Reliability-Based Software Rejuvenation Scheduling for Cloud-Based Systems

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

Abstract—**The reliability and availability of a cloud-based system play an important role in evaluating its system performance. Due to the promised high reliability of physical facilities provided for cloud services, software faults have become a major factor for failures of cloud-based systems. In this paper, we focus on the software aging phenomenon where system performance may be progressively degraded due to exhaustion of system resources, fragmentation and accumulation of errors. We present a proactive technique, called software rejuvenation, to counteract the software aging problem. The dynamic fault tree (DFT) formalism is adopted to model the system reliability before and during a software rejuvenation process in an aging cloud-based system. Then it is converted into Markov Chains to derive the system reliability function. We use a case study of a cloud-based system to illustrate the validity of our approach. Based on the reliability analysis results, we show how to estimate software rejuvenation schedules that can keep the system reliability above a predefined critical level for required system availability.**

Keywords-Software aging; software rejuvenation; reliability analysis; dynamic fault tree (DFT); Markov chain; scheduling.

I. INTRODUCTION

With the promised high reliability and availability of physical facilities, including the hardware facilities and their associated redundancy mechanisms, provided by cloud service providers, software faults have now become a major factor of cloud-based system failures. Since software reliability is considered one of the weakest points in system reliability, software fault tolerance and failure forecasting require more attentions than hardware fault tolerance in modern computer-based systems [1][2]. This work is motivated to deal with the software faults in cloud computing in order to assure high reliability and availability of cloud-based software systems. Reliability and availability are two common ways to express system fault tolerance in industry. A reliable computer-based system typically has high availability if unreliability is the major cause for unavailability. In this paper, we focus on analyzing the reliability of cloud-based systems for software fault tolerance in software reliability engineering (SRE). Traditional SRE has been based on analysis of software defects and bugs such as Bohrbugs or Heisenbugs without considering software aging related bugs [1]. The concept of software aging phenomenon was introduced in the middle 90s, which explains that the system resources used by the software degrade gradually in function of time [3][4]. Software aging starts to show up due to multiple factors such as memory bloating, memory leaks, unterminated threads, data corruption, unreleased file-locks, storage space and fragmentation, and accumulation of round-off errors when running a software. Software aging has considerably changed the SRE field of study, and become a major factor for the reliability of fully tested and deployed software systems. To deal with software aging and to assure software fault tolerance, software rejuvenation process has been introduced as a proactive approach to counteracting software aging and maintaining a reliable software system [3]. Software rejuvenation involves actions such as stopping the running software occasionally, cleaning its internal state (e.g., garbage collection, flushing operating system kernel tables, and reinitializing internal data structures). The simplest way to perform software rejuvenation is to restart the application that causes the aging problem, or to reboot the whole system.

Due to the ever-growing cloud computing technology and its vast markets, the workload of a cloud-based system has increased dramatically. A heavy workload of cloud-based system will inevitably lead to more software aging problems. In this paper, we introduce an approach to developing rejuvenation schedules for cloud-based systems in order to maintain their high system reliability. In our approach, we adopt an analytical-based approach to compute the reliability of a cloud-based system using Dynamic Fault Tree (DFT). To maintain high system reliability and ensure a zero-downtime rejuvenation process, we introduce cloud-based spare parts as major software components. Once the DFT model is developed, it is converted into Continuous Time Markov Chains (CTMC) to calculate the system reliability. We assume a practical reliability threshold for the core software components of the system. When the threshold is reached, the software rejuvenation process is triggered, and the reliability of the cloud-based system is boosted to its initial state. Our case study shows that software rejuvenation scheduling based on the reliability analysis of a cloud-based system can significantly enhance its system reliability and availability.

Previous studies on software aging and software rejuvenation for predicting a rejuvenation schedule can be classified into two categories, namely analytical-based and measurement-based approaches [5]. In an analytical-based approach, a failure distribution is assumed for software faults related to the software aging phenomenon, and software rejuvenation is executed at a fixed interval based on the analytical results of the system reliability and availability [6].

Several analytic models have been proposed to determine the optimal time for rejuvenation. Bobbio *et al.* proposed a fine-grained software degradation model for optimal rejuvenation policies [7]. Based on the assumption that the current degradation level of the system can be identified, they presented two different strategies to determine whether and when to rejuvenate. Vaidyanathan *et al.* presented an analytical model of a software system using inspection-based software rejuvenation [8]. In their approach, they showed that inspection-based maintenance was advantageous in many cases over non-inspection based maintenance. Although the above approaches proposed various models for software rejuvenation, they are not intended to address complex system components' behaviors and interactions, such as dynamic relationships between software components including sparing relationship and functional dependency. Different from the existing analytical-based approaches, we focus on the dynamic behaviors of software components in the context of cloud-based systems. We use sparing relationships as an example to show how dynamic relationships of software components in a cloud-based system can be modeled using DFT.

On the other hand, measurement-based approach applies statistical analysis to the measured data of resources usage and degradation that may lead to the software aging problem. In a measurement-based approach, a monitoring program is used to continuously collect the system performance data, which are analyzed to estimate the system degradation level. When exhaustion reaches a critical level, the software rejuvenation process is triggered. Machida *et al.* used Mann-Kendall test to detect software aging from traces of computer system metrics [9]. They tested for existence of monotonic trends in time series, which are often considered indication of software aging. Grottke *et al.* studied the resource usage in a web server subject to an artificial workload [10]. They applied non-parametric statistical methods to detect and estimate trends in the data sets for predicting future resource usage and software aging issues. The existing measurement-based approaches are feasible ways to detect software aging problems in real-world computer-based systems; however, they typically involve processing of large amount of system data. Thus, they are not as efficient as analytical-based approaches. On the other hand, measurement-based approaches do provide useful insights about the system behaviors and failure distributions related to software aging. As such, our research is complementary to research efforts that investigate the relationships of static features of software and metrics for software faults with the software aging phenomenon using statistical analysis.

II. REJUVENATION OF CLOUD-BASED COMPONENTS

In a cloud-based system, virtualization allows one to share a machine's physical resources among multiple virtual environments, called virtual machines (VM). As shown in Fig. 1, a VM is not bounded to the hardware directly; rather it is bounded to generic drivers that are created by a virtual machine manger (VMM) or a hypervisor. Since a VM can be easily created and destroyed, it is particularly useful in a disaster recovery process of a cloud-based system. In this paper, we refer a cloud-based system as a software system that consists of multiple VMs, where each VM is considered a software component of the cloud-based system.
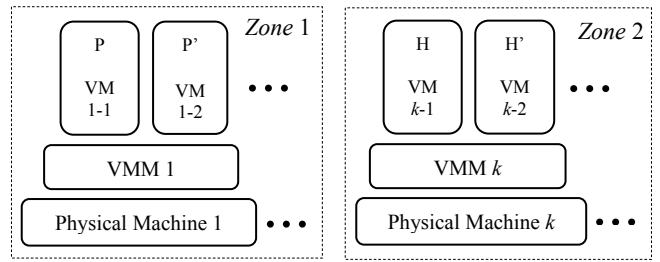

Figure 1. An example of reliable cloud-based systems

As a proactive fault management technique, software rejuvenation has been used to refresh system internal states and prevent the occurrence of software failures due to software aging. As we have mentioned, a simple way of software rejuvenation in a cloud-based system is system reboot, e.g., to restart a VM or all VMs of the system. The basic idea of our approach is to create a new instance of VM that replaces the one to be rejuvenated. Since the newly deployed VM instance has not yet been affected by the software aging phenomenon, the reliability of the software component is boosted back to its initial condition. To achieve high fault tolerance and reliability, we further adopt the software redundancy technique using two different types of software standby spares, namely Cold Spare Part (CSP) and Hot Spare Part (HSP). In the context of cloud-based systems, cold standby means that the software component is available as an image of a VM, rather than an active VM instance. Data between primary component and the spare one is regularly mirrored based on a specified schedule, e.g., multiple times a day. Since a CSP is not up running continuously and does not take any workload, its reliability approaches to 1 with a failure rate 0. Since a CSP can be started very quickly, the recovery time using CSP typically takes just a few minutes to no more than two hours. Note that a software-defined CSP is different from a hardware-based CSP in terms of its cost and efficiency. The cost of a software-defined CSP is its storage and very little CPU time; while a hardware-based CSP is a physical device that must be available all the time in order to assure fast failover [1]. Furthermore, a software-defined CSP can be started very quickly, but a hardware-based CSP typically requires manual configuration and adjustment in the event of partial or total failure.

Similarly, an HSP in the context of cloud-based systems is a hot standby VM instance. This means the software component serving as an HSP must be installed and deployed, and it must be instantly available in a case that the primary component fails. Although an HSP is deployed and running along with the primary component, it typically does not take any workload for processing user requests. To ensure fault tolerance, critical data is mirrored in near real time from the primary VM instance. This generally provides a recovery time of a few seconds in case of a failure. In our system design, each critical primary component is equipped with at least one HSP and one CSP in order to maintain the needed reliability. When calculating the system reliability, we only need to consider the primary component and its HSP; while the failure rate of a CSP is constantly 0. In the following, for simplicity, we denote a primary VM instance/component as P, which is active and processing workload, an HSP as H, which is active

but does not take any workload, and a CSP as *C*, which is inactive and also does not take any workload.

In our approach, a rejuvenation scheduling is based on the results of reliability modeling and analysis of a cloud-based system. When the reliability of a system component or the whole system reaches a predefined threshold, the rejuvenation process is triggered. We assume the rejuvenation process takes about 30 minutes, which is typically sufficient for starting a CSP and transfer all requests to the new VM. As a simple example illustrated in Fig. 1, suppose we have two instances, i.e., a primary component *P* and a hot standby one *H*, which are deployed on two different physical machines. The two physical machines usually belong to two different zones (denoted as Zone 1 and Zone 2 in Fig. 1), so a power/network outage in one zone, will not affect the availability of the other one. To rejuvenate the whole system, we need to start two CSPs *P'* and *H'* to replace *P* and *H*, respectively. As shown in Fig. 1, *P'* and *H'* are deployed on the same physical machine where *P* and *H* are deployed, respectively, but in reality, both *P'* and *H'* can be deployed on any physical servers.

Once the spare components *P'* and *H'* are up and running, *P'* will start processing new system requests, while *H'* is kept alive and will not take any workload. Meanwhile, we allow 30 minutes for the old components *P* and *H* to finish processing their existing requests. After 30 minutes, we shut down and delete the components *P* and *H*, which has been successfully replaced by *P'* and *H'* after completion of the rejuvenation process. Note that we do not try to restart and reuse the same instances *P* and *H* in our approach. This is because different from a physical machine, a VM can be easily created and deployed, thus deploying new instances *P'* and *H'* is a much more efficient way than restarting *P* and *H*.

During the rejuvenation procedure, we consider two scenarios. One scenario is to rejuvenate the major software components all together. In this case, we replicate the whole system at the same time when the system reliability reaches its threshold. We call this scenario as a system-specific rejuvenation. The second scenario is a component-specific one, meaning that each time, we only rejuvenate the critical component whose reliability is the lowest one when the system reliability reaches its reliability threshold. As we will show in a case study, the component-specific rejuvenation usually demonstrates certain advantages over the system-specific approach.

III. MODELING AND ANALYSIS USING DFT

In this section, we first briefly introduce DFT, then we show how to use DFT to model and analyze the reliability of a cloud-based system subject to software rejuvenation. To simplify matters, we assume that the time-to-failure for the software components (i.e., the VMs) has a probability density function that is exponentially distributed. Therefore, all VMs have constant failure rates.

A. Dynamic Fault Tree

The fault tree modeling technique was introduced in 1962 at Bell Telephone Lab, which provides a conceptual modeling approach to representing system level reliability in terms of interactions between component reliabilities [1]. Fault tree analysis (FTA) is by far the most commonly used technique for risk and reliability analysis, where the system failure is described in terms of the failure of its components. Standard fault trees are combinatorial models and are built using static gates (e.g., AND-gate, OR-gate, and K/M-gate) and basic events. As combinatorial models can only capture the combination of events without considering the order of occurrence of their failures, they are usually inadequate to model today's complex dynamic systems.

DFT augments the standard combinatorial gates of a regular fault tree, and introduces three novel modeling capabilities, namely spare component management and allocation, functional dependency, and failure sequence dependency. These modeling capabilities are realized using three main dynamic gates: the spare gate, the functional dependency gate, and the priority-AND gate. The work done in this paper uses the dynamic spare gate, in particular the hot spare gate or HSP gate. Note that a spare gate has one primary input and one or more alternate inputs (i.e., the spares). The primary input is initially powered on, and when it fails, it is replaced by an alternate input. The spare gate fails when the primary and all the alternate inputs fail.

Since a DFT failure model is typically used to describe dynamic relationships rather than simple combinatorial ones, we need to transform it into a state-based formalism, such as Markov chains, for formal analysis. In the following section, we show how to convert a DFT model into Markov chains.

B. Modeling and Analysis Using DFT

To model and analyze the reliability of a cloud-based system with spare parts, we consider two different phases. **Phase 1** represents the pre-rejuvenation phase, where the reliability analysis is based on the failure rates of the primary components and their HSPs. CSPs are not considered for reliability analysis, as they cannot take over the system load instantly when both the primary and hot spare components fail. We model the system reliability using DFT, and then the DFT is converted into a CTMC to derive the system reliability function.

Figure 2 shows a simple hot spare gate with one primary component denoted as *P* and one hot spare part denoted as *H*. At the right-hand side of the figure, we show the CTMC model corresponding to the HSP gate. There are four states 1 to 4 defined in the CTMC model, which are denoted as *PH*, *P*, *H**, and *FAILURE*, respectively. The state *PH* (State 1) refers to the one in which both the primary component and the hot spare part are functioning. When the hot spare part component or the primary component fails, the model enters its *P* state (State 2) or *H** state (State 3), respectively.
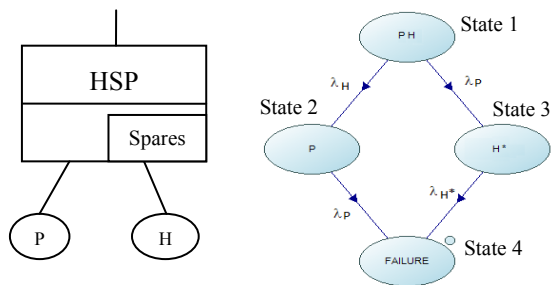

Figure 2.   An HSP gate and its corresponding CTMC model

Note that we denote State 3 as *H** instead of *H* because in State 3, the hot spare part has a different failure rate as the one in State 1. The reason why *H* and *H** have different failure rates is described as follows. In State 1, the hot spare part does not take any workload, therefore its failure rate λ_H is fairly low; however, in State 3, the hot spare part takes the normal workload as the primary one before it fails, its failure rate becomes higher due to the software aging phenomenon. Suppose the hot spare part has the same configuration as the primary one, then in State 3, its failure rate shall equal to the primary component's failure rate λ_P.

Let $P_i(t)$ be the probability of the system in state *i* at time *t*, where $1 \le i \le 4$, and $P_{ij}(dt) = P[X(t+dt) = j \mid X(t) = i]$ be the incremental transition probability with random variable $X(t)$. The following matrix $[P_{ij}(dt)]$, where $1 \le i, j \le 4$, is the incremental one-step transition matrix [1] of the CTMC defined in Fig. 2.

$$[P_{ij}(dt)] = \begin{bmatrix} 1-(\lambda_P+\lambda_H)dt & \lambda_H dt & \lambda_P dt & 0 \\ 0 & 1-\lambda_P dt & 0 & \lambda_P dt \\ 0 & 0 & 1-\lambda_{H*} dt & \lambda_{H*} dt \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1)$$

The matrix $[P_{ij}(dt)]$, where $1 \le i, j \le 4$, is a stochastic matrix with each row sums to 1. This matrix provides the probabilities for each state either remaining (when $i = j$) or transit to a different state (when $i \ne j$) during the time interval *dt*. Given the initial probabilities of the states, the matrix can be used to describe the state transition process completely. From the matrix defined in Eq. (1), we can derive the following relations as in Eqs. (2-4).

$$P_1(t+dt) = (1-(\lambda_p+\lambda_H)dt)P_1(t) \quad (2)$$

$$P_2(t+dt) = (\lambda_H dt)P_1(t) + (1-\lambda_p dt)P_2(t) \quad (3)$$

$$P_3(t+dt) = (\lambda_P dt)P_1(t) + (1-\lambda_{H*} dt)P_3(t) \quad (4)$$

where the initial probabilities is defined as the probability of the system being at State 1; thus, we have $P_1(0) = 1$, and $P_2(0) = P_3(0) = 0$. By applying *dt* limit to 0, we get a set of linear first-order differential equations as in Eqs. (5-7), which are state equations for states 1-3.

$$P_1'(t) = \lim_{dt \to 0} \frac{P_1(t+dt)-P_1(t)}{dt} = -(\lambda_P+\lambda_H)P_1(t) \quad (5)$$

$$P_2'(t) = \lim_{dt \to 0} \frac{P_2(t+dt)-P_2(t)}{dt} = \lambda_H P_1(t) - \lambda_P P_2(t) \quad (6)$$

$$P_3'(t) = \lim_{dt \to 0} \frac{P_3(t+dt)-P_3(t)}{dt} = \lambda_P P_1(t) - \lambda_{H*} P_3(t) \quad (7)$$

The state equations defined in Eqs. (5-7) can be solved using Laplace transformation, which allows transforming a linear first order differential equation into a linear algebraic equation that is easy to solve.

Let the Laplace transformation of $P_i(t)$ be $F_i(s)$, where $1 \le i \le 3$, we can solve the original linear first order differential equations in Eqs. (5-7) as follows.

$$F_1(s) = \frac{1}{(s+\lambda_P+\lambda_H)} \Rightarrow P_1(t) = e^{-(\lambda_P+\lambda_H)t}$$

$$F_2(s) = \frac{\lambda_H}{(s+\lambda_P+\lambda_H)(s+\lambda_P)} \Rightarrow P_2(t) = e^{-\lambda_P t} - e^{-(\lambda_P+\lambda_H)t}$$

$$F_3(s) = \frac{\lambda_P}{(s+\lambda_P+\lambda_H)(s+\lambda_{H*})} \Rightarrow P_3(t) = \frac{\lambda_P}{\lambda_H}(e^{-(\lambda_{H*})t} - e^{-(\lambda_P+\lambda_H)t})$$

The reliability function $R(t)$ is the summation of $P_1(t)$, $P_2(t)$ and $P_3(t)$, which can be calculated as in Eq. (8), assuming $\lambda_{H*} = \lambda_P$, i.e., *H* has the same configuration as the primary one.

$$R(t) = P_1(t) + P_2(t) + P_3(t) = (1+\tfrac{\lambda_P}{\lambda_H})e^{-\lambda_P t} - (\tfrac{\lambda_P}{\lambda_H})e^{-(\lambda_P+\lambda_H)t} \quad (8)$$

Note that $P_4(t)$ is the probability of system's being in the *FAILURE* state at time *t*. Therefore, the system unreliability function $U(t) = P_4(t) = 1 - R(t)$.

Phase 2 is the software rejuvenation phase. When the predefined reliability threshold is reached, the software rejuvenation process is initiated, and the system enters the software rejuvenation phase. As we have mentioned, there are two rejuvenation scenarios, namely the system-specific rejuvenation and the component-specific rejuvenation. To illustrate the basic idea of calculating reliability in this phase, we use the first scenario. In this scenario, we start two CSPs *P'* and *H'* to replace *P* and *H*, respectively. During the rejuvenation period, the four software components *P*, *H*, *P'* and *H'* coexist. As shown in Fig. 3, we decompose the dynamic fault tree model into two sub-trees, *S*1 and *S*2, which are connected by an AND-gate. Subtree *S*1 consists of the components *P* and *H* that are to be rejuvenated, while subtree *S*2 consists of the newly deployed components *P'* and *H'*, which are used to replace *P* and *H*. Both of the subtrees are defined as HSP gates, each of which can be computed using the same analysis technique as described in Phase 1. Since both of the two HSP gates are functioning at the same time, any of them fails during the rejuvenation phase will not lead to the failure of the whole system, and the system fails only when both of the two HSP gates fail. Therefore, the two HSP gates shall be connected by an AND-gate.
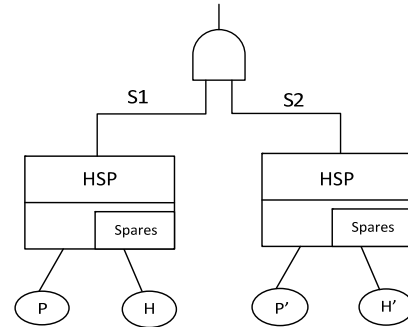

Figure 3.   A DFT model with 2 HSP gates (Phase 2)

Once we have solutions to *S*1 and *S*2, the static component, i.e., the AND-gate can be easily solved using the sum-of-disjoint-products (SDP) method [1]. Specifically, to calculate the reliability of the whole system in this phase, we first calculate the unreliability functions $U_{S1}(t)$ and $U_{S2}(t)$ for *S*1 and *S*2, respectively. Then we calculate the reliability of the AND-gate as in Eq. (9).

$$R(t) = 1 - U_{AND}(t) = 1 - U_{S1}(t) * U_{S2}(t) \quad (9)$$

In the following section, we describe a case study considering both of the two scenarios during the rejuvenation process. Scenario 1 involves rejuvenation of the whole system by replicating all major software components when system reliability reaches the threshold; while in Scenario 2, we rejuvenate the most critical component with the lowest reliability when the system reliability reaches its threshold.

IV. CASE STUDY

A typical cloud-based system is illustrated in Fig. 4, which consists of an application server *PA* and a database server *PB*, all deployed on VMs. To enhance the system reliability, two hot spare components *HA* and *HB* are set up for *PA* and *PB*, respectively, which are ready to take over the workload once the primary ones fail. Note that all servers are deployed on VMs in different zones for fault-tolerance purpose. As a clarification for the reliability analysis in this case study, we view a VM with its OS, the server and the deployed services as a single software component. In addition, we only consider the reliability of the servers within the box with dashed lines, and assume the proxy server's reliability is ideal. Furthermore, we assume that the proxy server and the application server can monitor and detect failures of the application server and the database server, respectively.

To ensure a high reliability of the system, we set a reliability threshold of 0.99. For this case study, we assume the typical failure rates for the servers, where $\lambda_{PA} = 0.004$, $\lambda_{HA} = 0.0025$, $\lambda_{PB} = 0.005$, $\lambda_{HB} = 0.003$. Note that the failure rates of the hot spare parts are lower than their corresponding primary ones because the spare parts do not take any workload when the primary ones are functioning. However, when the primary servers fail, the failure rates of the hot spare parts will be increased, i.e., $\lambda_{HA}* = \lambda_{PA} = 0.004$, $\lambda_{HB}* = \lambda_{PB} = 0.005$. This case study involves 8 software components split into two groups. The first group consists of the four servers shown in Fig. 4. The second group consists of four CSP components that are used to replace the servers in the first group during the rejuvenation process. We name the severs in the second group as *PA'*, *HA'*, *PB'*, and *HB'*. As the CSP components are undeployed VM images, their failure rates are 0. Once deployed, they will have the same failure rates as their corresponding software components.
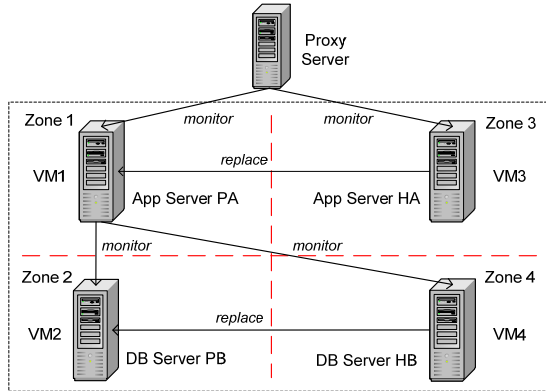

Figure 4. A cloud-based system with servers deployed on VMs

Figure 5 shows the DFT model of the cloud-based system in Phase 1. As the system fails when either the application servers fail or the database servers fail, the two HSP gates are connected by an OR-gate, which can solved as in Eq. (10).

$$R(t) = 1 - U_{OR}(t) = 1 - (U_{S1}(t) + (1 - U_{S1}(t)) * U_{S2}(t)) \qquad (10)$$

where $U_{OR}(t)$, $U_{S1}(t)$ and $U_{S2}(t)$ are the unreliability functions of the OR-gate, the subtrees $S1$ and $S2$, respectively. According to Eq. (8), $U_{S1}(t)$ and $U_{S2}(t)$ can be calculated as in Eq. (11) and Eq. (12), respectively.
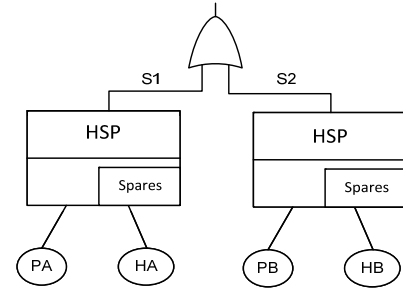

Figure 5. DFT model of the cloud-based system (Phase 1)

$$U_{S1}(t) = 1 - R_{S1}(t) = 1 - (1 + \tfrac{\lambda_{PA}}{\lambda_{HA}})e^{-\lambda_{PA}t} + (\tfrac{\lambda_{PA}}{\lambda_{HA}})e^{-(\lambda_{PA}+\lambda_{HA})t} \qquad (11)$$

$$U_{S2}(t) = 1 - R_{S2}(t) = 1 - (1 + \tfrac{\lambda_{PB}}{\lambda_{HB}})e^{-\lambda_{PB}t} + (\tfrac{\lambda_{PB}}{\lambda_{HB}})e^{-(\lambda_{PB}+\lambda_{HB})t} \qquad (12)$$

In Phase 2, we consider both of the scenarios mentioned in the end of Section III.B, so their impacts on system reliability as well as their consequent rejuvenation schedules can be compared. Figure 6 shows the DFT model of the cloud-based system in Phase 2 based on Scenario 1. For the same reason as in Phase 1, the system reliability can be calculated as in Eq. (13). According to Eq. (9), $U_{S3}(t)$ and $U_{S4}(t)$ can be calculated as in Eq. (14) and Eq. (15), respectively.

$$R(t) = 1 - U_{OR}(t) = 1 - (U_{S3}(t) + (1 - U_{S3}(t)) * U_{S4}(t)) \qquad (13)$$

$$U_{S3}(t) = U_{S1}(t) * U_{S1'}(t) \qquad (14)$$

$$U_{S4}(t) = U_{S2}(t) * U_{S2'}(t) \qquad (15)$$

Note that in Eqs. (14-15), $U_{S1}(t)$, $U_{S1'}(t)$, $U_{S2}(t)$ and $U_{S2'}(t)$ can be calculated in a similar way as in Eqs. (11-12).
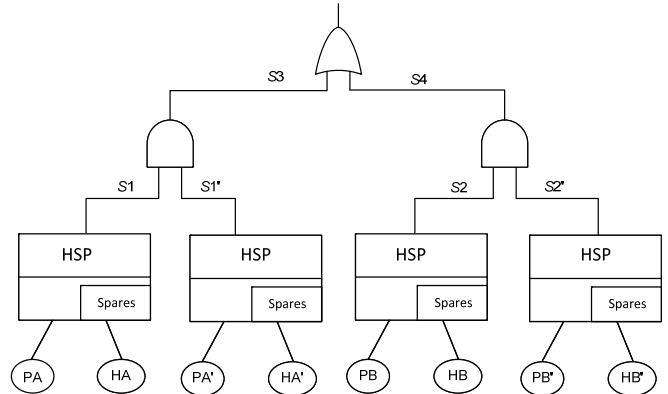

Figure 6. DFT model of the cloud-based system in Phase 2 (Scenario 1)

The reliability analysis results for Scenario 1 are listed in Table 1. The table shows that the reliability threshold (0.99) is reached every 18 days according to the reliability analysis results. Hence, both application and database servers are rejuvenated at the end of Phase 1. Phase 2 has a 30-minute time duration; therefore, we calculate the system reliability at 5, 10, 20 and 30 minutes in Phase 2 to illustrate how system reliability may change during the rejuvenation process. From the table, we can see that the system reliability is kept very high during the transition. After 30 minutes, the newly deployed servers completely take over the system, and the servers to be rejuvenated are shut down. When this happens, the system returns to its initial state, and starts a new life cycle

with a very high initial reliability. Therefore, Table 1 suggests that the system should be rejuvenated every 18 days in order to keep the system reliability above the threshold.

Table 1. System reliability with software rejuvenation (Scenario 1)

Phase	Time (Days)	App Servers Reliability	DB Servers Reliability	System Reliability
	0	1	1	1
1	1	0.99998705	0.9999801	0.9999671502577
	5	0.9996806	0.9995107	0.9991914562824
	10	0.998745	0.998085	0.9968324033250
	18	0.996044	0.994004	0.9900717201760
2	18.0035	0.999999999999	0.999999999999	0.9999999999979
	18.0069	0.999999999997	0.999999999994	0.9999999999917
	18.0139	0.999999999990	0.999999999977	0.9999999999669
	18.0208	0.999999999978	0.999999999940	0.9999999999177
1	19	0.99998705	0.9999801	0.9999671502577
	23	0.9996806	0.9995107	0.9991914562824
	28	0.998745	0.998085	0.9968324033250
	36	0.996044	0.994004	0.9900717201760
2	36.0035	0.999999999999	0.999999999999	0.9999999999979
	36.0069	0.999999999997	0.999999999994	0.9999999999917
	36.0139	0.999999999990	0.999999999977	0.9999999999669
	36.0208	0.999999999978	0.999999999940	0.9999999999177
...	...	...	...	...

By further looking into Table 1, we can see that when the system reliability reaches 0.99 after 18 days, the reliability of the database server subsystem is lower than that of the application server subsystem. This suggests that we may rejuvenate the most critical components (i.e., the component with the lowest reliability) first. Now suppose we choose to rejuvenate the database servers first. Then we wait until the system reliability reaches the threshold again, and rejuvenate the application servers next, as they now become the most critical components. This is exactly what happens for the rejuvenation scheduling in Scenario 2, where the application servers and the database servers are rejuvenated alternatively. The system reliability in Scenario 2 can be calculated in a similar way as in Scenario 1.

We now illustrate the rejuvenation scheduling for both Scenario 1 and Scenario 2 as in Fig. 7. In the figure, the start of rejuvenation is indicated by a sudden increment of the system reliability.
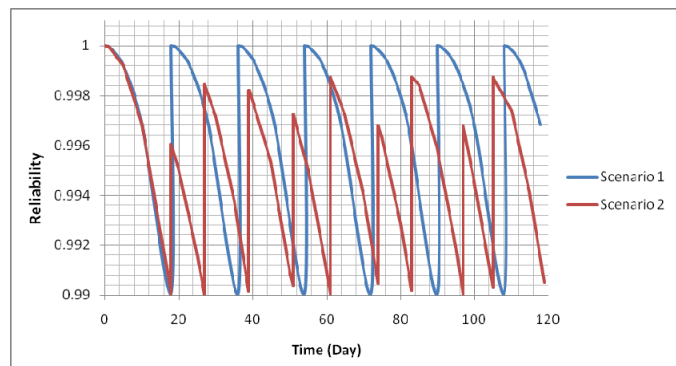

Figure 7. Rejuvenation scheduling (Scenario 1 vs. Scenario 2)

By comparing the two rejuvenation schedules, we can see that during 119 days, Scenario 1 has 6 rejuvenation process which requires rejuvenating both of the application and database servers. On the other hand, Scenario 2 has 9 rejuvenation process which only requires rejuvenating either the application servers or the database servers. It is easy to see that Scenario 2 requires less management of the servers in order to keep the system reliability above the 0.99 threshold all the time. Suppose the rejuvenation of the application servers has the same cost as that of the database servers, by using the rejuvenation scheduling defined in Scenario 2, the cost can be reduced by $(6*2 - 9)/(6*2) = 25\%$ comparing to the rejuvenation scheduling defined in Scenario 1.

V. CONCLUSIONS AND FUTURE WORK

In this paper, we propose a reliability-based approach to estimating a rejuvenation scheduling for cloud-based systems. The system requires using hot spare parts during normal running time, and cold spare parts during the rejuvenation process in order to keep the system reliability above a predefined threshold. By modeling the reliability of a cloud-based system using DFT, we are able to derive the reliability function for each software component as well as the whole system. We define two phases for the software rejuvenation, and discuss about two scenarios of the rejuvenation process in Phase 2. The analysis results of our case study show that Scenario 2 is more cost-effective than Scenario 1.

For future work, we will extend our current work for components with non-constant failure rates. We will adopt a measurement-based approach to collecting empirical data in order to determine the probability density function of the system reliability affected by software aging. Software tools will also be developed for modeling and analyzing the reliability of cloud-based systems, as well as deriving effective rejuvenation schedules. Finally, modeling and analyzing cloud-based systems with active standby spare components that can share workload with the primary ones, is envisioned as a future, and more ambitious research direction.

REFERENCES

[1] H. Pham, *System Software Reliability*, Springer Series in Reliability Engineering, Springer-Verlag London, 2006.

[2] A. Somani and N. Vaidya, "Understanding Fault Tolerance and Reliability," *IEEE Computer*, Vol. 30, No. 4, April 1997, pp. 45-50.

[3] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton, "Software Rejuvenation: Analysis, Module and Applications," *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing,* 1995, pp. 381-390.

[4] M. Grotte, R. Matias and K. S. Trivedi, "The Fundamentals of Software Aging," *Proceedings of Workshop on Software Aging and Rejuvenation*, ISSRE, Nov. 11-14, 2008, pp. 1-6.

[5] V. Castelli, R.E. Harper, and P. Heidelberger, *et al.*, "Proactive Management of Software Aging," *IBM Journal of Research and Development*, Vol. 45, No. 2, March 2001, pp. 311-332.

[6] L. Jiang and G. Xu, "Modeling and Analysis of Software Aging and Software Failure," *Journal of Systems and Software*, Vol. 80, No. 4, April 2007, pp. 590-595.

[7] A. Bobbio, M. Sereno and C. Anglano, "Fine Grained Software Degradation Models for Optimal Rejuvenation Policies," *Performance Evaluation*, Vol. 46, 2001, pp. 45-62.

[8] K. Vaidyanathan, D. Selvamuthu and K. S. Trivedi, "Analysis of Inspection-Based Preventive Maintenance in Operational Software Systems," *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS 2002)*, Suita, Japan, 2002, pp. 286-295.

[9] F. Machida, A. Andrzejak, R. Matias, E. Vicente, "On the Effectiveness of Mann-Kendall Test for Detection of Software Aging," *Proceedings of the IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Pasadena, CA, November 4-7, 2013, pp. 269-274.

[10] M. Grottke, L. Li, K. and Vaidyanathan, *et al.*, "Analysis of Software Aging in a Web Server," *IEEE Transactions on Reliability,* Vol. 55, No. 3, 2006, pp. 411-420.