^p \lambda_{\mathrm{H}}^p (\tau_1^{(p-1)} \tau_2^{(p-1)})(\lambda_{\mathrm{H}} e^{-(\lambda_{\mathrm{H}}\tau_2)^p})(\lambda_{\mathrm{P}} e^{-(\lambda_{\mathrm{p}}\tau_1)^p}) d\tau_1 d\tau_2 \tag{5}$$

The reliability function $R(t)$ for a SSP gate with 1-HSS is given in a general form as $R(t) = 1- U(t)$, where $U(t)$ is given as in Eq. (6). Refer to the detailed derivation of Eq. (6) in previous work [9].

$$U(t) = \Pr(p_1) + \Pr(p_2) \tag{6}$$

B. An SSP Gate for Cloud-Based Systems with 2-HSSs

Figure 2 shows a SSP gate with one primary component P and two HSS components H_1 and H_2. Similar to the previous case with a single HSS, P is initially powered on. When P fails, it is replaced by one of the HSSs depending on their enumeration order. An SSP gate fails when the primary component and all the alternate inputs fail. When H_1 takes the lead to replace P, it becomes H_1*, with $\lambda_{H_1*} \geq \lambda_{H_1}$, due to the software aging phenomenon when it takes the full workload. In this case, H_1* serves as a primary one, and H_2 serves as its hot software spare. Similarly, when H_1* fails, H_2 replaces H_1*, and behave as H_2*, with $\lambda_{H_2*} \geq \lambda_{H_2}$.

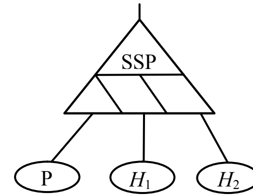

Fig. 2. An SSP gate with a primary component and two HSSs

Let τ_1, τ_2 and τ_3 be the failure times of component P, H_1 and H_2, respectively. We now identify all the possible paths that lead to the failure of a SSP gate according to the component failure sequence. To calculate the reliability function of an SSP gate, we investigate six disjoint paths (denoted as p_1 to p_6, respectively) as follows.

Path 1: The components fail in the sequence of P, H_1, and H_2, denoted as $P \prec H_1 \prec H_2$. In this case, it is impossible for H_1 to fail during $(0, \tau_1]$ and for H_2 to fail during $(0, \tau_2]$. The HSS H_1 takes over the workload and becomes H_1* right after P

fails; similarly, H_2 takes over the workload and becomes H_2* right after H_1* fails. Hence, the probability of the path event $P \prec H_1 \prec H_2$ = Pr(p_1) can be calculated as in Eq. (7) with τ_{H_2*} =$[(h_H(\tau_2))/(h_{H*}(\tau_2))](\tau_2+(\tau_2-\tau_{H_1*}))$, which is a generalized form of the equation $\tau_{H_2*} = (\frac{\lambda_{H_2}}{\lambda_{H_2*}})^p(\tau_2 + (\tau_1 - \tau_{H_1*}))$ for time shifting derived in previous work [6, 9].

$$\Pr(p_1) = \int_0^t \int_{\tau_{H1*}}^{(t-\tau_1+\tau_{H1*})} \int_{\tau_{H2*}}^{(t-[\tau_2+\tau_1-\tau_{H1*}]+\tau_{H2*})} f_P(\tau_1) f_{H_1*}(\tau_2) f_{H_2*}(\tau_3) d\tau_3 d\tau_2 d\tau_1$$
$$= \int_0^t \int_{\tau_{H1*}}^{(t-\tau_1+\tau_{H1*})} \int_{\tau_{H2*}}^{(t-[\tau_2+\tau_1-\tau_{H1*}]+\tau_{H2*})} p^3 \lambda_P^p \lambda_{H_1*}^p \lambda_{H_2*}^p (\tau_1^{(p-1)} \tau_2^{(p-1)} \tau_3^{(p-1)}) (\lambda_P e^{-(\lambda_P \tau_1)^p})(\lambda_{H_1*} e^{-(\lambda_{H_1*} \tau_2)^p})(\lambda_{H_2*} e^{-(\lambda_{H_2*} \tau_3)^p}) d\tau_3 d\tau_2 d\tau_1 \quad (7)$$

Path 2: The components fail in the sequence of P, H_2, and H_1, denoted as $P \prec H_2 \prec H_1$. Similar to previous work [9], the integration of H_1* requires to shift the integration limit from τ_{H_1*} to $\tau_{H_1*}+(\tau_3-\tau_1)$, which leads to Eq. (8).

$$\Pr(p_2) = \int_0^t \int_{\tau_1}^t \int_{\tau_{H_1*}+(\tau_3-\tau_1)}^{t-(\tau_1-\tau_{H_1*})} p^3 \lambda_P^p \lambda_{H_1*}^p \lambda_{H_2}^p (\tau_1^{(p-1)} \tau_2^{(p-1)} \tau_3^{(p-1)}) (\lambda_P e^{-(\lambda_P \tau_1)^p})(\lambda_{H_1*} e^{-(\lambda_{H_1*} \tau_2)^p})(\lambda_{H_2} e^{-(\lambda_{H_2} \tau_3)^p}) d\tau_2 d\tau_3 d\tau_1 \quad (8)$$

Path 3: The components fail in the sequence of H_2, P, and H_1, denoted as $H_2 \prec P \prec H_1$. Note that this case is a simple one similar to Eq. (4). The probability that the SSP gate fails can be calculated as in Eq. (9).

$$\Pr(p_3) = \int_0^t \int_{\tau_3}^t \int_{(\frac{\lambda_{H_1}}{\lambda_{H_1*}})^p \tau_1}^{t-(\tau_1-\tau_{H_1*})} p^3 \lambda_P^p \lambda_{H_1*}^p \lambda_{H_2}^p (\tau_1^{(p-1)} \tau_2^{(p-1)} \tau_3^{(p-1)}) (\lambda_P e^{-\lambda_P \tau_1})(\lambda_{H_1*} e^{-\lambda_{H_1*} \tau_2})(\lambda_{H_2} e^{-\lambda_{H_2} \tau_3}) d\tau_3 d\tau_1 d\tau_2 \quad (9)$$

Path 4: The components fail in the sequence of H_1, H_2, and P, denoted as $H_1 \prec H_2 \prec P$. In this case, it is impossible for P to fail during (0, τ_3]. The probability that the SSP gate fails during (0, t] can be calculated as in Eq. (10).

$$\Pr(p_4) = \int_0^t \int_{\tau_2}^t \int_{\tau_3}^t p^3 \lambda_P^p \lambda_{H_1}^p \lambda_{H_2}^p (\tau_1^{(p-1)} \tau_2^{(p-1)} \tau_3^{(p-1)}) (\lambda_P e^{-\lambda_P \tau_1})(\lambda_{H_1} e^{-\lambda_{H_1} \tau_2})(\lambda_{H_2} e^{-\lambda_{H_2} \tau_3}) d\tau_2 d\tau_3 d\tau_1 \quad (10)$$

Path 5: The components fail in the sequence of H_1, P, and H_2, denoted as $H_1 \prec P \prec H_2$. Similar to Path p_3, this is where H_1 fails first as a spare, then P fails before H_2 fails. In this case, the probability that the SSP gate fails can be calculated as in Eq. (11), where τ_{H_2*} can be calculated in a similar way to the calculation of τ_{H_1*} as in Eq. (4).

$$\Pr(p_5) = \int_0^t \int_{\tau_2}^t \int_{(\frac{\lambda_{H_2}}{\lambda_{H_2*}})^p \tau_2}^{t-(\tau_2-\tau_{H_2*})} p^3 \lambda_P^p \lambda_{H_1}^p \lambda_{H_2*}^p (\tau_1^{(p-1)} \tau_2^{(p-1)} \tau_3^{(p-1)}) (\lambda_P e^{-\lambda_P \tau_1})(\lambda_{H_1} e^{-\lambda_{H_1} \tau_2})(\lambda_{H_2*} e^{-\lambda_{H_2*} \tau_3}) d\tau_3 d\tau_1 d\tau_2 \quad (11)$$

Path 6: The components fail in the sequence of H_2, H_1, and P, denoted as $H_2 \prec H_1 \prec P$. In this case, the probability that the SSP gate fails during (0, t] can be calculated as in Eq. (12).

$$\Pr(p_6) = \int_0^t \int_{\tau_3}^t \int_{\tau_2}^t p^3 \lambda_P^p \lambda_{H_1}^p \lambda_{H_2}^p (\tau_1^{(p-1)} \tau_2^{(p-1)} \tau_3^{(p-1)}) (\lambda_P e^{-\lambda_P \tau_1})(\lambda_{H_1} e^{-\lambda_{H_1} \tau_2})(\lambda_{H_2} e^{-\lambda_{H_2} \tau_3}) d\tau_1 d\tau_2 d\tau_3 \quad (12)$$

The reliability function for a SSP gate with 2-HSSs is given in a general form as $R(t)$ = 1–$U(t)$, where $U(t)$ is given as in Eq. (13).

$$U(t) = \Pr(p_1) + \Pr(p_2) + \Pr(p_3) + \Pr(p_4) + \Pr(p_5) + \Pr(p_6) \quad (13)$$

It is worth noting that there are major differences between Eqs. (7-12) and the equations derived in previous work [9]. In Eqs. (7-12), the probabilities are calculated based on non-constant failure rates for the software components; while in previous work [9], the derived equations only work for constant failure rates. Though Eqs. (7-12) cannot be directly verified using Continuous Time Markov Chain (CTMC) due to the non-constant failure rates, they have been proved correct in reference [9] for the special case when the shape parameter p = 1, i.e., when the failure rates are constant values.

III. Case Study

In this section, we show how to model and analyze the reliability of a cloud-based software system with 1-HSS and 2-HSSs, where all software components have the Weibull time-to-failure distribution with increasing failure rates due to software aging. The software system is modeled using an extended DFT for software sparing [6, 9], and the reliability analysis is conducted as described in Sections II.A and II.B. Our goal is to derive feasible software rejuvenation schedules based on reliability quantitative analysis.

Figure 3 shows the extended DFT model of two cloud-based systems during the pre-rejuvenation stage, i.e., Phase 1. The model on the top contains a single HSS that is ready to replace the primary one when it fails; while the model at the bottom contains 2-HSSs to make the system more reliable and fault-tolerant. The cloud-based system being modeled consists of an application server PA and a database server PB. In the 1-HSS case, HA is set up for PA, and HB is set up for PB to assure high reliability. Similarly, in the 2-HSS case, two HSSs are deployed for each primary server. We assume the reliability threshold to be 0.99 as a minimum constraint for system reliability. In the case study, we define the following scale parameters: λ_{PA} = 0.004/day, λ_{HA_1}= λ_{HA_2} = 0.0025/day, λ_{PB} = 0.005/day, λ_{HB_1} = λ_{HB_2} = 0.003/day. For comparison purposes, we set them the same values as the constant failure rates of exponential distribution used in our previous work [9]

The failure rate of an HSS increases after switching to the primary-component mode when the primary one fails. Hence, $h_{PA}(\tau_1)$ = $h_{HA_1*}(\tau_2)$ = $h_{HA_2*}(\tau_3)$ and $h_{PB}(\tau_1)$ = $h_{HB_1*}(\tau_2)$ = $h_{HB_2*}(\tau_3)$. In summary, the following Weibull parameter values are used for reliability analysis in the case study: application server (shape p = 1.2; scale λ_{PA} = λ_{HA_1*} = λ_{HA_2*} = 0.004, and λ_{HA_1} = λ_{HA_2}= 0.0025); and database servers (shape p = 1.1; scale λ_{PB} = λ_{HB_1*} = λ_{HB_2*} = 0.005, and λ_{HA_1} = λ_{HA_2} = 0.003).

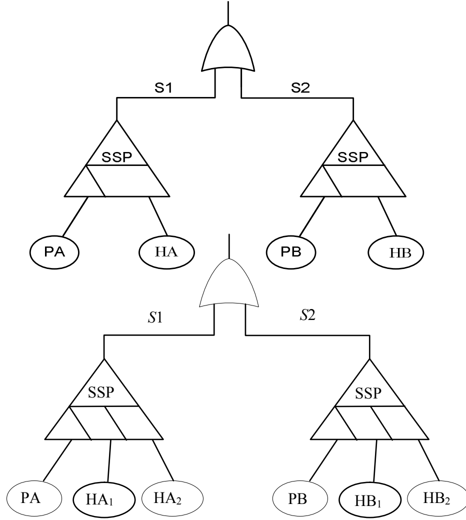

Fig. 3. DFT model with 1-HSS vs. 2-HSSs (Phase1) (adapted from [9])

Note that the rejuevantion process also involves Cold Software Spare (CSS) components, which are images of VM instances that can be easily deployed. Since a CSS is simply a cloud image that is not running, its failure rate equals 0. As such, a CSS does not appear in the DFT model because it does not affect the system reliability. We consider a CSS only when it is activated and deployed as a primary one or an HSS.

From Fig. 3, we can see that the system fails when either the application server or the database server fails. We use the sum of disjoint product method to derive the reliability function for an OR-gate, which can be applied to both of the two DFT models, as in Eq. (14).

$$R(t) = 1 - U_{OR}(t) = 1 - (U_{S1}(t) + (1 - U_{S1}(t)) * U_{S2}(t)) \quad (14)$$

In Eq. (14), the unreliability functions $U_{S1}(t)$ and $U_{S2}(t)$ can be derived using Eq. (6) and Eq. (13) for the 1-HSS and 2-HSSs cases, respectively. Both system-specific (Scenario 1) and component-specific (Scenario 2) rejuvenation approaches are addressed in the case study. As defined in reference [6], a system-specific rejuvenation schedule restarts the whole system when the system reliability reaches a safety threshold. On the other hand, a component-specific rejuvenation schedule only refreshes the most critical component when the system reliability is below the safety threshold.

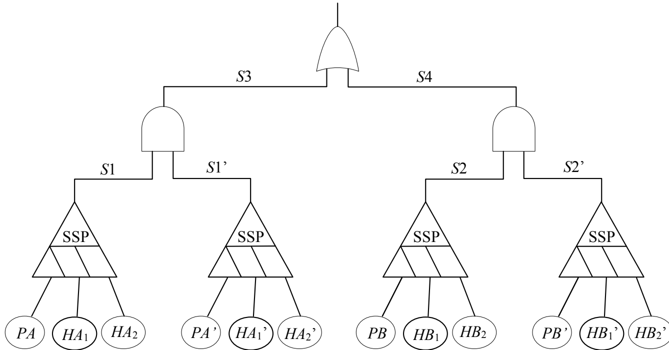

Fig. 4. DFT model with 2-HSSs - Phase 2 (Scenario1) (adapted from [9])

Figure 4 represents the DFT model of the cloud-based system with 2-HSSs in Phase 2 based on Scenario 1. Similar to the reliability analysis for Phase 1, we can analyze the DFT model for Phase 2 (Scenario1) by decomposing it into sub-trees. Thus, the unreliability functions of the subtrees $U_{S1}(t)$, $U_{S1'}(t)$, $U_{S2}(t)$ and $U_{S2'}(t)$ can be computed using Eq. (6) for 1-HSS and Eq. (13) for 2-HSSs. As for $U_{S3}(t)$ and $U_{S4}(t)$, since they are AND-gates, their unreliability can be calculated using the sum of disjoint product method as shown in Eqs. (15-16). Finally, the reliability of the whole system can be derived as in Eq. (14), similar to the case of Phase 1.

$$U_{S3}(t) = U_{S1}(t) * U_{S1'}(t) \quad (15)$$

$$U_{S4}(t) = U_{S2}(t) * U_{S2'}(t) \quad (16)$$

Once we have derived the reliability function for Scenario 1 in Phase 2, we can use the same approach to deal with Scenario 2 in Phase 2. The DFT model for Scenario 2 in Phase 2 is illustrated in Fig. 5, in which the application server is rejuvenated. Note that when the database server is rejuvenated, the DFT model can be derived in a similar way.

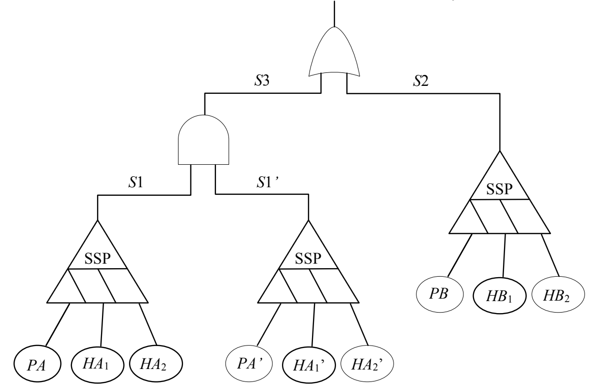

Fig. 5. DFT model with 2-HSSs - Phase 2 (Scenario2) (adapted from [9])

In Fig. 5, the subtrees $U_{S1}(t)$, $U_{S1'}(t)$, $U_{S2}(t)$ and $U_{S3}(t)$ can be calculated as we did for Scenario 1 in Phase 2. In other words, we can calculate $U_{S1}(t)$, $U_{S1'}(t)$ and $U_{S2}(t)$ according to Eq. (6) and Eq. (13) for the cases of 1-HSS and 2-HSSs, respectively. As node S3 represents the output of an AND-gate, $U_{S3}(t)$ is derived using the sum of disjoint product method for an AND-gate as in Eq. (15). Finally, the reliability function of the whole system is defined as in Eq. (14).

Different from previous work [9], all software components defined in the DFT models are subject to non-constant failure rates as their time-to-failure follows the Weibull distribution. The next step is to show the analysis results and visualize the differences and the impacts of employing 2-HSSs vs. 1-HSS on rejuvenation schedules in a cloud-based system. In addition, we study the impact of using Scenario 1 vs. Scenario 2 for rejuvenation scheduling in a cloud-based system with multiple HSSs subject to the software-aging phenomenon.

Table 1 shows the reliability analysis results for the application server subsystem in both of the 1-HSS and 2-HSSs cases. It is easy to see that the 2-HSSs case is more reliable than the 1-HSS case since the system design employs two HSSs for each primary one, and thus it is more fault-tolerant.

Table 1. Application sever reliability with 1-HSS and 2-HSSs

Time (days)	1-HSS App. Server R(t)	2-HSS App. Server R(t)
0	1	1
1	0.999998667	0.99999999879
5	0.999771280	0.999999605
10	0.998960870	0.999995272
20	0.998303980	0.999944405
30	0.995622592	0.999768668
60	0.978742690	0.997495766
90	0.948655000	0.990481970
120	0.906980000	0.976492615
180	0.798764000	0.923257827
240	0.673967900	0.837729710
300	0.548546400	0.729271367
365	0.423781190	0.600339000

Similarly, Table 2 shows the reliability analysis results for the database server subsystem in both of the 1-HSS and 2-HSSs cases. Again, the 2-HSSs case is more reliable than the 1-HSS case since the system design employs two HSSs for each primary one, and thus it is more fault-tolerant.

Table 2. Database server reliability with 1-HSS and 2-HSSs

Time (days)	1-HSS DB Server R(t)	2-HSSs DB Server R(t)
0	1	1
1	0.999993319	0.9999999862
5	0.999937115	0.9999972565
10	0.999671339	0.9999735858
20	0.995347000	0.9997521730
30	0.988965000	0.9991023013
60	0.953890079	0.992467550
90	0.898541000	0.975684900
120	0.828575990	0.946938900
180	0.665978740	0.856144800
240	0.499787000	0.735050867
300	0.349438900	0.603242334
365	0.213935800	0.466220270

Table 3 shows how the system reliability evolves in rejuvenation cycles and phases with duration of 125 days for the 1-HSS case in Scenario 2. The rows highlighted in blue in Table 3 indicate that system reliability has reached the reliability threshold, and threfore a software rejuvenation occurs as scheduled. Comparatively, we illustrate the system reliability with the 2-HSSs case in Scenario 2, as shown in Table 4, for the similar time span.

Figure 6 illustrates in details the differences between the two cases, 1-HSS vs. 2-HSSs, based on Scenario 1 for system-specific rejuvenation. From Tables 3 and 4, we can see that the system reliability reaches the threshold after 25 days and 59 days for the 1-HSS and 2-HSSs cases, respectively. According to Scenario 1, the whole system is restarted when the threshold is reached, and the system returns to its initial state. As a result, the rejuvenization must be repeated regularly every 25 and 59 days for the 1-HSS and 2-HSSs cases, respectively. Such rejuvenization strategies are reflected in Fig. 6 as recurrent rejuvenation schedules for the two cases in Scenario 1.

On the other hand, both Tables 3 and 4 show irregular ocurrences of rejuvenation in Scenario 2. This is because in Scenario 2, we rejuvenate the component that has the lowest reliability when the system reliability reaches threshold 0.99.

Table 3. System reliability with rejuvenation (1-HSS Scenario 2)

Phase	Time (days)	System Reliability 1-HSS (Scenario 2)
1	0	1
	1	0.999991986
	5	0.999708409
	10	0.998632551
	20	0.993658872
	25	0.99
2	25.003472	0.99742372
	25.006944	0.997421
	25.01389	0.997419
	25.020833	0.9974178
1	26	0.99685634
	30	0.995559949
	35	0.993418915
	40	0.99
2	40.003472	0.9974958
	40.006944	0.997495500
	40.01389	0.997495200
	40.020833	0.9974948
1	41	0.997120047
	45	0.994606441
	52	0.99
2	52.003472	0.99949305
	52.006944	0.999493030
	52.01389	0.999493000
	52.020833	0.999492970
1	55	0.999065
	60	0.997667
	70	0.9919284
	72	0.99
2	72.003472	0.995344
	72.006944	0.995343
	72.01389	0.995341
	72.020833	0.995339
1	75	0.999065
	80	0.99
2	80.003472	0.999806827
	80.006944	0.999806825
	80.01389	0.99980623
	80.020833	0.999806821
1	81	0.998490189
	85	0.997335279
	95	0.995147035
	101	0.99
2	101.003472	0.995954210
	101.006944	0.995954207
	101.01389	0.995954205
	101.020833	0.995954203
1	102	0.9956159
	105	0.99446611
	110	0.991807548
	112	0.99
2	112.003472	0.99872147
	112.006944	0.99872146
	112.01389	0.99872144
	112.020833	0.99872142
1	115	0.9956159
	120	0.99446611
	125	0.991894655

Figure 7 shows the differences between the two cases, 1-HSS vs. 2-HSSs, based on Scenario 2 for component-specific rejuvenation. According to the figure, when the reliability threshold is reached, the component with the lowest reliability, e.g., the database server, is rejuvenated first.

It is worth mentioning that in Scenario 2 with 1-HSS, the database server gets rejuvenated for two consecutive times on day 80 and day 101, as shown in Table 3. We can see how this irregularity affects the reliability pattern in Fig. 7.

Table 4. System reliability with rejuvenation (2-HSSs Scenario 2)

Phase	Time (days)	System Reliability 2-HSSs (Scenario 2)
1	0	1
	1	0.999999985
	5	0.999996861
	10	0.999968858
	30	0.998871177
	59	0.990485251
2	59.003472	0.997632898
	59.006944	0.997632430925
	59.01389	0.997631493009
	59.020833	0.997630555246
1	64	0.99688844
	69	0.995980045
	80	0.9930624
	90	0.99
2	90.003472	0.99946301
	90.006944	0.9994629
	90.01389	0.9994628
	90.020833	0.9994626
1	95	0.998411679000
	100	0.997619850000
	120	0.991000000000
	123	0.990560000000
2	123.003472	0.999677418000
	123.006944	0.999677416000
	123.01389	0.999677415000
	123.020833	0.999677412000
1	128	0.999470858000

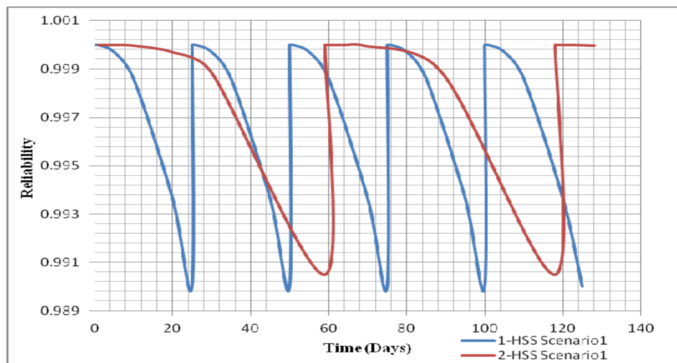

Fig. 6. Rejuvenation scheduling: 2-HSSs vs. 1-HSS (Scenario1)

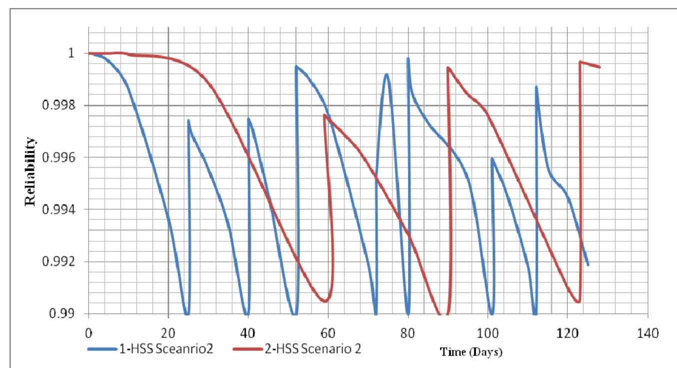

Fig. 7. Rejuvenation scheduling: 2-HSSs vs. 1-HSS (Scenario 2)

Figure 7 also shows that 3 rejuvenations are needed for Scenario 2 with 2-HSSs vs. 7 rejuvenations needed for the 1-HSS case during 125 days. Therefore, compared with Scenario 2 with 1-HSS, using Scenario 2 with 2-HSSs results in (7–3)/(7) = 57% reduction in cost and management for software rejuvenation while keeping the system reliability well above the 0.99 threshold. This result was as expected because using 2-HSSs for each primary one surely makes the whole system more reliable and dependable.

IV. Conclusions and Future Work

In this paper, we perform reliability analysis for cloud-based systems with software spares subject to non-constant failure rates. The proposed work is based on an analytical approach for scheduling a preventive maintenance procedure, called software rejuveantion. We adopted an extension of DFT, called SSP gate, to model and evaluate the reliability of a cloud-based system with multiple hot software spares. We used the Weibull distribution to emulate an increasing failure rate due to the software-aging phenomenon. The case study showed that our approach was feasible and could produce useful preventive maintance schudules.

For future work, in order to forecast increasing failure rates for software components, we will develop an e-commerce application, deploy it on reputable cloud-based platfroms, such as Amazon Web Service AWS, Window Azure, and Google App Engine, and collect empirical data related to resource degradation. Data fitting technique will be used to derive the most suitable probability density function for the system time-to-failure. Stochastic partial differential equations may be considered and applied to this field of study to help predict how software aging affects the failure rate. As such, more accurate results for system reliability can be used to derive preventive maintencance schedules for cloud-based systems.

References

[1] M. Rausand and A. Høyland, *System Reliability Theory: Models, Statistical Methods, and Applications*, Second Edition, Hoboken, New Jersey, USA, John Wiley & Sons, Inc., 2004.

[2] M. Grotte, A. Nikoran, and K. S. Trivedi, "An empirical investigation of fault types in space mission system software," in *Proc. of the International Conference on Dependable Systems & Networks (DSN 2010)*, June 28-July 1, 2019, Chicago, IL, USA, pp. 447-456.

[3] M. Grotte, R. Matias, and K. S. Trivedi, "The fundamentals of software aging," in *Proc. of the First International Workshop on Software Aging and Rejuvenation (WoSAR)*, in conjounction with the 19th IEEE International Symposium on Software Reliability Engineering (ISSRE), Seattle, WA, USA, November 11-14, 2008, pp. 1-6.

[4] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton, "Software rejuvenation: analysis, module and applications," in *Proc. of the Twenty-Fifth International Symposium on Fault-Tolerant Computing (FTCS '95),* Pasadena, CA, USA, June 27-30, 1995, pp. 381-390.

[5] M. Lyu, "Software reliability engineering: a roadmap," *in Proc. of the 29th International Conference on Software Engineering, Future of Software Engineering*, Minneapolis, USA, 2010, pp. 153-170.

[6] J. Rahme and H. Xu, "A software reliability model for cloud-based software rejuvenation using dynamic fault trees," *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, Vol. 25, Nos. 9 & 10, 2015, pp. 1491-1513.

[7] H. Boudali, P. Crouzen and M. Stoelinga, "Dynamic fault tree analysis using input/output interactive markov chains," in *Proc of the 37th International Conference on Dependable Systems and Networks (DSN),* Edeinburgh, UK , June 25-28, 2007, pp. 708-717.

[8] R. B. Abernethy, *The New Weibull Handbook*, 2nd Edition, Abernethy, North Palm Beach, FL, USA, 1996.

[9] J. Rahme and H. Xu, "Dependable and Reliable Cloud-Based Systems Using Multiple Software Spare Components," To appear in *Proc. of the International Conference on Advanced and Trusted Computing (ATC-17),* San Francisco Bay Area, CA, USA, Aug. 4-8, 2017.