SOFTWARE REJUVENATION - MODELING AND ANALYSIS

Kishor S. Trivedi
Dept. of Electrical & Computer Engineering
Duke University, Durham, NC 27708, USA
kst@ee.duke.edu

Kalyanaraman Vaidyanathan
Sun Microsystems, Inc.
San Diego, CA 92121, USA
kalyan.vaidyanathan@sun.com

Abstract Several recent studies have established that most system outages are due to software faults. Given the ever increasing complexity of software and the well-developed techniques and analysis for hardware reliability, this trend is not likely to change in the near future. In this paper, we first classify software faults and discuss various techniques to deal with them in the testing/debugging phase and the operational phase of the software. We discuss the phenomenon of software aging and a preventive maintenance technique to deal with this problem called software rejuvenation. Stochastic models to evaluate the effectiveness of preventive maintenance in operational software systems and to determine optimal times to perform rejuvenation for different scenarios are described. We also present measurement-based methodologies to detect software aging and estimate its effect on various system resources. These models are intended to help develop software rejuvenation policies. An automated online measurement-based approach has been used in the software rejuvenation agent implemented in a major commercial server.

Keywords: Availability, Measurement-based dependability evaluation, Software reliability, Software aging, Software rejuvenation

1. Introduction

Several studies have now shown that outages in computer systems are more due to software faults than due to hardware faults [24, 42]. Recent studies have also reported the phenomenon of "software aging" [20, 29] in which the state of the software degrades with time. The primary causes of this degradation are the exhaustion of operating system resources, data corruption and numerical error accumulation. Eventually, this may lead to performance degradation of the software or crash/hang failure or both. Some common examples of "software aging" are memory bloating and leaking, unreleased file-locks, data corruption, storage space fragmentation and accumulation of round-off errors [20]. Aging has not only been observed in software used on a mass scale but also in specialized software used in high-availability and safety-critical applications [29]. Since aging leads to transient failures in software systems, environment diversity, a software fault tolerance technique, can be employed proactively to prevent degradation or crashes. This involves occasionally stopping the running software, "cleaning" its internal state or its environment and restarting it. Such a technique known as "software rejuvenation" was proposed by Huang et al. [29].[1] This counteracts the aging phenomenon in a proactive manner by removing the accumulated error conditions and freeing up operating system resources. Garbage collection, flushing operating system kernel tables and reinitializing internal data structures are some examples by which the internal state or the environment of the software can be cleaned.

Software rejuvenation has been implemented in the AT&T billing applications [29]. An extreme example of a system level rejuvenation, proactive hardware reboot, has been implemented in the real-time system collecting billing data for most telephone exchanges in the United States [7]. Occasional reboot is also performed in the AT&T telecommunications switching software [3]. On reboot, called *software capacity restoration,* the service rate is restored to its peak value. On-board preventive maintenance in spacecraft has been proposed and analyzed by Tai et al. [43]. This maximizes the probability of successful mission completion by the spacecraft. These operations, called *operational redundancy,* are invoked whether or not faults exist. Proactive fault management was also recommended for the Patriot missiles' software system [36]. A warning was issued saying that a very long running time could affect the targeting accuracy. This decrease in accuracy was evidently due to error accumulation caused by software aging. The warning however failed to inform the troops how many hours "very long" was and that it would help if the computer system was switched off and on every eight hours. This exemplifies the necessity and the use of proactive fault management even in safety critical systems. More recently, rejuvenation has been implemented in cluster systems to improve performance and availability [11, 30, 47]. Two kinds of policies have been implemented taking advantage of the cluster failover feature. In the periodic policy, rejuvenation of the cluster nodes is done in a rolling fashion after every deterministic interval. In the prediction-based policy, the time to rejuvenate is estimated based on the collection and statistical analysis of system data. The implementation and analysis are described in detail in [11, 47]. A software rejuvenation feature known as process recycling has been implemented in the Microsoft IIS 5.0 web server software [48]. The popular web server software Apache implements a form of rejuvenation by killing and recreating processes after a certain numbers of requests have been served [34, 49]. Software rejuvenation is also implemented in specialized transaction processing servers [10]. Rejuvenation has also been proposed for cable and DSL modem gateways [15], in Motorola's Cable Modem Termination System [35] and in middleware applications [9] for failure detection and prevention. Automated rejuvenation strategies have been proposed in the context of self-healing and autonomic computing systems [27]. Software rejuvenation (preventive maintenance) incurs an overhead (in terms of performance, cost and downtime) which should be balanced against the loss incurred due to unexpected outage caused by a failure. Thus, an important research issue is to determine the optimal times to perform rejuvenation.

In this paper, we present two approaches for analyzing software aging and studying aging-related failures. The rest of this paper is organized as follows. In Section 2, we show how to include faults attributed to software aging into the framework of traditional classification of software faults - deterministic and transient. We also study the treatment and recovery strategies for each of the fault classes, discussing the relative advantages and disadvantages. This will help us choose the best possible recovery strategy when a fault is triggered and the system experiences a crash or a performance degradation. Section 3 describes various analytical models for software aging and to determine optimal times to perform rejuvenation. Measurement-based models are dealt with in Section 4. The implementation of a software rejuvenation agent in a major commercial server is discussed in Section 5. Section 6 describes various approaches and methods of rejuvenation and Section 7 concludes the paper with pointers to future work.

2. Classification and Treatment of Software Faults

In this section, we describe how we can include software faults attributed to software aging into Jim Gray's fault classification [22] and discuss the various fault tolerance techniques to deal with these faults in the operational phase of the software. Particular attention is given to environment diversity, explaining its need, various approaches and methods in practice.

Classification of software faults

Faults, in both hardware and software, can be classified according to their phase of creation or occurrence, system boundaries (internal or external), domain (hardware or software), phenomenological cause, intent and persistence [5]. In this section, we restrict ourselves to the classification software faults based on their phase of creation.

Some studies have suggested that since software is not a physical entity and hence not subject to transient physical phenomena (as opposed to hardware), software faults are permanent in nature [28]. Some other studies classify software faults as both permanent and transient. Gray [22] classifies software faults into *Bohrbugs* and *Heisenbugs.* Bohrbugs are essentially permanent design faults and hence almost deterministic in nature. They can be identified easily and weeded out during the testing and debugging phase (or early deployment phase) of the software life cycle. A software system with Bohrbugs is analogous to a faulty deterministic finite state machine. Heisenbugs, on the other hand, are design faults that behave in a way similar to hardware transient or intermittent faults. Their conditions of activation occur rarely or are not easily reproducible. These faults are extremely dependent on the operating environment (other programs, OS and hardware resources). Hence these faults result in transient failures, i.e., failures which may not recur if the software is restarted. Some typical situations in which Heisenbugs might surface are boundaries between various software components, improper or insufficient exception handling and interdependent timing of various events. It is for this reason that Heisenbugs are extremely difficult to identify through testing. In fact, any attempt to detect such a bug may alter the operating environment enough to change the symptoms. A software system with Heisenbugs is analogous to a faulty non-deterministic finite state machine. A mature piece of software in the operational phase, released after its development and testing stage, is more likely to experience failures caused by Heisenbugs than due to Bohrbugs. Most recent studies on failure data have reported that a large proportion of software failures are transient in nature [22, 23], caused by phenomena such as overloads or timing and exception errors [12, 42]. The study of failure data from Tandem's fault tolerant computer system indicated that 70% of the failures were transient failures, caused by race conditions and timing problems [33].

We now describe how to explicitly account for the phenomenon of software aging in Gray's classification of software faults. We designate faults attributed to software aging as *aging-related* faults. Aging-related faults can fall under Bohrbugs or Heisenbugs depending on whether the failure is deterministic (repeatable) or transient.

Figure 1 illustrates this classification of software faults. Following are examples of software faults in each of these categories. A software fault which is environment independent and hence deterministic, falls under the category of non-aging related Bohrbug (for example, a set of inputs resulting in the same failure every time). If the software bug, for example, is related to the arrival order of messages to a process, it is classified as a non-aging related Heisenbug. Reorder of messages and replay might result in the system working correctly. A bug causing a gradual resource exhaustion deterministically every time is classified as an aging-related Bohrbug. A bug causing an unknown resource leak during rare instances which are difficult to reproduce could be classified as an aging-related Heisenbug.

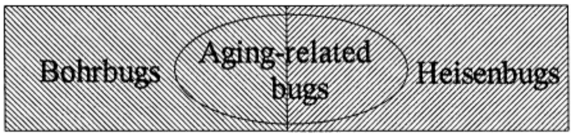

Figure 1. Venn diagram of software fault types

Software fault tolerance techniques

Design diversity [4] has been advocated as a technique for software fault tolerance. The design diversity approach was developed mainly to deal with Bohrbugs. It relies on the assumption of independence between multiple variants of software. However, as some studies have shown, this assumption may not always be valid [32]. Design diversity can also be used to treat Heisenbugs. Since there are multiple versions of software operating, it not likely that all of them will experience the same transient failure. One of the disadvantages of design diversity is the high cost involved in developing multiple variants of software.

Data diversity [2] can work well with Bohrbugs and is less expensive to implement than design diversity. To some extent, data diversity can also deal with Heisenbugs since different input data is presented and by definition, these bugs are non-deterministic and non-repeatable.

Environment diversity is the simplest technique for software fault tolerance and it effectively deals with Heisenbugs and aging-related bugs. Although this technique has been used for long in an *ad hoc* manner, only recently has it gained recognition and importance. Having its basis on the observation that most software failures are transient in nature, environment diversity utilizes reexecuting the software in a different environment [31].

Adams [1] has proposed restarting the system as the best approach to masking software faults. Environment diversity, a generalization of restart, has been proposed in [28, 31] as a cheap but effective technique for fault tolerance in software. Transient faults typically occur in computer systems due to design faults in software which result in unacceptable and erroneous states in the OS environment. Therefore, environment diversity attempts to provide a new or modified operating environment for the running software. Usually, this is done at the instance of a failure in the software. When the software fails, it is restarted in a different, error-free OS environment state which is achieved by some clean up operations. Examples of environment diversity techniques include retry operation, restart application and rebooting the node. The retry and restart operations can be done on the same node or on another spare (cold/warm/hot) node.

Tandem's fault tolerant computer system [33] is based on the *process pair* approach. It was noted that many application failures did not recur once the application was restarted on the second processor. This was due to the fact that the second processor provided a different environment which did not trigger the same error conditions which led to the failure of the application on the first processor. Hence, in this case (as well as in Avaya's SwiFT [21]), hardware redundancy coupled with software replication[2] was used to tolerate most of the software faults.

The basic observation in all these transient failures is that the same error condition is unlikely to occur if the software is reexecuted in a different environment. For aging-related bugs, environment diversity can be particularly effective if utilized proactively in the form of software rejuvenation.

3. Analytic Models for Software Rejuvenation

The aim of the analytic modeling is to determine optimal times to perform rejuvenation which maximize *availability* and minimize the *probability of loss* or the *response time of a transaction* (in the case of a transaction processing system). This is particularly important for business-critical applications for which adequate response time can be as important as system uptime. The analysis is done for different kinds of software systems exhibiting varied failure/aging characteristics.

The accuracy of a modeling based approach is determined by the assumptions made in capturing aging. In [16–18, 29, 43] only the failures causing unavailability of the software are considered, while in [38] only a gradually decreasing service rate of a software which serves transactions is assumed. Garg et al. [19], however, consider both these effects of aging together in a single model. Models proposed in [16, 17, 29] are restricted to hypo-exponentially distributed time to failure. Those proposed in [18, 38, 43] can accommodate general distributions but only for the specific aging effect they capture. Generally distributed time to failure, as well as the service rate being an arbitrary function of time are allowed in [19]. It has been noted [42] that transient failures are partly caused by overload conditions. Only the model presented by Garg et al. [19] captures the effect of load on aging. Existing models also differ in the measures being evaluated. In [18, 43] software with a finite mission time is considered. In the [16, 17, 19, 29] measures of interest in a transaction based software intended to run forever are evaluated.

Bobbio et al.[8] present fine grained software degradation models, where one can identify the current degradation level based on the observation of a system parameter, are considered. Optimal rejuvenation policies based on a risk criterion and an alert threshold are then presented. Dohi et al. [13, 14] present software rejuvenation models based on semi-Markov processes. The models are analyzed for optimal rejuvenation strategies based on cost as well as steady-state availability. Given a sample data of failure times, statistical non-parametric algorithms based on the total time on test transform are presented to obtain the optimal rejuvenation interval.

Basic model for rejuvenation

Figure 2 shows the basic software rejuvenation model proposed by Huang et al. [29]. The software system is initially in a "robust" working state, 0. As time progresses, it eventually transits to a "failure-probable" state 1. The system is still operational in this state but can fail (move to state 2) with a non-zero probability. The system can be repaired and brought back to the initial state 0. The software system is also rejuvenated at regular intervals from the failure probable state 1 and brought back to the robust state 0.

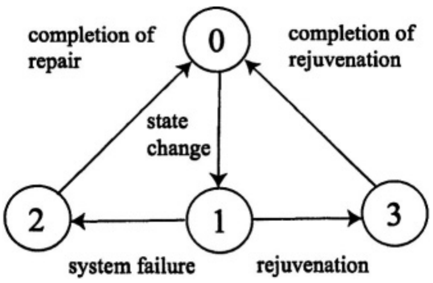

Figure 2. State transition diagram for rejuvenation

Huang et al. [29] assume that the stochastic behavior of the system can be described by a simple continuous-time Markov chain (CTMC) [45]. Let Z be the random time interval when the highly robust state changes to the failure probable state, having the exponential distribution $\Pr\{Z \le t\} = F_0(t) = 1 - \exp(-t/\mu_0)$ $(\mu_0 > 0)$. Just after the state becomes the failure probable state, a system failure may occur with a positive probability. Without loss of generality, we assume that the random variable Z is observable during the system operation. Define the failure time X (from state 1) and the repair time Y, having the exponential distributions $\Pr\{X \le t\} = F_f(t) = 1 - \exp(-t/\lambda_f)$ and $\Pr\{Y \le t\} = F_a(t) = 1 - \exp(-t/\mu_a)$ $(\lambda_f > 0,\ \mu_a > 0)$. If the system failure occurs before triggering a software rejuvenation, then the repair is started immediately at that time and is completed after the random time Y elapses. Otherwise, the software rejuvenation is started. Note that the software rejuvenation cycle is measured from the time instant just after the system enters state 1. Define the distribution functions of the time to invoke the software rejuvenation and of the time to complete software rejuvenation by $F_r(t) = 1 - \exp(-t/\mu_r)$ and $F_c(t) = 1 - \exp(-t/\mu_c)$ $(\mu_c > 0,\ \mu_r > 0)$, respectively. The CTMC is then analyzed and the expected system down time and the expected cost per unit time in the steady state is computed. An optimal rejuvenation interval which minimizes expected downtime (or expected cost) is obtained.

It is not difficult to introduce the periodic rejuvenation schedule and to extend the CTMC model to the general one. Dohi et al. [13, 14] developed semi-Markov models with the periodic rejuvenation and general transition distribution functions. More specifically, let Z be the random variable having the common distribution function $\Pr\{Z \leq t\} = F_0(t)$ with finite mean μ_0 (> 0). Also, let X and Y be the random variables having the common distribution functions $\Pr\{X \leq t\} = F_f(t)$ and $\Pr\{Y \leq t\} = F_a(t)$ with finite means λ_f (> 0) and μ_a (> 0), respectively. Denote the distribution function of the time to invoke the software rejuvenation and the distribution of the time to complete software rejuvenation by $F_r(t)$ and $F_c(t)$ (with mean μ_c (> 0)), respectively. After completing the repair or the rejuvenation, the software system becomes as good as new, and the software age is initiated at the beginning of the next highly robust state. Consequently, we define the time interval from the beginning of the system operation to the next one as one cycle, and the same cycle is repeated again and again. The time to software rejuvenation (the rejuvenation interval) is a constant, t_0, i.e., $F_r(t) = U(t - t_0)$, where $U(\cdot)$ is the unit step function.

The underlying stochastic process is a semi-Markov process with four regeneration states. If the sojourn times in all states are exponentially distributed, this model is the CTMC in Huang et al. [29]. Using the renewal theory [39], the steady-state system availability is computed as

$$\begin{aligned} A(t_0) &= \Pr\Big\{\text{software system is operative in the steady state}\Big\} \\ &= \frac{\mu_0 + \int_0^{t_0} \overline{F}_f(t)dt}{\mu_0 + \mu_a F_f(t_0) + \mu_c \overline{F}_f(t_0) + \int_0^{t_0} \overline{F}_f(t)dt} \\ &= S(t_0)/T(t_0), \end{aligned} \tag{1}$$

where in general $\overline{\phi}(\cdot) = 1 - \phi(\cdot)$ The problem is to derive the optimal software rejuvenation interval t_0^* which maximizes the system availability in the steady state $A(t_0)$. We make the following assumption that the mean time to repair is strictly larger than the mean time to complete the software rejuvenation (i.e., $\mu_a > \mu_c$). This assumption is quite reasonable and intuitive. The following result gives the optimal software rejuvenation schedule for the semi-Markov model.

Assume that the failure time distribution is strictly IFR (increasing failure rate) [45]. Define the following non-linear function:

$$q(t_0) = T(t_0) - \left\{(\mu_a - \mu_c)r_f(t_0) + 1\right\}S(t_0), \tag{2}$$

where $r_f(t) = (dF_f(t)/dt)/\overline{F}_f(t)$ is the failure rate.

(i) If $q(0) > 0$ and $q(\infty) < 0$, then there exists a finite and unique optimal software rejuvenation schedule t_0^* $(0 < t_0^* < \infty)$ satisfying $q(t_0^*) = 0$, and the maximum system availability is

$$A(t_0^*) = \frac{1}{(\mu_a - \mu_c)r_f(t_0^*) + 1}. \tag{3}$$

(ii) If $q(0) \leq 0$, then the optimal software rejuvenation schedule is $t_0^* = 0$, *i.e.* it is optimal to start the rejuvenation just after entering the failure probable state, and the maximum system availability is $A(0) = \mu_0/(\mu_0 + \mu_c)$.

(iii) If $q(\infty) \geq 0$, then the optimal rejuvenation schedule is $t_0^* \to \infty$, *i.e.* it is optimal not to carry out the rejuvenation, and the maximum system availability is $A(\infty) = (\mu_0 + \lambda_f)/(\mu_0 + \mu_a + \lambda_f)$.

If the failure time distribution is DFR (decreasing failure rate), then the system availability $A(t_0)$ is a convex function of t_0, and the optimal rejuvenation schedule is $t_0^* = 0$ or $t_0^* \to \infty$ [13, 14].

Garg et al. [16] have developed a Markov Regenerative Stochastic Petri Net (MRSPN) model where rejuvenation is performed at deterministic intervals assuming that the failure probable state 1 is not observable.

Preventive maintenance in transactions based software systems

In [19], Garg et al. consider a transaction-based software system whose macro-states representation is presented in Figure 3. The state in which the software is available for service (albeit with decreasing service rate) is denoted as state *A*. After failure a recovery procedure is started. In state *B* the software is recovering from failure and is unavailable for service. Lastly, the software occasionally undergoes preventive maintenance (PM), denoted by state *C*. PM is allowed only from state *A*. Once recovery from failure or PM is complete, the software is reset to state *A* and is as good as new. From this moment, which constitutes a renewal, the whole process stochastically repeats itself.

The system consists of a server type software to which transactions arrive at a constant rate λ. Each transaction receives service for a random period. The service rate of the software is an arbitrary function measured from the

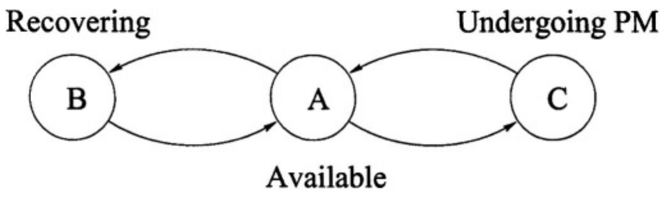

Figure 3. Macro-states representation of the software behavior

last renewal of the software (because of aging) denoted by $\mu(\cdot)$. Therefore, a transaction which starts service at time t_1, occupies the server for a time whose distribution is given by $1 - e^{-\int_{t_1}^{t} \mu(\cdot)\, dt}$. If the software is busy processing a transaction, arriving customers are queued. Total number of transactions that the software can accommodate is K (including the one being processed) and any more arriving when the queue is full are lost. The service discipline is FCFS. The software fails with a rate $\rho(\cdot)$, that is, the *CDF* of the time to failure X is given by $F_X(t) = 1 - e^{-\int_0^t \rho(\cdot)\, dt}$. Times to recover from failure Y_f and to perform PM Y_r are random variables with associated general *CDFs* F_{Y_f} and F_{Y_r} respectively. The model does not require any assumptions on the nature of F_{Y_f} and F_{Y_r}. Only the respective expectations $\gamma_f = E[Y_f]$ and $\gamma_r = E[Y_r]$ are assumed to be finite. Any transactions in the queue at the time of failure or at the time of initiation of PM are assumed to be lost. Moreover, any transactions which arrive while the software is recovering or undergoing PM are also lost.

The effect of aging in the model may be captured by using decreasing service rate and increasing failure rate, where the decrease or the increase respectively can be a function of time, instantaneous load, mean accumulated load or a combination of the above.

Two policies which can be used to determine the time to perform PM are considered. Under policy I which is purely time-based, PM is initiated after a constant time δ has elapsed since it was started (or restarted). Under policy II, which is based on instantaneous load and time, a constant waiting period δ must elapse before PM is attempted. After this time PM is initiated if and only if there are no transactions in the system. Otherwise, the software waits until the queue is empty upon which PM is initiated. The actual PM interval under Policy II is determined by the sum of PM wait δ and the time it takes for the queue to get empty from that point onwards B. Since the latter quantity is dependent on system parameters and can not be controlled, the actual PM interval has a range $[\delta, \infty)$.

Given the above behavioral model the following measures are derived for each policy: steady state availability of the software A_{SS}, long run probability of loss of a transaction P_{loss}, and expected response time of a transaction given that it is successfully served T_{res}. The goal is to determine optimal values of δ (PM interval under policy I and PM wait under policy II) based on the constraints on one or more of these measures.

According to the model described above at any time t the software can be in any one of three states: up and available for service (state *A*), recovering from a failure (state *B*) or undergoing PM (state *C*). Let $\{Z(t), t \geq 0\}$ be a stochastic process which represents the state of the software at time t. Further, let the sequence of random variables $S_i, i > 0$ represent the times at which transitions among different states take place. Since the entrance times S_i constitute renewal points $\{Z(S_i), i > 0\}$ is an embedded discrete time Markov chain (DTMC) with a transition probability matrix *P* given by:

$$\mathbf{P} = \begin{bmatrix} 0 & P_{AB} & P_{AC} \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}. \tag{4}$$

The steady state probability π_i of the DTMC being in state $i, i \in \{A, B, C\}$ is:

$$\pi = [\pi_A, \pi_B, \pi_C] = \left[\frac{1}{2}, \frac{1}{2}P_{AB}, \frac{1}{2}P_{AC}\right]. \tag{5}$$

The software behavior is modeled via the stochastic process $\{(Z(t), N(t)), t \geq 0\}$. If $Z(t) = A$, then $N(t) \in \{0, 1, \ldots, K\}$ as the queue can accommodate up to *K* transactions. If $Z(t) \in \{B, C\}$, then $N(t) = 0$, since by assumption all transactions arriving while the software is either recovering or undergoing PM are lost. Further, the transactions already in the queue at the transition instant are also discarded. It can be shown that the process $\{(Z(t), N(t)), t \geq 0\}$ is a Markov regenerative process (MRGP). Transition to state *A* from either *B* or *C* constitutes a regeneration instant.

Let *U* be a random variable denoting the sojourn time in state *A*, and denote its expectation by *E[U]*. Expected sojourn times of the MRGP in states *B* and *C* are already defined to be γ_f and γ_r. The steady state availability is obtained using the standard formulae from MRGP theory:

$$A_{SS} = Pr\{\textit{software is in state } A\}$$

$$= \frac{\pi_A E[U]}{\pi_B \gamma_f + \pi_C \gamma_r + \pi_A E[U]}. \tag{6}$$

The probability that a transaction is lost is defined as the ratio of expected number of transactions which are lost in an interval to the expected total number of transactions which arrive during that interval. Since the evolution of

$\{Z(t), N(t)), t > 0\}$ in the intervals comprising of successive visits to state *A* is stochastically identical it suffices to consider just one such interval. The number of transactions lost is given by the summation of three quantities: (1) transactions in the queue when the system is exiting state *A* because of the failure or initiation of PM (2) transactions that arrive while failure recovery or PM is in progress and (3) transactions that are disregarded due to the buffer being full. The last quantity is of special significance since the probability of buffer being full will increase due to the degrading service rate. It follows that the probability of loss is given by

$$P_{loss} = \frac{\pi_A E[N_l] + \lambda\left(\pi_B\gamma_f + \pi_C\gamma_r + \pi_A \int_0^\infty p_K(t)dt\right)}{\lambda\left(\pi_B\gamma_f + \pi_C\gamma_r + \pi_A E[U]\right)} \tag{7}$$

where $E[N_l]$ is the expected number of transactions in the buffer when the system is exiting state *A*. Equation 7 is valid only for policy II. Under policy I sojourn time in state *A* is limited by δ, so the upper limit in the integral $\int_0^\infty p_K(t)dt$ is δ instead of ∞.

Next an upper bound on the mean response time of a transaction given that it is successfully served, T_{res}, is derived. The mean number of transactions, denoted by *E,* which are accepted for service while the software is in state *A* is given by the mean number of transactions which are not accepted due to the buffer being full, subtracted from the mean total number of transactions which arrive while the software is in state *A,* that is, $E = \lambda\left[E[U] - \int_{t=0}^\infty p_K(t)dt\right]$. Out of these transactions, on the average, $E[N_l]$ are discarded later because of failure or initiation of PM. Therefore, the mean number of transactions which actually receive service given that they were accepted is given by $E - E[N_l]$. The mean total amount of time the transactions spent in the system while the software is in state *A* is $W = \int_{t=0}^\infty \sum_i ip_i(t)\ dt$. This time is composed of the mean time spent by the transactions which were served as well as those which were discarded, denoted as W_S and W_D, respectively. Therefore, $W = W_S + W_D$. The response time we are interested in is given by $T_{res} = W_S/(E - E[N_l])$, which is upper bounded by $T_{res} < \frac{W}{E-E[N_l]}$.

$p_i(t)$ is the probability that there are i transactions queued for service, which is also the probability of being in state i of the subordinated process at time t. $p_{i'}(t)$ is the probability that the system failed when there were i transactions queued for service. These transient probabilities for both policies can be obtained by solving the systems of forward differential-difference equations given in [19]. In general they do not have a closed-form analytical solution and must be evaluated numerically. Once these probabilities are obtained, the rest of the quantities P_{AB}, P_{AC}, *E[U]* and $E[N_l]$ can be easily computed [19] and then used to obtain the steady state availability A_{SS}, the probability of transaction lost P_{loss} and the upper bound on the response time of a transaction T_{res}.

Examples are presented to illustrate the usefulness of the presented model in determining the optimum value of δ (PM interval in the case of policy I and PM wait in the case of policy II). First, the service rate and failure rate are assumed to be functions of real time, where $\rho(t)$ is defined to be the hazard function of Weibull distribution, while $\mu(t)$ is defined to be a monotone non-increasing function that approximates the service degradation. Figure 4 shows A_{ss} and P_{loss} for both policies plotted against δ for different values of the mean time to perform PM γ_r. Under both policies, it can be seen that for any

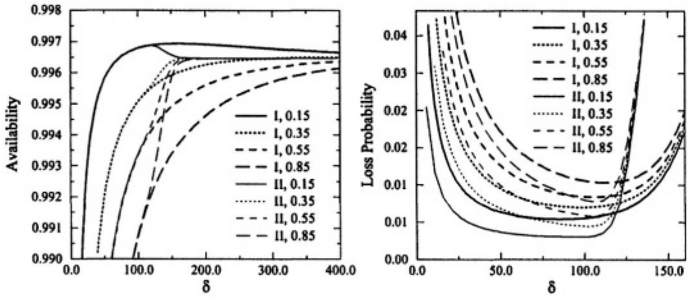

Figure 4. Results for experiment 1

particular value of δ, higher the value of γ_r, lower is the availability and higher is the corresponding loss probability. It can also be observed that the value of δ which minimizes probability of loss is much lower than the one which maximizes availability. In fact, the probability of loss becomes very high at values of δ which maximize availability. For any specific value of γ_r, policy II results in a lower minima in loss probability than that achieved under policy I. Therefore, if the objective is to minimize long run probability of loss, such as in the case of telecommunication switching software, policy II always fares better than policy I.

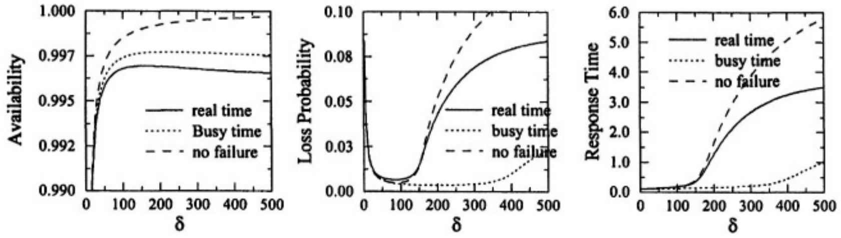

Figure 5. Results of experiment 2

Figure 5 shows A_{SS}, P_{loss} and upper bound on T_{res} plotted against δ under policy I. Each of the figures contains three curves. $\mu(\cdot)$ and $\rho(\cdot)$ in the solid curve are functions of real time $\mu(t)$ and $\rho(t)$, whereas in the dotted curve they are functions (with the same parameters) of the mean total processing time $\mu(L(t))$ and $\rho(L(t))$. The dashed curve represents a third system in which no crash/hang failures occur $\rho(\cdot) = 0$, but service degradation is present with $\mu(\cdot) = \mu(t)$. This experiment illustrates the importance of making the right assumptions in capturing aging because as seen from the figure, depending on the forms chosen for $\mu(\cdot)$ and $\rho(\cdot)$, the measures vary in a wide range.

Software rejuvenation in a cluster system

Software rejuvenation has been applied to cluster systems [11, 47]. This is a novel application, which significantly improves cluster system availability and productivity. The Stochastic Reward Net (SRN) model of a cluster system employing simple time-based rejuvenation is shown in Figure 6. The cluster consists of n nodes which are initially in a "robust" working state, P_{up}. The aging process is modeled as a 2-stage hypo-exponential distribution (increasing failure rate) [45] with transitions T_{fprob} and $T_{noderepair}$. Place P_{fprob} represents a "failure-probable" state in which the nodes are still operational. The nodes then can eventually transit to the fail state, $P_{nodefail1}$. A node can be repaired through the transition $T_{noderepair}$, with a coverage *c*. In addition to individual node failures, there is also a common-mode failure (transition T_{cmode}). The system is also considered down when there are a $(a \leq n)$ individual node failures. The system is repaired through the transition $T_{sysrepair}$.

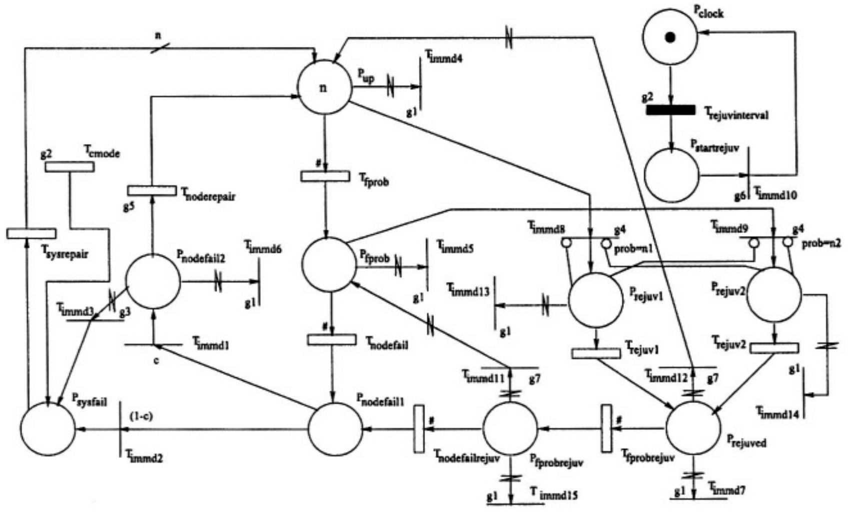

Figure 6. SRN model of a cluster system employing simple time-based rejuvenation

classified as an aging-related Bohrbug. A bug causing an unknown resource leak during rare instances which are difficult to reproduce could be classified as an aging-related Heisenbug.

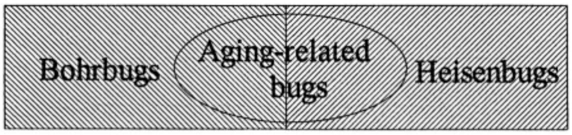

Figure 1. Venn diagram of software fault types

Software fault tolerance techniques

Design diversity [4] has been advocated as a technique for software fault tolerance. The design diversity approach was developed mainly to deal with Bohrbugs. It relies on the assumption of independence between multiple variants of software. However, as some studies have shown, this assumption may not always be valid [32]. Design diversity can also be used to treat Heisenbugs. Since there are multiple versions of software operating, it not likely that all of them will experience the same transient failure. One of the disadvantages of design diversity is the high cost involved in developing multiple variants of software.

Data diversity [2] can work well with Bohrbugs and is less expensive to implement than design diversity. To some extent, data diversity can also deal with Heisenbugs since different input data is presented and by definition, these bugs are non-deterministic and non-repeatable.

Environment diversity is the simplest technique for software fault tolerance and it effectively deals with Heisenbugs and aging-related bugs. Although this technique has been used for long in an *ad hoc* manner, only recently has it gained recognition and importance. Having its basis on the observation that most software failures are transient in nature, environment diversity utilizes reexecuting the software in a different environment [31].

Adams [1] has proposed restarting the system as the best approach to masking software faults. Environment diversity, a generalization of restart, has been proposed in [28, 31] as a cheap but effective technique for fault tolerance in software. Transient faults typically occur in computer systems due to design faults in software which result in unacceptable and erroneous states in the OS environment. Therefore, environment diversity attempts to provide a new or modified operating environment for the running software. Usually, this is done at the instance of a failure in the software. When the software fails, it is restarted in a different, error-free OS environment state which is achieved by some clean up operations. Examples of environment diversity

For the analyses, the following values are assumed. The mean times spent in places P_{up} and P_{fprob} are 240 hrs and 720 hrs respectively. The mean times to repair a node, to rejuvenate a node and to repair the system are 30 min, 10 min and 4 hrs respectively. In this analysis, the common-mode failure is disabled and node failure coverage is assumed to be perfect. All the models were solved using the SPNP (Stochastic Petri Net Package) tool [26]. The measures computed were expected unavailability and the expected cost incurred over a fixed time interval. It is assumed that the cost incurred due to node rejuvenation is much less than the cost of a node or system failure since rejuvenation can be done at predetermined or scheduled times. In our analysis, we fix the value for $cost_{nodefail}$ at \$5,000/hr, the $cost_{rejuv}$ at \$250/hr. The value of $cost_{sysfail}$ is computed as the number of nodes, n, times $cost_{nodefail}$.

Figure 8 shows the plots for an 8/1 configuration (8 nodes including 1 spare) system employing simple time-based rejuvenation. The upper plot and lower plots show the expected cost incurred and the expected downtime (in hours) respectively in a given time interval, versus rejuvenation interval (time between successive rejuvenation) in hours. If the rejuvenation interval is close to zero, the system is always rejuvenating and thus incurs high cost and downtime. As the rejuvenation interval increases, both expected unavailability and cost incurred decrease and reach an optimum value. If the rejuvenation interval goes beyond the optimal value, the system failure has more influence on these measures than rejuvenation. The analysis was repeated for 2/1, 8/2, 16/1 and 16/2 configurations. For time-based rejuvenation, the optimal rejuvenation interval was 100 hours for the 1-spare clusters, and approximately 1 hour for the 2-spare clusters. In our analysis of condition-based rejuvenation, we assumed 90% prediction coverage. For systems that have one spare, time-based rejuvenation can reduce downtime by 26% relative to no rejuvenation. Condition-based rejuvenation does somewhat better, reducing downtime by 62% relative to no rejuvenation. However, when the system can tolerate more than one failure at a time, downtime is reduced by 98% to 95% via time-based rejuvenation, compared to a mere 85% for condition-based rejuvenation.

4. Measurement Based Models for Software Rejuvenation

While all the analytical models are based on the assumption that the rate of software aging is known, in the measurement based approach, the basic idea is to monitor and collect data on the attributes responsible for determining the health of the executing software. The data is then analyzed to obtain predictions about possible impending failures due to resource exhaustion.

In this section we describe the measurement-based approach for detection and validation of the existence of software aging. The basic idea is to periodically monitor and collect data on the attributes responsible for determining the

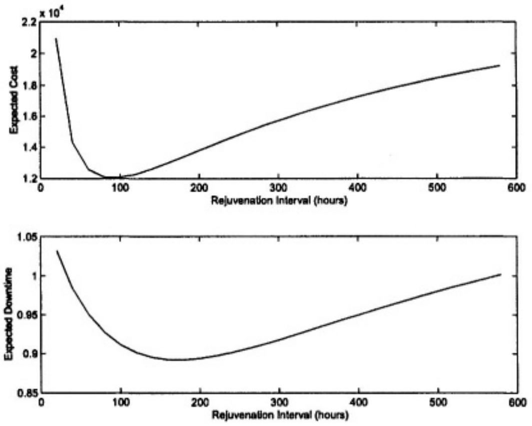

Figure 8. Results for an 8/1 cluster system employing time-based rejuvenation

health of the executing software, in this case the UNIX operating system. Garg et al. [20] propose a methodology for detection and estimation of aging in the UNIX operating system. An SNMP-based distributed resource monitoring tool was used to collect operating system resource usage and system activity data from nine heterogeneous UNIX workstations connected by an Ethernet LAN at the Department of Electrical and Computer Engineering at Duke University. A central monitoring station runs the manager program which sends *get* requests periodically to each of the agent programs running on the monitored workstations. The agent programs in turn obtain data for the manager from their respective machines by executing various standard UNIX utility programs like *pstat, iostat* and *vmstat.* For quantifying the effect of aging in operating system resources, the metric *Estimated time to exhaustion* is proposed. The earlier work [20] uses a purely time-based approach to estimate resource exhaustion times, whereas the the work presented in [46] takes into account the current system workload as well.

A methodology based on time-series analysis to detect and estimate resource exhaustion times due to software aging in a web server while subjecting it to an artificial workload, is proposed in [34]. Avritzer and Weyuker [3] monitor production traffic data of a large telecommunication system and describe a rejuvenation strategy which increases system availability and minimizes packet loss. Cassidy et al. [10] have developed an approach to rejuvenation for large online transaction processing servers. They monitor various system parameters over a period of time. Using pattern recognition methods, they come to the conclusion that 13 of those parameters deviate from normal behavior just prior to a crash, providing sufficient warning to initiate rejuvenation.

Time-based estimation

In the time-based estimation method presented by Garg et al. [20], data was collected from the UNIX machines at intervals of 15 minutes for about 53 days. Time-ordered values for each monitored object are obtained, constituting a time series for that object. The objective is to detect aging or a long term trend (increasing or decreasing) in the values. Only results for the data collected from the machine *Rossby* are discussed here.

First, the trends in operating system resource usage and system activity are detected using *smoothing* of observed data by *robust locally weighted regression,* proposed by Cleveland [20]. This technique is used to get the global trend between outages by removing the local variations. Then, the slope of the trend is estimated in order to do prediction. Figure 9 shows the smoothed data superimposed on the original data points from the time series of objects for Rossby. Amount of *real memory free* (plot 1) shows an overall decrease, whereas *file table size* (plot 2) shows an increase. Plots of some other resources not discussed here also showed an increase or decrease. This corroborates the hypothesis of aging with respect to various objects.

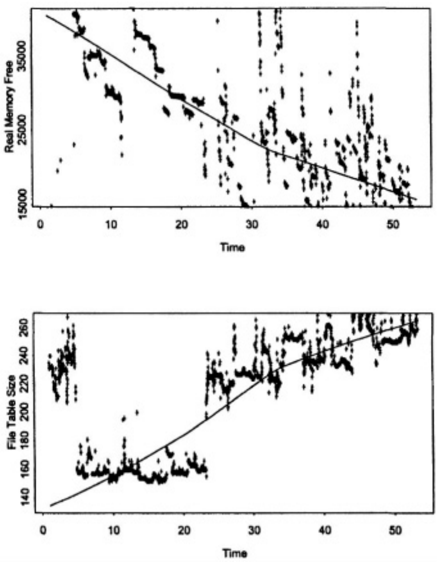

Figure 9. Non-parametric regression smoothing for Rossby objects

The seasonal Kendall test [20] was applied to each of these time series to detect the presence of any global trends at a significance level, α, of 0.05. With Z_α=1.96, all values are such that the null hypothesis (H_0) that no trend

Table 1. **Estimated slope and time to exhaustion for Rossby, Velum and Jefferson objects**

Resource Name	*Initial Value*	*Max Value*	*Sen's Slope Estimation*	*95% Confidence Interval*	*Estimated Time to Exh. (days)*
Rossby					
Real Memory Free	40814.17	84980	-252.00	-287.75 : -219.34	161.96
File Table Size	220	7110	1.33	1.30 : 1.39	5167.50
Process Table Size	57	2058	0.43	0.41 : 0.45	4602.30
Used Swap Space	39372	312724	267.08	220.09 : 295.50	1023.50
Jefferson					
Real Memory Free	67638.54	114608	-972.00	-1006.81 : -939.08	69.59
File Table Size	268.83	7110	1.33	1.30 : 1.38	5144.36
Process Table Size	67.18	2058	0.30	0.29 : 0.31	6696.41
Used Swap Space	47148.02	524156	577.44	545.69 : 603.14	826.07

exists is rejected for the variables considered. Given that a global trend is present and that its slope is calculated for a particular resource, the time at which the resource will be exhausted because of aging only, is estimated. Table 1 refers to several objects on Rossby and lists an estimate of the slope (change per day) of the trend obtained by applying Sen's slope estimate for data with seasons [20]. The values for real memory and swap space are in Kilobytes. A negative slope, as in the case of *real memory,* indicates a decreasing trend, whereas a positive slope, as in the case of *file table size,* is indicative of an increasing trend. Given the slope estimate, the table lists the estimated time to failure of the machine due to aging only with respect to this particular resource. The calculation of the time to exhaustion is done by using the standard linear approximation $y = mx + c$.

A comparative effect of aging on different system resources can be obtained from the above estimates. Overall, it was found that *file table size* and *process table size* are not as important as *used swap space* and *real memoryfree* since they have a very small slope and high estimated times to failure due to exhaustion. Based on such comparisons, we can identify important resources to monitor and manage in order to deal with aging related software failures. For example, the resource *used swap space* has the highest slope and *real memory free* has the second highest slope. However, *real memory free* has a lower time to exhaustion than *used swap space.*

Time and workload-based estimation

The method discussed in the previous subsection assumes that accumulated use of a resource over a time period depends only on the elapsed time. However, it is intuitive that the rate at which a resource is consumed is dependent on the current workload. In this subsection, we discuss a measurement-based model to estimate the rate of exhaustion of operating system resources as a function of both time and the system workload [46]. The SNMP-based distributed resource monitoring tool described previously was used for collecting operating system resource usage and system activity parameters (at 10 min intervals) for over 3 months. Only results for the data collected from the machine Rossby are discussed here. The *longest* stretch of sample points in which no reboots or failures occurred were used for building the model. A semi-Markov reward model [44] is constructed using the data. First different workload states are identified using statistical cluster analysis and a state-space model is constructed. Corresponding to each resource, a reward function based on the rate of resource exhaustion in the different states is then defined. Finally the model is solved to obtain trends and the estimated exhaustion rates and time to exhaustion for the resources.

The following variables were chosen to characterize the system workload - *cpuContextSwitch, sysCall, pageIn,* and*pageOut. Hartigan's k-means clustering algorithm* [25] was used for partitioning the data points into clusters based on workload. The statistics for the eleven workload clusters obtained are shown in Table 2. Clusters whose centroids were relatively close to each other and those with a small percentage of data points in them, were merged to simplify computations. The resulting clusters are $W_1 = \{1,2,3\}$, $W_2 = \{4,5\}$, $W_3 = \{6\}$, $W_4 = \{7\}$, $W_5 = \{8\}$, $W_6 = \{9\}$, $W_7 = \{10\}$ and $W_8 = \{11\}$.

Table 2. **Statistics for the workload clusters**

No.	Cluster Center				% of
	cpuConSw	sysCall	pgOut	pgIn	pts.
1	48405.16	94194.66	5.16	677.83	0.98
2	54184.56	122229.68	5.39	81.41	0.76
3	34059.61	193927.00	0.02	136.73	0.93
4	20479.21	45811.71	0.53	243.40	1.89
5	21361.38	37027.41	0.26	12.64	7.17
6	15734.65	54056.27	0.27	14.45	6.55
7	37825.76	40912.18	0.91	12.21	11.77
8	11013.22	38682.46	0.03	10.43	42.87
9	67290.83	37246.76	7.58	19.88	4.93
10	10003.94	32067.20	0.01	9.61	21.23
11	197934.42	67822.48	415.71	184.38	0.93

Transition probabilities from one state to another were computed from data, resulting in transition probability matrix P of the embedded discrete time

Markov chain The sojourn time distribution for each of the workload states was fitted to either 2-stage hyper-exponential or 2-stage hypo-exponential distribution functions. The fitted distributions were tested using the Kolmogorov-Smirnov test at a significance level of 0.01.

Two resources, *usedSwapSpace* and *realMemoryFree,* are considered for the analysis, since the previous time-based analysis suggested that they are critical resources. For each resource, the reward function is defined as the rate of corresponding resource exhaustion in different states. The true slope (rate of increase/decrease) of a resource at every workload state is estimated by using Sen's non-parametric method [46]. Table 3 shows the slopes with 95% confidence intervals.

It was observed that slopes in a given workload state for a particular resource during different visits to that state are almost the same. Further, the slopes across different workload states are different and generally higher the system activity, higher is the resource utilization. This validates the assumption that resource usage *does* depend on the system workload and the rates of exhaustion vary with workload changes. It can also be observed from Table 3 that the slopes for *usedSwapSpace* in all the workload states are non-negative, and the slopes for *realMemoryFree* are non-positive in all the workload states except in one. It follows that *usedSwapSpace* increases whereas *realMemoryFree* decreases over time which validates the software aging phenomenon.

Table 3. **Slope estimates (in KB/10 min)**

State	usedSwapSpace		realMemoryFree	
	Slope Est.	95 % Conf. Interval	Slope Est.	95 % Conf. Interval
W_1	119.3	5.5 - 222.4	-133.7	-137.7 - -133.3
W_2	0.57	0.40 - 0.71	-1.47	-1.78 - -1.09
W_3	0.76	0.73 - 0.80	-1.43	-2.50 - -0.62
W_4	0.57	0.00 - 0.69	-1.23	-1.67 - -0.80
W_5	0.78	0.75 - 0.80	0.00	-5.65 - 6.00
W_6	0.81	0.64 - 1.00	-1.14	-1.40 - -0.88
W_7	0.00	0.00 - 0.00	0.00	0.00 - 0.00
W_8	91.8	72.4 - 111.0	91.7	-369.9 - 475.2

The semi-Markov reward model was solved using the SHARPE tool [40] developed by researchers at Duke University. The slope for the workload-based estimation is computed as the expected reward rate in steady state from the model. The times to resource exhaustion is computed as the job completion time (mean time to accumulate x amount of reward) of the Markov reward model. Table 4 gives the estimates for the slope and time to exhaustion for

Table 4. **Estimates for slope (in KB/10 min) and time to exhaustion (in days) for *usedSwapSpace* and *realMemoryFree***

Method of Estimation	usedSwapSpace			realMemoryFree		
	Slope Estimate	95 % Conf. Interval	Est. Time to Exh.	Slope Estimate	95 % Conf. Interval	Est. Time to Exh.
Time based	0.787	0.786 - 0.788	2276.46	-2.806	-3.026 - -2.630	60.81
Workload based	4.647	1.191 - 7.746	490.50	-4.1435	-9.968 - 2.592	41.38

usedSwapSpace and *realMemoryFree.* It can be seen that workload based estimations gave a lower time to resource exhaustion than those computed using time based estimations. Since the machine failures due to resource exhaustion were observed much before the times to resource exhaustion estimated by the time based method, it follows that the workload based approach results in better estimations.

Time Series and ARMA Models

In this section, a measurement-based approach based on time-series analysis to detect software aging and to estimate resource exhaustion times due to aging in a web server is described [34]. The experiments are conducted on an Apache web server running on the Linux platform. Before carrying out other experiments, the capacity of the web server is determined so that the appropriate workload to use in the experiments can be decided. The capacity of the web server was found to be around 390 requests/sec. In the next part of the experiment, the web server was run without rejuvenation for a long time until the performance degraded or until the server crashed. The requests were generated by *httperf* [37] to get one of five specified files from the server of sizes 500 bytes, 5KB, 50KB, 500KB and 5MB. The corresponding probabilities that a given file is requested are: 0.35, 0.5, 0.14, 0.009 and 0.001, respectively. During the period of running, the performance measured by the workload generator and system parameters collected by the Linux system monitoring tool, *procmon,* were recorded.

The first data set was collected in a 7-day period with a connection rate of 350 requests/sec. The second set was collected in a 25-day period with connection rate of 400 request/sec. During the experiment, we recorded more than 100 parameters, but for our modeling purposes, six representative parameters pertaining to system resources were selected (Table 5). In addition to the six system status parameters, the response time of the web server, recorded by

Table 5. **Analyzed parameters and their physical meaning**

Parameter	Physical meaning
PhysicalMemoryFree	Free physical memory
SwapSpaceUsed	Used swap space
LoadAvg5Min	Average CPU load in the last five minutes
NumberDiskRequests	Number of disk requests in the last five minutes
PageOutCounter	Number of pages paged out in the last five minutes
NewProcesses	Number of newly spawned processes in the last five minutes
ResponseTime	The interval from the time *httperf* sends out the first byte of request until it receives the first byte of reply

Table 6. **Estimated slope of parameters**

Data Set	Parameter	Slope	95% confidence interval
I	response time	0.027 ms/hour	(0.019, 0.036) ms/hour
	free physical memory	-88.472 KB/hour	(-93.337, -83.607) KB/hour
	used swap space	29.976 KB/hour	(29.290, 30.662) KB/hour
II	response time	0.063 ms/hour	(0.057, 0.068) ms/hour
	free physical memory	15.183 KB/hour	(14.094, 16.271) KB/hour
	used swap space	7.841 KB/hour	(7.658, 8.025) KB/hour

httperf on the client machine, is also included in the model as a measure of performance of the web server.

After collecting the data, it needs to be analyzed to determine if software aging exists, which is indicated by degradation in performance of the web server and/or exhaustion of system resources. The performance of the web server is measured by response time which is the interval from the time a client sends out the first byte of request until it receives the first byte of reply. Figure 10(a) shows the plot of the response time in data set I. To identify the trend, the range of y-axis is magnified (Figure 10(b)). The response time becomes longer with the running time of the experiment. To determine whether the trend is just a fluctuation due to noise or an essential characteristic of the data, a linear regression model is used to fit the time series of the response time. The least squares solution is $r = 15.5655 + 0.027t$, where r is response time in milliseconds, t is the time from the beginning of the experiment. The 95% confidence interval for the slope is (0.019, 0.036) ms/hour. Since the slope is positive, it can be concluded that the performance of the web server is degrading.

Performing the same analysis to the parameters related to system resources, it was found that the available resources are decreasing. Estimated slopes of some of the parameters using linear regression model are listed in Table 6.

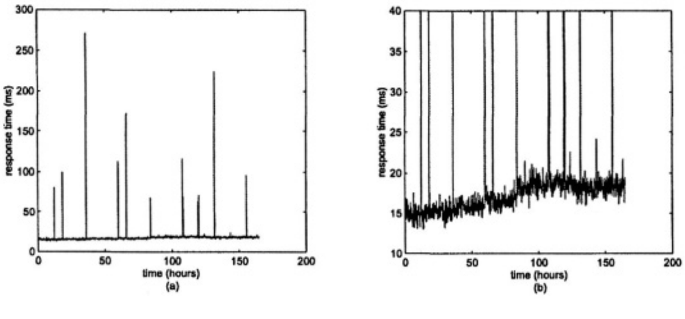

Figure 10. Response time of the web server

The parameters in data set II are used as the modeling objects since the duration of data set II is longer than that of data set I. In this case, there are seven parameters to be analyzed. The analysis can be done using two different approaches: (1) building a univariate model for each of the outputs or, 2) building only one multivariate model with seven outputs. In this case, seven univariate models are built and then combined into a single multivariate model. First, the parameters are determined to determine their characteristics and build an appropriate model with one output and four inputs for each parameter - connection rate, linear trend, periodic series with a period of one week, and periodic series with a period of one day. The autocorrelation function (ACF) and the partial autocorrelation function (PACF) for the output are computed. The ACF and the PACF help us decide the appropriate model for the data [41]. For example, from the ACF and PACF of *used swap space* it can be determined that an autoregressive model of order 1 [AR(1)] is suitable for this data series. Adding the inputs to the AR(1) model, we get the ARX(1) model for used swap space:

$$Y_t = aY_{t-1} + b_1X_t + b_2L_t + b_3W_t + b_4D_t, \tag{8}$$

where Y_t is the used swap space, X_t is the connection rate, L_t is the time step which represents the linear trend, W_t is the weekly periodic series and D_t is the daily periodic series. After observing the ACF and PACF of all the parameters, we find that all of the PACFs cut off at certain lags. So all the multiple input single output (MISO) models are of the ARX type, only with different orders. This gives great convenience in combining them into a multiple input multiple output (MIMO) ARX model which is described later.

In order to combine the MISO ARX models into a MIMO ARX model, we need to choose the order between different outputs. This is done by inspecting the CCF (cross-correlation function) between each pair of the outputs to find out the leading relationship between them. If the CCF between parameter *A* and *B* gets its peak value at a positive lag k, we say that *A* leads *B* by k steps and it might be possible to use *A* to predict *B*. In our analysis, there are 21

CCFs that need to be computed. And in order to reduce the complexity, we only use the CCFs that exhibit obvious leading relationship with lags less than 10 steps. The next step after determination of the orders is to estimate the coefficients of the model by the least squares method. The first half of the data is used to estimate the parameters and the rest of the data is then used to verify the model. Figure 11 shows the two-hour-ahead (24-step) predicted used swap

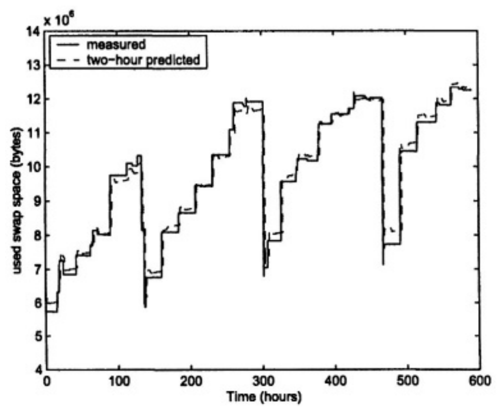

Figure 11. Measured and two-hour ahead predicted used swap space

space which is computed using the established model and the data measured up to two hours before the predicted time point. From the plots, we can see that the predicted values are very close to the measured values.

Explicit link between resource leaks and software aging

In [6] a model is developed to account for the gradual loss of system resources, especially, the memory resource. In a client-server system, for example, every client process issues memory requests at varying points in time. An amount of memory is granted to each new request (when there is enough memory available), held by the requesting process for a period of time, and presumably released back to the system resource reservoir when it is no longer in use. A memory leak occurs when the amount of allocated memory is not fully released. The available memory space is gradually reduced as such resource leaks accumulate over time. As a consequence, a resource request that would have been granted in the leak-less situation may not be granted when the system suffers from memory resource leaks. This model accommodates both the leak-free case and the leak-present case. The model relates system degradation to resource requests, releases or resource holding intervals and memory leaks. These quantities can be monitored and modeled directly from obtainable data measurements [34].

An operating software system is modeled as a continuous time Markov chain (CTMC). The ideal, leak-free case is shown in Figure 12.

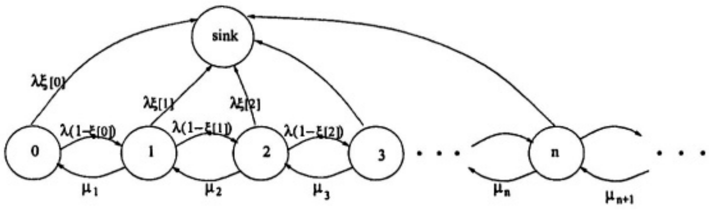

Figure 12. Leak-free model of a system

Denote by M the initial total amount of available memory. The memory unit is application-specific. The system is in workload state k, $k \geq 0$, when there are k independent processes holding a portion of the resource. The total number of states is practically finite. It is assumed that the memory requests are independent of each other and arrive from a Poisson process with rate λ. A request is granted when sufficient memory is available, else the system is considered to have failed. In other words, each incoming request may cause the system to transit to the sink state when it asks for more memory than the available amount. Denote by $\xi[k]$ the conditional probability that the system fails in state k upon the arrival of a new request. The amount of each memory request is modeled as a continuous random variable with the density function $g(x)$. The allocated resource is held for a random period of time, which is dependent upon the processing or service rate and determines the resource release rate. When the holding time per request is exponentially distributed with rate μ, the release rate μ_k at state is equal to $k\,\mu$. Here, the time unit is also application-specific.

Provided with the specification of the leak-free model, one can derive system failure rate $h(t)$ and the system reliability $R(t) = 1-\pi_{\text{sink}}(t) = \sum_k \pi_k(t)$. Conversely, given a specified requirement on system reliability, the model can be used to derive a lower bound on the total amount M of system resource to meet this requirement.

In a system with a leak present, the conditional probability that the system transits to the sink state from state k upon a new request becomes leak dependent and hence time dependent. The memory leak function $\ell(t)$ is related to the system failure via the variable of available resource amount. The variable is bounded from above by the total amount M of the system resource. The failure rate of a leak-present system with the initial amount M of available memory, is denoted by $h(t, M)$.

5. Implementation of a Software Rejuvenation Agent

The first commercial version of a software rejuvenation agent (SRA) for the IBM xSeries line of cluster servers has been implemented with our collaboration [11, 30, 47]. The SRA was designed to monitor consumable resources, estimate the time to exhaustion of those resources, and generate alerts to the management infrastructure when the time to exhaustion is less than a user-defined notification horizon. For Windows operating systems, the SRA acquires data on exhaustible resources by reading the registry performance counters and collecting parameters such as available bytes, committed bytes, non-paged pool, paged pool, handles, threads, semaphores, mutexes, and logical disk utilization. For Linux, the agent accesses the /proc directory structure and collects equivalent parameters such as memory utilization, swap space, file descriptors and inodes. All collected parameters are logged on to disk. They are also stored in memory preparatory to time-to-exhaustion analysis.

In the current version of the SRA, rejuvenation can be based on elapsed time since the last rejuvenation, or on prediction of impending exhaustion. When using Timed Rejuvenation, a user interface is used to schedule and perform rejuvenation at a period specified by the user. It allows the user to select when to rejuvenate different nodes of the cluster, and to select "blackout" times during which no rejuvenation is to be allowed. Predictive Rejuvenation relies on curve-fitting analysis and projection of the utilization of key resources, using recently observed data. The projected data is compared to prespecified upper and lower exhaustion thresholds, within a notification time horizon. The user specifies the notification horizon and the parameters to be monitored (some parameters believed to be highly indicative are always monitored by default), and the agent periodically samples the data and performs the analysis. The prediction algorithm fits several types of curves to the data in the fitting window. These different curve types have been selected for their ability to capture different types of temporal trends. A model-selection criterion is applied to choose the "best" prediction curve, which is then extrapolated to the user-specified horizon. The several parameters that are indicative of resource exhaustion are monitored and extrapolated independently. If any monitored parameter exceeds the specified minimum or maximum value within the horizon, a request to rejuvenate is sent to the management infrastructure. In most cases, it is also possible to identify which process is consuming the preponderance of the resource being exhausted, in order to support selective rejuvenation of just the offending process or a group of processes.

6. Approaches and Methods of Software Rejuvenation

Software rejuvenation can be divided broadly into two approaches as follows:

- **Open-loop approach:** In this approach, rejuvenation is performed without any feedback from the system. Rejuvenation in this case, can be based just on elapsed time (periodic rejuvenation) [29, 16] and/or instantaneous/cumulative number ofjobs on the system [19].

- **Closed-loop approach:** In the closed-loop approach, rejuvenation is performed based on information on the system "health". The system is monitored continuously (in practice, at small deterministic intervals) and data is collected on the operating system resource usage and system activity. This data is then analyzed to estimate time to exhaustion of a resource which may lead to a component or an entire system degradation/crash. This estimation can be based purely on time, and workload-independent [20, 11] or can be based on both time and system workload [46].

  The closed-loop approach can be further classified based on whether the data analysis is done off-line or on-line. Off-line data analysis is done based on system data collected over a period of time (usually weeks or months). The analysis is done to estimate time to rejuvenation. This off-line analysis approach is best suited for systems whose behavior is fairly deterministic. The on-line closed-loop approach, on the other hand, performs on-line analysis of system data collected at deterministic intervals. Another approach to estimate the optimal time to rejuvenation could be based on system failure data [14]. This approach is more suited for off-line data analysis.

This classification of approaches to rejuvenation is shown in Figure 13.

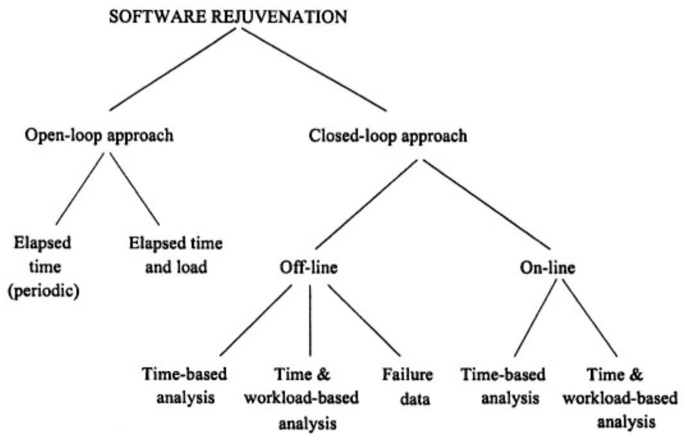

Figure 13. Approaches to software rejuvenation

Rejuvenation is a very general proactive fault management approach and can be performed at different levels - the system level or the application level. An example of a system level rejuvenation is a hardware-reboot. At the application level, rejuvenation is performed by stopping and restarting a particular offending application, process or a group of processes. This is also known as a *partial rejuvenation.* The above rejuvenation approaches when performed on a single node can lead to undesired and often costly downtime. Rejuvenation has been recently extended for cluster systems, in which two or more nodes work together as a single system [11, 47]. In this case, rejuvenation can be performed by causing no or minimal downtime by failing over applications to another spare node.

7. Conclusions

In this paper, we classified software faults based on an extension of Gray's classification and discussed the various techniques to deal with them. Attention was devoted to software rejuvenation, a proactive technique to counteract the phenomenon of software aging. Various analytical models for software aging and to determine optimal times to perform rejuvenation were described. Measurement-based models based on data collected from operating systems were also discussed. The implementation of a software rejuvenation agent in a major commercial server was then briefly described. Finally, various approaches to rejuvenation and rejuvenation granularity were discussed.

In the measurement-based models presented in this paper, only aging due to each individual resource has been captured. In the future, one could improve the algorithm used for aging detection to involve multiple parameters simultaneously, for better prediction capability and reduced false alarms. Dependences between the various system parameters could be studied. The best statistical data analysis method for a given system is also yet to be determined.

Notes

1. Although we use the by-now-established phrase "software aging", it should be clear that no deterioration of the software system per se is implied but rather, the software appears to age due to the gradual depletion of resources [6]. Likewise, "software rejuvenation" actually refers to rejuvenation of the environment in which the software is executing.
2. identical copies

References

[1] E. Adams. Optimizing Preventive Service of the Software Products. *IBM Journal of R&D,* 28(1):2-14, January 1984.

[2] P. E. Amman and J. C. Knight. Data Diversity: An Approach to Software Fault Tolerance. In *Proc. of 17th Int'l. Symposium on Fault Tolerant Computing,* pages 122-126, June 1987.

[3] A. Avritzer and E. J. Weyuker. Monitoring Smoothly Degrading Systems for Increased Dependability. *Empirical Software Eng. Journal,* Vol.2, No.1, pages 59-77, 1997.

[4] A. Avizienis and L. Chen. On the Implementation of N-version Programming for Software Fault Tolerance During Execution. In *Proc. IEEE COMPSAC 77,* pages 149-155, November 1977.

[5] A. Avizienis, J-C. Laprie and B. Randell. Fundamental Concepts of Dependability LAAS Technical Report No. 01-145, LAAS, France, April 2001.

[6] Y. Bao, X. Sun and K. Trivedi. Adaptive Software Rejuvenation: Degradation Models and Rejuvenation Schemes. In *Proc. of The Int'l. Conference on Dependable Systems and Networks, DSN-2003* June 2003.

[7] L. Bernstein. Text of Seminar Delivered by Mr. Bernstein. *University Learning Center,* George Mason University, January 29, 1996.

[8] A. Bobbio, A. Sereno and C. Anglano. Fine grained software degradation models for optimal rejuvenation policies. *Performance Evaluation,* Vol. 46, pp 45-62, 2001.

[9] T. Boyd and P. Dasgupta Premptive Module Replacement Using the Virtualizing Operating System In *Proc. of the Workshop on self-healing, adaptive and self-managed systems, SHAMAN 2002,* New York, NY, June 2002.

[10] K. Cassidy, K. Gross and A. Malekpour. Advanced Pattern Recognition for Detection of Complex Software Aging in Online Transaction Processing Servers. In *Proc. of DSN 2002,* Washington D.C., June 2002.

[11] V. Castelli, R. E. Harper, P. Heidelberger, S. W. Hunter, K. S. Trivedi, K. Vaidyanathan and W. Zeggert. Proactive Management of Software Aging. *IBM Journal of Research & Development,* Vol. 45, No.2, March 2001.

[12] R. Chillarege, S. Biyani, and J. Rosenthal. Measurement of Failure Rate in Widely Distributed Software. In *Proc. of 25th IEEE Int'l. Symposium on Fault Tolerant Computing,* pages 424–433, Pasadena, CA, July 1995.

[13] T. Dohi, K. Goseva–Popstojanova and K. S. Trivedi. Analysis of Software Cost Models with Rejuvenation. In *Proc. of the 5th IEEE International Symposium on High Assurance Systems Engineering, HASE 2000,* Albuquerque, NM, Nov. 2000.

[14] T. Dohi, K. Goseva–Popstojanova and K. S. Trivedi. Statistical Non-Parametric Algorithms to Estimate the Optimal Software Rejuvenation Schedule. *Proc. of the 2000 Pacific Rim International Symposium on Dependable Computing, PRDC 2000,* Los Angeles, CA, Dec. 2000.

[15] C. Fetzer and K. Hostedt Rejuvenation and Failure Detection in Partitionable Systems In *Proc. of the Pacific Rim Int'l. Symposium on Dependable Computing, PRDC 2001,* Seoul, South Korea, December 2001.

[16] S. Garg, A. Puliafito and K. S. Trivedi. Analysis of Software Rejuvenation Using Markov Regenerative Stochastic Petri Net. In *Proc. of the Sixth Int'l. Symposium on Software Reliability Engineering,* pages 180-187, Toulouse, France, October 1995.

[17] S. Garg, Y. Huang, C. Kintala and K. S. Trivedi. Time and Load Based Software Rejuvenation: Policy, Evaluation and Optimality. In *Proc. of the First Fault-Tolerant Symposium,* Madras, India, December 1995.

[18] S. Garg, Y. Huang and C. Kintala, K.S. Trivedi, Minimizing Completion Time of a Program by Checkpointing and Rejuvenation. *Proc. 1996 ACM SIGMETRICS Conference,* Philadelphia, PA, pp. 252-261, May 1996.

[19] S. Garg, A. Puliafito, M. Telek and K. S. Trivedi. Analysis of Preventive Maintenance in Transactions Based Software Systems. *IEEE Trans. on Computers,* pages 96-107, Vol. 47, No. 1, January 1998.

[20] S. Garg, A. van Moorsel, K. Vaidyanathan, K. Trivedi. A Methodology for Detection and Estimation of Software Aging. In *Proc. of 9th Int'l. Symposium on Software Reliability Engineering,* pages 282-292, Paderborn, Germany, November 1998.

[21] S. Garg, Y. Huang, C. M. R. Kintala, K. S. Trivedi and S. Yagnik. Performance and Reliability Evaluation of Passive Replication Scheme s in Application Level Fault Tolerance. In *Proc. of the Fault Tolerant Computing Symp., FTCS 1999,* Madison, WI, pp. 322-329, June 1999.

[22] J. Gray. Why do Computers Stop and What Can be Done About it? In *Proc. of 5th Symposium on Reliability in Distributed Software and Database Systems,* pages 3-12, January 1986.

[23] J. Gray. A Census of Tandem System Availability Between 1985 and 1990. *IEEE Trans. on Reliability,* 39:409-418, October 1990.

[24] J. Gray and D. P. Siewiorek. High-availability Computer Systems. *IEEE Computer,* pages 39–48, September 1991.

[25] J. A. Hartigan. *Clustering Algorithms.* New York:Wiley, 1975.

[26] C. Hirel, B. Tuffin and K. S. Trivedi. SPNP: Stochastic Petri Net Package. Version 6.0. B. R. Haverkort et al. (eds.): TOOLS 2000, Lecture Notes in Computer Science 1786, pp 354-357, Springer-Verlag Heidelberg, 2000.

[27] Y. Hong, D. Chen, L. Li and K.S. Trivedi. Closed Loop Design for Software Rejuvenation In *Proc. of the Workshop on self-healing, adaptive and self-managed systems, SHAMAN 2002,* New York, NY, June 2002.

[28] Y. Huang, P. Jalote, and C. Kintala. *Lecture Notes in Computer Science, Vol. 774,* Two techniques for transient software error recovery, pages 159–170. Springer Verlag, Berlin, 1994.

[29] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. Software Rejuvenation: Analysis, Module and Applications. In *Proc. of 25th Symposium on Fault Tolerant Computing, FTCS-25,* pages 381–390, Pasadena, California, June 1995.

[30] IBM Netfinity Director Software Rejuvenation - White Paper. IBM Corp., Research Triangle Park, NC, Jan 2001.

[31] P. Jalote, Y. Huang, and C. Kintala. A Framework for Understanding and Handling Transient Software Failures. In *Proc. 2nd ISSAT Int'l. Conf. on Reliability and Quality in Design,* Orlando, FL, 1995.

[32] J. C. Knight and N. G. Leveson. An Experimental Evaluation of the Assumption of Independence in Multiversion Programming *Software Engineering Journal,* pages 96-109, Vol. 12, No. 1, 1986.

[33] I. Lee and R. K. Iyer. Software Dependability in the Tandem GUARDIAN System. *IEEE Trans. on Software Engineering,* pages 455-467, Vol. 21, No. 5, May 1995.

[34] L. Li, K. Vaidyanathan and K. S. Trivedi. An Approach to Estimation of Software Aging in a Web Server. In *Proc. of the Int'l. Symp. on Empirical Software Engineering, ISESE 2002,* Nara, Japan, October 2002.

[35] Y. Liu, Y. Ma, J.J. Han, H. Levendel and K.S. Trivedi. Modeling and Analysis of Software Rejuvenation in Cable Modem Termination System. In *Proc. of the Int'l. Symp. on Software Reliability Engineering, ISSRE 2002,* Annapolis, MD, November 2002.

[36] E. Marshall. Fatal Error: How Patriot Overlooked a Scud. *Science,* page 1347, March 13, 1992.

[37] D. Mosberger and T. Jin. Httperf - A Tool for Measuring Web Server Performance *In First Workshop on Internet Server Performance, WISP,* Madison, WI, pp.59-67, June 1998.

[38] A. Pfening, S. Garg, A. Puliafito, M. Telek and K. S. Trivedi. Optimal Rejuvenation for Tolerating Soft Failures. *Performance Evaluation,* 27 & 28, pages 491-506, October 1996.

[39] S. M. Ross. *Stochastic Processes.* John Wiley & Sons, New York, 1983.

[40] R. A. Sahner, K. S. Trivedi, A. Puliafito. *Performance and Reliability Analysis of Computer Systems - An Example-Based Approach Using the SHARPE Software Package.* Kluwer Academic Publishers, Norwell, MA, 1996.

[41] R. H. Shumway and D. S. Stoffer. *Time Series Analysis and Its Applications,* Springer-Verlag, New York, 2000.

[42] M. Sullivan and R. Chillarege. Software Defects and Their Impact on System Availability - A Study of Field Failures in Operating Systems. In *Proc. 21st IEEE Int'l. Symposium on Fault-Tolerant Computing,* pages 2–9, 1991.

[43] A. T. Tai, S. N. Chau, L. Alkalaj and H. Hecht. On-Board Preventive Maintenance: Analysis of Effectiveness and Optimal Duty Period. In *3rd Int'l. Workshop on Object Oriented Real-time Dependable Systems,* Newport Beach, CA, February 1997.

[44] K. S. Trivedi, J. Muppala, S. Woolet and B. R. Haverkort. Composite Performance and Dependability Analysis. *Performance Evaluation,* Vol. 14, nos. 3-4, pp. 197-216, February 1992.

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

[46] K. Vaidyanathan and K. S. Trivedi. A Measurement-Based Model for Estimation of Resource Exhaustion in Operational Software Systems. In *Proc. of the Tenth IEEE Int'l. Symposium on Software Reliability Engineering,* pages 84-93, Boca Raton, Florida, November 1999.

[47] K. Vaidyanathan, R. E. Harper, S. W. Hunter, K. S. Trivedi. Analysis and Implementation of Software Rejuvenation in Cluster Systems. In *Proc. of the Joint Int'l. Conference on Measurement and Modeling of Computer Systems, ACM SIGMETRICS 2001/Performance 2001,* Cambridge, MA, June 2001.

[48] http://www.microsoft.com/technet/prodtechnol/windows2000serv/technologies/iis/default.mspx

[49] http://www.apache.org