Performance Assurance via Software Rejuvenation: Monitoring, Statistics and Algorithms

Alberto Avritzer[*], Andre Bondi[*], Michael Grottke[†], Kishor S. Trivedi[†], and Elaine J. Weyuker[‡]

Abstract

We present three algorithms for detecting the need for software rejuvenation by monitoring the changing values of a customer-affecting performance metric, such as response time. Applying these algorithms can improve the values of this customer-affecting metric by triggering rejuvenation before performance degradation becomes severe. The algorithms differ in the way they gather and use sample values to arrive at a rejuvenation decision. Their effectiveness is evaluated for different sets of control parameters, including sample size, using simulation. The results show that applying the algorithms with suitable choices of control parameters can significantly improve system performance as measured by the response time.

1 Introduction

The increasing complexity of large industrial systems has created a need for sophisticated software monitoring to ensure performance and availability. We have introduced in [1] and [2] a new software rejuvenation approach that can ensure stable performance of degradable systems. Our approach was designed to distinguish between performance degradation that occurs as a result of burstiness in the arrival process and software degradation that occurs as a result of software aging. This is done by frequently monitoring the metric of greatest interest to the customer and triggering software rejuvenation whenever severe degradation is detected. We will refer to this metric as the customer-affecting metric. Software aging has been observed in complex systems and has been described extensively in the literature [3, 8, 9, 15]. We will describe a case study using a large complex industrial e-commerce system for which a severe fault that led to significant performance degradation eluded detection for several months. This happened because the metric of greatest interest to the customer, response time, was not being monitored, while metrics of less interest including CPU utilization and memory usage were being tracked. More detail about the fault can be found in [4].

Motivated by this experience, we developed software rejuvenation algorithms to track the customer-affecting metric, in this case response time (RT). In this paper we propose three new algorithms for software rejuvenation that use different sampling approaches to determine whether the RT has severly degraded. Our first algorithm, called *static software rejuvenation with averaging*, is a variant of the *static software rejuvenation* algorithm introduced in [1]. In this new algorithm, the observed values of the RT are averaged over a fixed sample size. Our second algorithm, called *static software rejuvenation with averaging and sample acceleration*, reduces the sample size requirement when a degradation in performance is detected. For this algorithm, the time to compute the average of the RT is proportional to the sample size and so the overall time required to trigger a software rejuvenation is reduced when a significant performance degradation is detected.

Our third algorithm is a direct application of the central limit theorem. Whenever the average of the samples exceeds the target quantile of the normal distribution, this algorithm will trigger a software rejuvenation. In this paper we evaluate the domain of applicability of the three algorithms by simulating their performance using the large e-commerce system mentioned above.

The outline of our paper is as follows. Section 2 provides a survey of related work. In Section 3 we present our model of software rejuvenation which has been designed to identify performance degradation using observations of the RT. In Section 4 we present three new software rejuvenation algorithms and the analytical motivation for our algorithms. Section 5 presents simulation results for the e-commerce system, while Section 6 contains our conclusions and suggestions for future research.

[*]Alberto Avritzer and Andre Bondi are with Siemens Corporate Research, Princeton, NJ 08540. E-mail: {alberto.avritzer, andre.bondi}@siemens.com

[†]Michael Grottke and Kishor Trivedi are with Duke University, Department of Electrical and Computer Engineering, Durham, NC 27708-0291. E-mail: {grottke,kst}@ee.duke.edu. This work was supported by a fellowship within the Postdoc Program of the German Academic Exchange Service.

[‡]Elaine Weyuker is with AT&T Labs - Research, 180 Park Avenue, Florham Park, NJ 07932. E-mail: weyuker@research.att.com

2 Related work

Avritzer and Weyuker [3] introduced an approach to restoring degraded software to full capacity that was tailored particularly for telecommunications or other similar systems that are characterized by predictably periodic traffic. It is common for telecommunications operating companies to collect detailed traffic pattern data, thereby allowing them to identify system degradations and the subsequent restoration of their systems to full capacity by executing software procedures that will free allocated memory, release database locks, and reinitialize other operating system tables.

Bobbio et al. [5] presented a method for the quantitative analysis of software rejuvenation policies. Their approach assumed that it was possible to quantify system degradation by monitoring a metric that was correlated with it. They defined a maximum degradation threshold level and introduced two rejuvenation policies based on this threshold. A risk-based approach was used for their first policy, defining a confidence level for the metric. Rejuvenation was performed with a probability proportional to this confidence level. They also introduced a deterministic policy for which rejuvenation was performed as soon as the threshold level was reached. The approach we study in this paper is closely related to this second policy but uses multiple threshold levels so that we can distinguish between bursts of arrivals and soft failures. When a *soft failure* occurs, the system remains operational, but in a degraded mode with the available system capacity greatly reduced.

In [1], a software rejuvenation approach for single server systems that tracks the value of the RT to determine the best times to perform software rejuvenation was introduced. In [2], it was extended by evaluating the applicability of the algorithms to clusters of hosts. The three rejuvenation algorithms that we consider in this paper base the decision to trigger rejuvenation on averages of several observations, rather than on the value of the RT itself.

A comparison of analytical models using Markov regenerative processes and measurement approaches using time series analysis, trend detection and estimation for software rejuvenation was presented in [15]. This paper provides motivation for the development of practical policies based on actual measurements.

Other papers of interest include [11], in which service failures were identified by monitoring the RT, and [6], which described the methodology used by IBM Director software for proactive software rejuvenation using statistical estimation of resource exhaustion.

3 The system and the simulation model

The subject of our case studies is a multi-tier distributed e-commerce system written in Java. It consists of 16 CPUs with a Java virtual machine (JVM) heap size of 3 GB. The maximum acceptable RT is 10 seconds, and the system must be able to handle up to 1.6 transactions/second. The two primary factors that were found to impact performance were the variability that occurred when garbage collections took place, and kernel overhead.

The system model used to generate the experimental results is a slightly modified version of the one used in [4].

1. Whenever a thread arrives at the JVM, a new thread arrival is scheduled with an exponentially distributed inter-arrival time (with arrival rate λ), and the number of active threads is incremented by one.

2. The thread queues for a CPU.

3. The CPU processing time of the thread is sampled from an exponential distribution with service rate $\mu = 0.2$ transactions/second.

4. If the number of threads executing in parallel is greater than the specified threshold of 50 threads, an overhead is incurred. In order to account for this, the processing time of the thread is multiplied by a factor of 2.0.

5. As soon as one of the 16 CPUs is obtained, the thread attempts to allocate 10 MB of memory.

6. If the remaining memory heap size is less than the memory threshold of 100 MB, a full garbage collection event is scheduled and all running threads are delayed by 60 seconds, the amount of time needed to perform a full garbage collection on a 3 GB Heap.

7. When a thread completes service, the total RT of the related job, consisting of the waiting time plus the processing time, is computed.

8. Based on the observed RTs, an algorithm decides whether software rejuvenation should be carried out. We will discuss several versions of a rejuvenation algorithm in the next section. When the system is rejuvenated, all threads in execution are terminated and all resources held by threads are released. This includes both the JVM memory heap and busy CPUs.

The goal of our software rejuvenation algorithms is to monitor the RT and do a capacity restoration whenever the RT has severely deteriorated. During rejuvenation, all CPU and memory queues are cleared. The cost is defined to be the percentage of transactions lost during the rejuvenation event.

We assume that the RT can be sampled frequently and that the average and standard deviation of the RT when the system is operating without performance degradation, serves as the basis for the system performance requirement.

4 The rejuvenation algorithms

4.1 Analytical motivation

Rejuvenation must be carried out often enough to avoid severe performance slowdowns, yet not unnecessarily, since rejuvenation incurs costs such as lost transactions. The algorithm that triggers rejuvenation must depend on recently observed values of the RT to decide whether the system behavior has deteriorated sufficiently from normal behavior to justify the costs of rejuvenation. While this idea conforms to our intuition, we have to make precise what we mean by "normal" system behavior and when the observed behavior is "significantly" different from normal. Abstracting from the garbage collections and the kernel overhead, which are the two characteristics that were responsible for the performance slowdowns, the simulation model described in the last section becomes a first-come, first-served (FCFS) queuing model with $c = 16$ parallel servers in which both the interarrival times and the service times follow an exponential distribution. For such an $M/M/c$ queuing model, the number of jobs in the system can be depicted by the Markovian state diagram shown in Fig. 1. λ and μ represent the arrival rate and the service rate, respectively. When the traffic intensity, $\rho = \frac{\lambda}{c\mu}$, is less than 1, the system is stable and will eventually reach a steady state; otherwise the number of jobs in the system tends to increase without bound. Let the random variable X_i denote the RT of a job randomly sampled from an FCFS-$M/M/c$ system in the steady state. Gross and Harris [7, p. 73] derive the cumulative distribution function of X_i, as

$$F_{X_i}(x) = W_c \cdot (1 - \exp(-\mu \cdot x)) + (1 - W_c) \times \left[\frac{c\mu - \lambda}{(c-1)\mu - \lambda} \cdot (1 - \exp(-\mu \cdot x)) - \frac{\mu}{(c-1)\mu - \lambda} \cdot (1 - \exp(-(c\mu - \lambda) \cdot x)) \right], \quad (1)$$

where

$$W_c = 1 - \left[\frac{(c\rho)^c}{c!} \cdot \frac{1}{1-\rho} \right] \cdot \left[\sum_{k=0}^{c-1} \frac{(c\rho)^k}{k!} + \frac{(c\rho)^c}{c!} \cdot \frac{1}{1-\rho} \right]^{-1}$$

represents the steady-state probability that fewer than c jobs are present in the system.

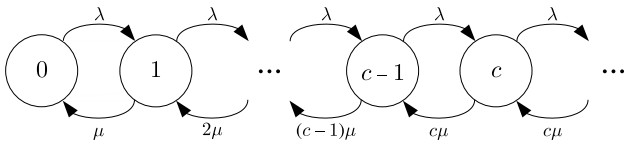

Figure 1. *State diagram of the $M/M/c$ queuing system*

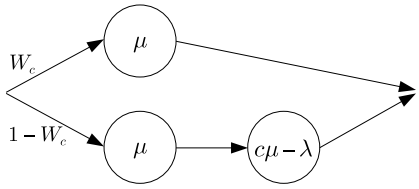

Figure 2. *X_i as a phase-type distribution*

Both the expected value and the variance of the RT can easily be derived from equation (1) by noting that the RT distribution is a mixture of an exponential distribution and a two-stage hypoexponential distribution, and hence is a phase-type distribution [10, pp. 41–80] representable by a parallel and serial combination of exponential distributions. In the structure shown in Fig. 2, each circle stands for an exponential distribution with a hazard rate given by the expression noted in the circle.

Applying the rules for calculating the moments of hypoexponential and hyperexponential distributions [14, pp. 223-224], we obtain the expectation of X_i,

$$\begin{aligned} \mu_X &:= E(X_i) = \frac{W_c}{\mu} + (1 - W_c) \cdot \left[\frac{1}{\mu} + \frac{1}{c\mu - \lambda} \right] \\ &= \frac{1}{\mu} + \frac{1 - W_c}{c\mu - \lambda}, \end{aligned} \quad (2)$$

as well as the variance

$$\sigma_X^2 := Var(X_i) = \frac{1}{\mu^2} + \frac{1 - W_c^2}{(c\mu - \lambda)^2}. \quad (3)$$

For our basic $M/M/16$ queuing system with a service rate of 0.2 transactions/second, when arrival rates are below 1 transaction/second, both the mean and the standard deviation of the RT remain constant at 5. The reason for this behavior is that under these low loads, the threads rarely have to queue for a CPU. Therefore, the RT basically follows an exponential distribution with service rate $\mu = 0.2$ transactions/second. For higher loads, the expected value and the standard deviation start to diverge from their baseline value of 5.

One approach might be to use the upper quantiles of the cumulative distribution function (1) for monitoring the system behavior and to trigger rejuvenation when the observed RT x_i exceeds some pre-determined quantile. However, such an approach would not be robust for *short-term* deviations in the system behavior, and we need to avoid costly rejuvenations under such circumstances.

The static and the dynamic rejuvenation algorithms introduced in [1] and [2] try to comply with this requirement by triggering rejuvenation only after repeated occurrences of large RTs.

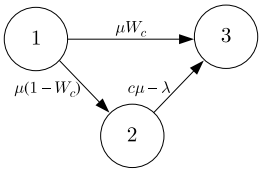

Figure 3. *Response time X_i as time to absorption in a continuous time Markov chain*

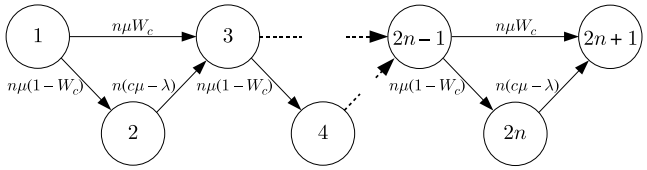

Figure 4. *Average response time $\bar{X}_n = \frac{1}{n}\sum_{i=1}^{n} X_i$ as time to absorption in a continuous-time Markov chain*

Another way of coping with short-term increases in the RT is to base the rejuvenation decision on averages calculated from several measured values of the RT rather than on the RT itself. This allows small numbers of subsequently observed large values to be "smoothed out". On the one hand, this idea can be included into the rejuvenation algorithms presented in [1] and [2]. On the other hand, it allows us to make use of the central limit theorem, which in its simplest form states that the average $\bar{X}_n = \frac{1}{n}\sum_{i=1}^{n} X_i$ of n independent and identically distributed random variables X_i approaches a normal distribution as $n \to \infty$. Therefore, if n is sufficiently large, we can approximate the distribution of $\bar{X}_n$ by a normal distribution, regardless X_i's true distribution. We can then base the decision of whether to trigger rejuvenation on the quantiles of the normal distribution.

How large must n be for the normal approximation to be "good enough"? In general, calculating the convolution of n random variables is difficult, hampering an examination of how quickly $\bar{X}_n$ approaches the normal distribution. However, there is a simple approach for deriving the distribution of $\bar{X}_n$, provided the distribution of X_i can be represented by the distribution of the time to absorption in a Markov chain [14]. This is the case for the RT in the $M/M/c$ queuing system. >From Fig. 2 we can derive the continuous-time Markov chain shown in Fig. 3. Assuming that the initial probability for being in state 1 is 100%, the time to reach absorbing state 3 follows the same distribution as the RT X_i.

To find a representation of $\bar{X}_n$ as a time to absorption, we begin by multiplying all transition rates in Fig. 3 by n and get a Markov chain in which the distribution of the time to absorption is identical to the distribution of each individual $\frac{X_i}{n}$. This is true because all transition times in a Markov chain are exponential and because dividing an exponentially distributed random variable with hazard rate z by some constant r yields an exponentially distributed random variable with hazard rate $r \cdot z$. Therefore, $\bar{X}_n = \frac{1}{n}\sum_{i=1}^{n} X_i = \sum_{i=1}^{n} \frac{X_i}{n}$ can be represented as the time to absorption in the Markov chain derived by concatenating n such Markov chains, fusing state 3 of the n^{th} sub-chain and state 1 of the $(n+1)^{st}$ sub-chain into state $2n+1$ of the resulting Markov chain. This is shown in Fig. 4.

Using the SHARPE [12] tool, we are able to derive the cumulative distribution function of the average RT, $F_{\bar{X}_n}(x)$. It is also possible to calculate the exact probability density function $f_{\bar{X}_n}(x)$ via the relationship

$$\begin{aligned} f_{\bar{X}_n}(x) &= \frac{dp_{2n+1}(x)}{dx} \\ &= p_{2n-1}(x) \cdot n\mu W_c + p_{2n}(x) \cdot n(c\mu - \lambda), \end{aligned} \tag{4}$$

where $p_i(x)$ is the probability that the Markov chain of Fig. 4 is in state i at time x.

For $c = 16$ servers, $\lambda = 1.6$ transactions/second and $\mu = 0.2$ transactions/second, Fig. 5 shows how this probability density function of the sample average approaches a normal density as n increases. For each value of n, the dashed curve is the probability density function of a normal distribution with expected value equal to $\mu_{\bar{X}_n} = \mu_X$ and variance equal to $\sigma^2_{\bar{X}_n} = \sigma^2_X/n$. While the full asymptotic properties only hold for an infinite sample size, we can see that the density of the sample average can be reasonably approximated by a normal density for sample sizes as low as 30 or even 15.

Since we want to detect "unnaturally" long RTs, we are especially interested in the upper tails of the distributions. For example, an algorithm could trigger rejuvenation whenever the sample mean exceeds the 97.5% quantile of the approximating normal distribution. Ideally, the false alarm probability of such a decision rule would be 2.5%. In fact, the probability mass that $f_{\bar{X}_n}(x)$, equation (4), allocates to the right of the respective quantiles of the normal distribution amounts to 3.69% for $n = 15$ and 3.37% for $n = 30$. While the false error proabilities are somewhat inflated, the approximation seems to be good enough for our purposes.

In the simplest form of the central limit theorem, the sampled values are assumed to be independent. Clearly, the RTs observed in a queuing system can be highly correlated. If the RT for a job is large because it had to queue to be served, the same is more likely to be true for the jobs following it, especially as the load increases. When the load is low enough so that no queuing occurs, then the RTs follow independent and identical exponential distributions. As mentioned above, for an $M/M/16$ queuing system with an arrival rate of $\mu = 0.2$ transactions/second, queueing starts to have an effect for arrival rates of about 1 transaction/second.

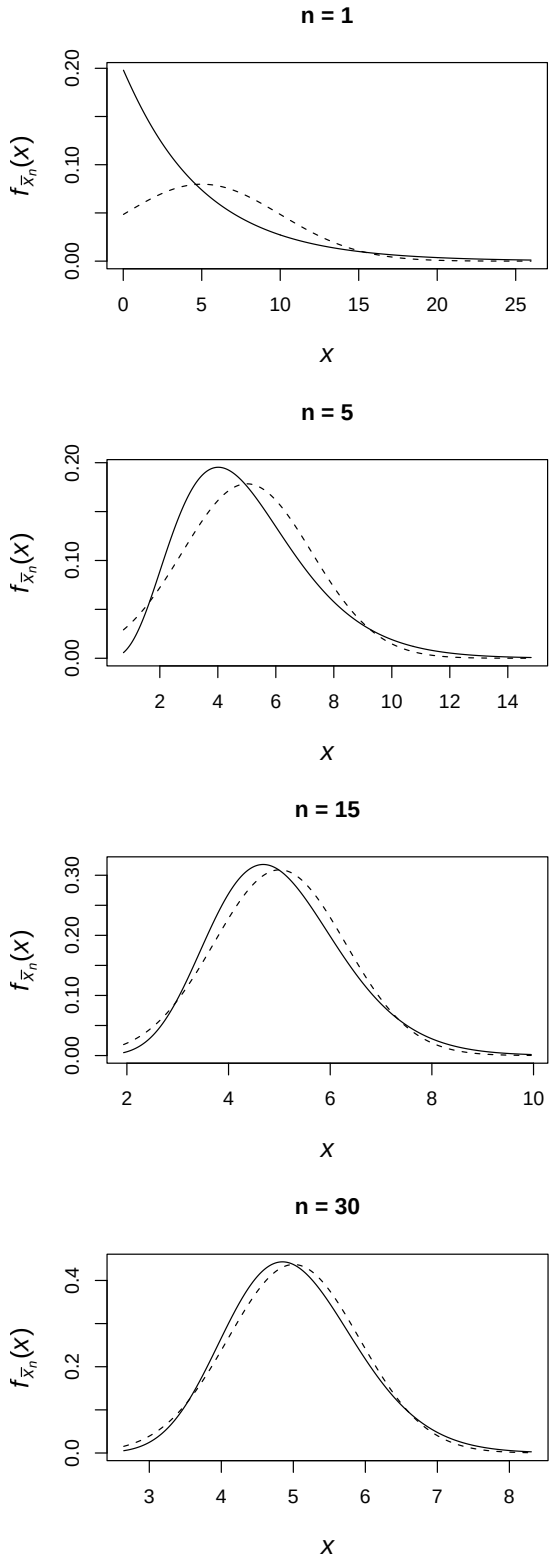

Figure 5. *Probability density function of average response time $\bar{X}_n$ for n=1, 5, 15, 30 and corresponding approximating normal densities $f_N(x; \mu_{\bar{X}_n}, \sigma^2_{\bar{X}_n})$; $\lambda = 1.6, \mu = 0.2$*

For an arrival rate of 1.6 transactions/second (the maximum arrival rate of interest) is the correlation too strong for the central limit theorem to be applicable? In order to investigate this question, we simulated the RTs for this configuration of the $M/M/16$ model. We obtained the simulation model from the one described in Section 3 by abstracting from the kernel overhead (step 4), the memory leaks (steps 5 and 6) and the rejuvenation (step 8). Five independent replications of $100,000$ transactions were run. For each one, we estimated the first-order autocorrelation coefficient as [13, p. 26]

$$\hat{\gamma} = \frac{\sum_{i=10,001}^{99,999}(x_{i+1} - \bar{x})(x_i - \bar{x})}{\sum_{i=10,001}^{100,000}(x_i - \bar{x})^2},$$

where $\bar{x} = \frac{1}{90,000}\sum_{i=10,001}^{100,000} x_i$. Note that the first $10,000$ transactions were omitted in order to eliminate transient effects. The autocorrelation coefficient is considered to be significantly different from zero at a confidence level of 95% if its absolute value exceeds $1.96/\sqrt{90,000}$. This was the case for only one of our five replications. As expected, at high loads subsequent RTs can show some dependence; however, even at the maximum load of interest the first-order correlation seems to play a minor role. In our case, it should therefore be possible to use a rejuvenation algorithm employing the results of the central limit theorem as an approximation, regardless of the actual system load.

4.2 Three algorithms

We now propose three new algorithms for detecting a significant and lasting deterioration in the RT. We assume that the service level agreement specifies the mean μ_X and the standard deviation σ_X of the RT under normal system behavior. Our algorithms further assume that small values of the observed metric are better than large ones, which is true for RT. In [1] and [2], we have introduced the so-called *static software rejuvenation algorithm*. This algorithm tracks the deterioration of the RT based on a bucket metaphor. Beginning with the first of the K buckets, for each sampled value $x_i > \mu_X$, one ball is added to the bucket, while one ball is removed if $x_i \leq \mu_X$. The bucket is keeping a count of the difference between the number of times the expected value μ_X is exceeded and the number of times the sampled value is less than or equal to μ_X. When the number of balls in the bucket reaches its allowed depth D, the bucket "overflows", and we move on to the second bucket, for which the target value used for adding/removing balls is $\mu_X + \sigma_X$. We use the index N $(= 0, 1, 2, \ldots, K-1)$ as a pointer to the current bucket. Since one standard deviation is added to the target whenever moving to the next bucket, the target value for bucket N is $\mu_X + N \cdot \sigma_X$. When the last one of the K buckets overflows, the rejuvenation routine is executed. The algorithm moves to the previously filled bucket, $N-1$, when the current bucket $N > 0$ "underflows", i.e., the bucket is empty and $x_i \leq \mu_X + N \cdot \sigma_X$. In

this version of the algorithm, the bucket depth D is constant for all buckets and so the algorithm is said to be *static*. Activation of the rejuvenation routine in the presence of transient degradation is prevented by the bucket depth D and the total number of buckets K. The minimum delay before a degradation can be affirmed is at least $D \cdot K$ observations. The value of K also determines how much the distribution of the RT must have shifted before a rejuvenation should be carried out using the implicit assumption that a shift by $K-1$ standard deviations σ_X is significant.

In the last section, we saw that one way to cope with short-term changes in the distribution of the RT is to calculate the sampling average $\bar{x}_n$ from n individual observations. We now introduce the static rejuvenation algorithm with averaging (SRAA), shown in Fig. 6, which tracks the development of the sampling average $\bar{x}_n$ of the RT. The target values used are still of the form $\mu_X + N \cdot \sigma_X$; this ensures that regardless of the sample size n rejuvenation is triggered as soon as the algorithm has detected evidence for a shift of the distribution of the RT X_i by $K-1$ standard deviations σ_X.

The next two algorithms, however, are based on a different paradigm. They do not try to "verify" that the distribution of the RT has shifted by a specific amount. Instead they focus on detecting whether the distribution has moved at all, trying to falsify the hypothesis that there has been no right-shift. If this hypothesis can be rejected based on the observed data, then software rejuvenation is carried out. As a consequence of the changed paradigm, the following algorithms employ the standard deviation of the sampling average, $\sigma_{\bar{X}_n} = \sigma_X/\sqrt{n}$, to determine whether the average calculated from the last n observations deviates significantly from the expected behavior. Therefore, the target values used are of the form $\mu_X + N\sigma_X/\sqrt{n}$.

```
function SRAA (d, K, n, μ_X, σ_X){
    u := 0; d := 0; N := 0;
    while (n additional observations available){
        u := u + 1;
        x̄_u := (1/n) Σ_{t=(u-1)·n+1}^{un} x_t;
        if (x̄_u > μ_X + Nσ_X) then
            { d := d + 1; } else { d := d − 1; }
        if (d > D) then
            { d := 0; N := N + 1; }
        if ((d < 0) AND (N > 0)) then
            { d := D; N := N − 1; }
        if ((d < 0) AND (N == 0)) then
            { d := 0; }
        if (N == K) then
            { rejuvenation_routine();
            d := 0; N := 0;}}}
```

Figure 6. *Pseudo-code for the static rejuvenation algorithm with averaging (SRAA)*

```
function SARAA (d, K, n_orig, μ_X, σ_X){
    u := 0; n := n_orig; d := 0; N := 0;
    while (n additional observations available){
        u := u + 1;
        x̄_u := (1/n) Σ_{t=(u-1)·n+1}^{un} x_t;
        if (x̄_u > μ_X + Nσ_X/√n) then
            { d := d + 1; } else { d := d − 1; }
        if (d > D) then
            { d := 0; N := N + 1;
            n := floor(1 + (n_orig − 1) · (1 − N/K)); }
        if ((d < 0) AND (N > 0)) then
            { d := D; N := N − 1;
            n := floor(1 + (n_orig − 1) · (1 − N/K)); }
        if ((d < 0) AND (N == 0)) then
            { d := 0; }
        if (N == K) then
            { rejuvenation_routine();
            d := 0; N := 0; n := n_orig; }}}
```

Figure 7. *Pseudo-code for the sampling acceleration rejuvenation algorithm with averaging (SARAA)*

Our second algorithm, the sampling acceleration rejuvenation algorithm with averaging (SARAA), collects a series of n sample values of the RT and computes $\bar{x}_u$ by taking the average. It then uses the bucket metaphor to keep a count of the difference between the number of times the target value ($\mu_X + N\sigma_X/\sqrt{n}$) of the current bucket has been exceeded, and the number of times the sampled value is less than or equal to this target value. N is again the pointer to the current bucket, K the total number of buckets used for the algorithm, and D is the depth of each bucket. When degradation is detected, sampling is accelerated by requiring fewer sample values to trigger a transition from one bucket to the next. The number of sample values used for a bucket is computed when the previous bucket overflows. The computation ensures that when degradation is detected, fewer samples are required to trigger a transition to the next bucket. SARAA uses linear sampling acceleration with rate $-N/K$. The `floor(y)` operator, which returns the largest integer less than or equal to `y`, guarantees that the calculation of the current sampling size always results in an integer value. The algorithm is initialized with n_{orig}, the sample size used for the first bucket. In Section 5, we study the performance of SARAA for n_{orig} values of 5 and 10.

The two algorithms described so far smooth short-term deviations of the RT by using multiple buckets, the specification of a bucket depth, and by taking averages of successive observations. Typically, the sampling sizes used for these algorithms are too small for employing the normal approximation following from the central limit theorem, and the two algorithms do not rely on this approximation. However, they do benefit from symmetry characteristics of the probability density function of the sample average $\bar{X}_n$ of

```
function CLTA (n, μ_X, σ_X, N){
    u := 0;
    while (n additional observations available){
        u := u + 1;
        x̄_u := (1/n) Σ_{t=(u−1)·n+1}^{un} x_t;
        if (x̄_u > μ_X + Nσ_X/√n) then
            { rejuvenation_routine(); }}}
```

Figure 8. *Pseudo-code for the central limit theorem-based rejuvenation algorithm (CLTA)*

the RT. As shown in Fig. 5, even for a sample size as small as five, the probability density function of $\bar{X}_n$ is much more symmetric than that of the RT itself. Therefore, the calculation of averages does help the two algorithms in a second way apart from the smoothing of short-term deviations: It increases the probability of values larger than μ_X and decreases the probability of values smaller than μ_X. As a consequence, for a larger sample size, both the SRAA and the SARAA are able to detect right shifts of the distribution faster.

Our third algorithm, the central limit theorem-based algorithm (CLTA), directly applies the theorem. It requires a larger sample size n, but it only waits for one significantly large sampling average $\bar{x}_n$. This means that the number of buckets as well as the bucket depth are both implicitly set to one. The factor N used in the calculation of the target value $\mu_X + N\sigma_X/\sqrt{n}$, which the sampling average has to exceed in order to cause a rejuvenation, is a quantile of the standard normal distribution chosen according to the acceptable probability of a false alarm. However, we have to take into account our earlier finding that due to the approximation the real error probabilities are larger than one might think. For example, if N is set to 1.96, the 97.5% quantile of the standard normal distribution, then the false alarm probability of decisions based on averages of $n = 30$ observations is 3.37%.

5 Empirical results

We have evaluated the three algorithms, SRAA, SARAA, and CLTA, by running the simulation model described in Section 3, which represents the e-commerce system discussed in [1]. We ran each simulation for 500,000 transactions divided into five replications of 100,000 transactions each. In each simulation experiment, the constant values of $\mu_X = \sigma_X = 5$ were used. For an $M/M/16$ queuing system with a service rate of $\mu = 0.2$ transactions/second and an arrival rate of up to 1.6 transactions/second, the mean and the standard deviation of the RT distribution are close to these values, as was seen in Section 4.1. Average RT and fractions of transactions lost are shown as functions of the offered load ($\frac{\lambda}{\mu}$) expressed in terms of the number of CPUs. In each figure we kept the value of the product of sample size, n, number of buckets, K, and bucket depth, D, constant. Throughout the paper, we refer to this product as $n \cdot K \cdot D$.

In each experiment described below, our goal is to evaluate the individual impact of each parameter on the algorithm performance as assessed in terms of average RT at high loads and the amount of transaction loss at low loads.

5.1 SRAA, $n \cdot K \cdot D$ = 15

Fig. 9 presents RT results for SRAA, with $n \cdot K \cdot D$ = 15. For this evaluation, we performed experiments setting $(n, K, D) = (1, 3, 5), (1, 5, 3), (3, 1, 5), (3, 5, 1), (5, 1, 3), (5, 3, 1), (15, 1, 1)$. We see a clear dichotomy of RTs in Fig. 9 with $(n, K, D) = (5, 1, 3), (3, 1, 5), (15, 1, 1)$ providing better average RTs than $(n, K, D) = (5, 3, 1), (1, 5, 3), (1, 3, 5)$. However, when we examine Fig. 10, we see that the improvement in performance achieved by $(n, K, D) = (5, 1, 3), (3, 1, 5), (15, 1, 1)$ is obtained at the cost of a larger fraction of transaction loss at low loads. Therefore, we conclude that for $n \cdot K \cdot D$ = 15, when only one bucket is used ($K = 1$) we observe better RTs over the entire range than for any other value of K, but incur a higher average transaction loss at low loads, and a lower average transaction loss at high loads. This is an interesting observation since the introduction of multiple buckets in SRAA was intended to distinguish between performance degradation that occurs as a result of bursts of arrivals, and degradation that occurs as a result of software aging. Here we observe that when several buckets were used, SRAA tolerates bursts of arrivals at low loads with negligible transaction loss as expected. However, this comes at the cost of a considerably higher transaction loss at high loads.

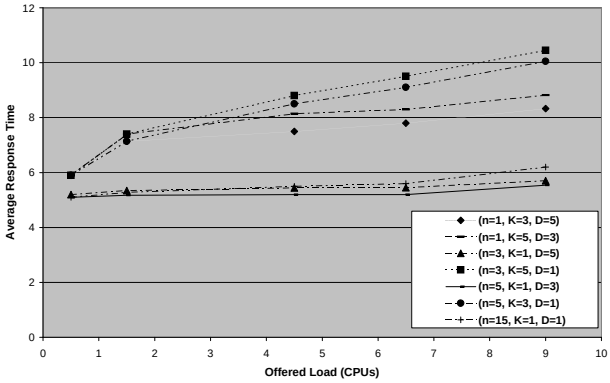

Figure 9. Response time results, SRAA, $n \cdot K \cdot D$ = 15

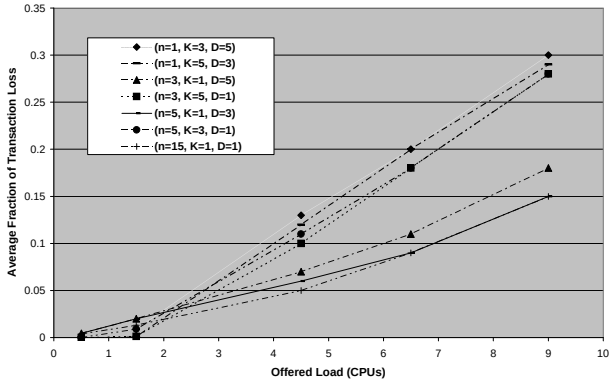

Figure 10. Fraction of transaction loss results, SRAA, $n \cdot K \cdot D$ = 15

Our next experiments are designed to investigate the impact of doubling the value of each parameter on the performance of the SRAA algorithm. In Sections 5.2, 5.3, and 5.4, we evaluate the performance of SRAA for parameters generated by doubling the sample size, the bucket depth and the number of buckets, while keeping $n \cdot K \cdot D$ = 30.

5.2 SRAA, sample size doubled

Fig. 11 presents RT results for SRAA, with $n \cdot K \cdot D$ = 30. We set $(n, K, D) = (2, 3, 5)$, $(2, 5, 3)$, $(6, 1, 5)$, $(6, 5, 1)$, $(10, 1, 3)$, $(10, 3, 1)$, $(30, 1, 1)$, by doubling the sample size component of the experiment settings from Section 5.1. The goal of this experiment is to assess the impact of the sample size parameter on the algorithm performance. We notice from Fig. 11 that doubling the values of sample size has a negative impact on the RT. For example, for a load of 9.0 CPUs and $(n, K, D) = (15, 1, 1)$, the average RT for SRAA is 6.2 seconds, while for $(n, K, D) = (30, 1, 1)$, the average RT for SRAA is 9.9 seconds.

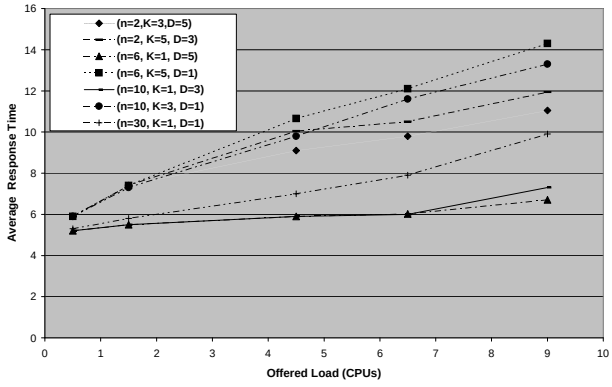

Figure 11. Response time results, SRAA, $n \cdot K \cdot D$ = 30, impact of sample size doubling

For the same load of 9.0 CPUS, for $(n, K, D) = (3, 5, 1)$, the average RT for SRAA is 10.45 seconds, while for $(n, K, D) = (6, 5, 1)$, the average RT for SRAA is 14.3 seconds. Note that the observed RTs are longer because the larger sample size means that rejuvenation is triggered later, as it takes longer to collect a larger sample.

5.3 SRAA, bucket depth doubled

Fig. 12 presents RT results for SRAA with $n \cdot K \cdot D$ = 30. We consider $(n, K, D) = (1, 3, 10)$, $(1, 5, 6)$, $(3, 1, 10)$, $(3, 5, 2)$, $(5, 1, 6)$, $(5, 3, 2)$, $(15, 1, 2)$, doubling the bucket depth component of the experiment settings in Section 5.1 to assess the impact on the RT.

Comparing Fig. 12 and 11, we see that doubling the bucket depth has a less severe impact on performance than doubling the sample size.

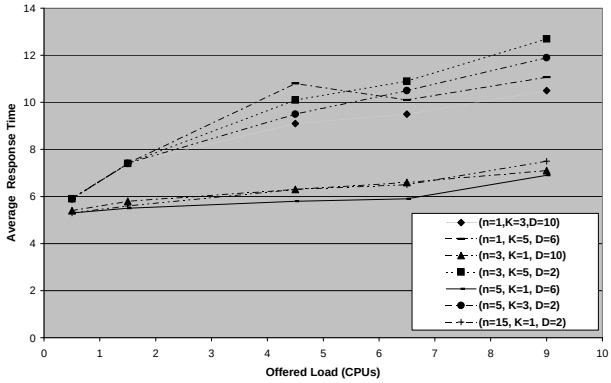

Figure 12. Response time results, SRAA, $n \cdot K \cdot D$ = 30, impact of bucket depth size doubling

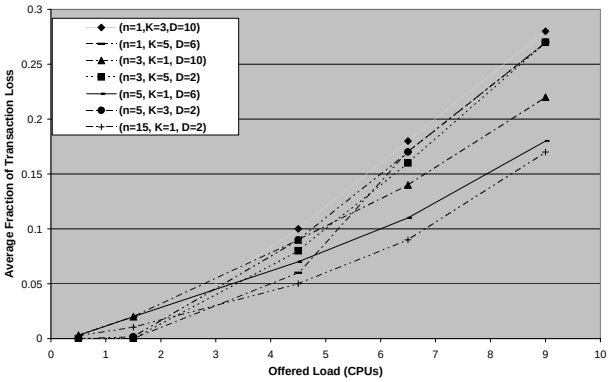

Figure 13. Fraction of transaction loss results, SRAA, $n \cdot K \cdot D$ = 30, impact of bucket depth size doubling

However, doubling of the bucket depth does decrease the average fraction of transaction loss as can be seen from Fig. 13 for the configurations with more than one bucket. Specifically, for $(n, K, D) = (1, 3, 10)$, $(1, 5, 6)$, $(5, 3, 2)$, the fraction of transaction loss for CPU loads of 0.5 CPUs is negligible. However, for $(n, K, D) = (3, 1, 10)$, $(5, 1, 6)$, $(15, 1, 2)$, we do observe measurable fractions of transaction loss for a low load of 0.5 CPUs.

5.4 SRAA, number of buckets doubled

Fig. 14 presents RT results SRAA, with $n \cdot K \cdot D$ = 30. For this evaluation we executed experiments with values the $(n, K, D) = (1, 6, 5)$, $(1, 10, 3)$, $(3, 2, 5)$, $(3, 10, 1)$, $(5, 6, 1)$, $(15, 2, 1)$, $(15, 1, 2)$, by doubling the number of buckets component of the experiment settings from section 5.1. The objective of these experiments are to assess the impact of the number of buckets parameter on the algorithm performance. We notice from Fig. 14 that doubling the number of buckets has a negative impact on performance. For example for the load of 9.0 CPUs and the value of $(n, K, D) = (15, 1, 1)$ the average RT for SRAA is 6.2 seconds, while for the value of $(n, K, D) = (15, 2, 1)$ the average RT for SRAA is 11.05 seconds. For the same CPU load of 9.0 CPUS, for the value of $(n, K, D) = (3, 5, 1)$ the average RT for SRAA is 10.45 seconds, while for the value of $(n, K, D) = (3, 10, 1)$ the average RT for SRAA is 14.9 seconds. However, doubling the number of buckets generates the best tradeoff configuration for low transaction loss at low loads and reasonable RT at high loads. For example, for configuration $(n, K, D) = (3, 2, 5)$, the average fraction of transaction loss for a 0.5 CPUs load is 0.000026 while the average RT for a 9.0 CPUs load is 10.3 seconds. In addition, the second best tradeoff is the configuration of $(n, K, D) = (5, 2, 3)$ for which the average fraction of transaction loss for 0.5 CPUs is 0.0003 while the average RT for 9.0 CPUs is 10.4 seconds.

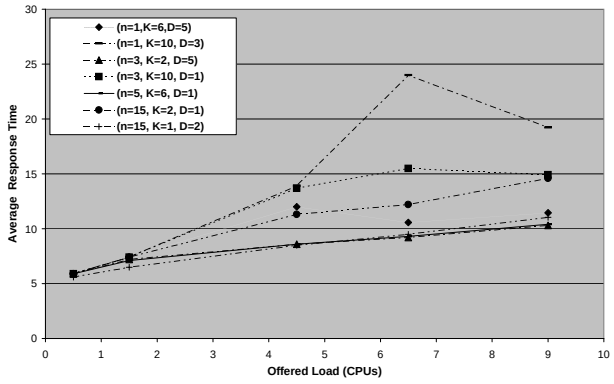

Figure 14. Response time results, SRAA, $n \cdot K \cdot D$ = 30, impact of number of buckets doubling

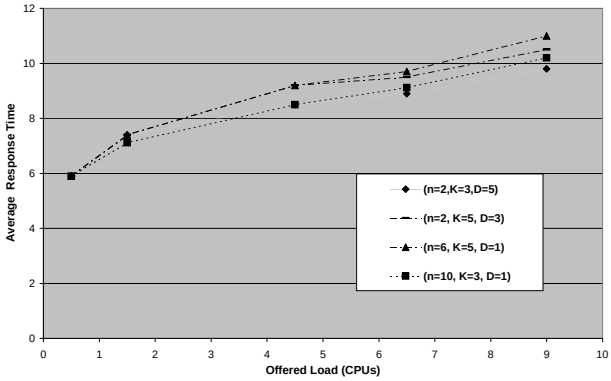

Figure 15. Response time results, SARAA, $n \cdot K \cdot D$ = 30

Therefore, we conclude that an adequate combination of bucket depth and number of buckets is required to deliver good performance at low loads.

5.5 SARAA

Fig. 15 presents RT results for SARAA, with $n \cdot K \cdot D$ = 30. We consider $(n, K, D) = (2, 3, 5)$, $(2, 5, 3)$, $(6, 5, 1)$, $(10, 3, 1)$. Fig. 15 shows that SARAA offers improvement over SRAA in RTs at high loads, while maintaining the negligible average transaction loss characteristics at low loads. For example for 9.0 CPUs and $(n, K, D) = (2, 5, 3)$, the SRAA RT is measured as 11.94 seconds while the SARAA RT is measured as 10.5 seconds. For $(n, K, D) = (2, 3, 5)$, SRAA RT is 11.05 seconds while SARAA is 9.8 seconds. For $(n, K, D) = (6, 5, 1)$, the for SRAA is 14.3 seconds while the for SARAA is 11 seconds.

5.6 Comparing SRAA, SARAA, and CLTA

Fig. 16 compares results for SRAA, SARAA, and CLTA. We use CLTA with $(n, K, D) = (30, 1, 1)$ and $N = 1.96$, and SRAA and SARAA with $(n, K, D) = (2, 5, 3)$. The objective of this experiment is to compare the performance of the three algorithms while $n \cdot K \cdot D$ = 30. The intent of CLTA was to use a sample size which is large enough for employing the normal approximation following from the central limit theorem. The value of N is the quantile of the normal distribution selected based on the acceptable rate of false alarms. Therefore, comparing the performance of CLTA with $(n, K, D) = (30, 1, 1)$ and SRAA and SARAA with $(n, K, D) = (2, 5, 3)$ helps clarify the domain of applicability of each algorithm. Recalling that our basis for assessment is the average at high loads and the amount of transaction loss at low loads, we see from Fig. 16 that CLTA with $(n, K, D) = (30, 1, 1)$ leads to performance degradation at both low and high loads as compared to SRAA and SARAA with the settings $(n, K, D) = (2, 5, 3)$.

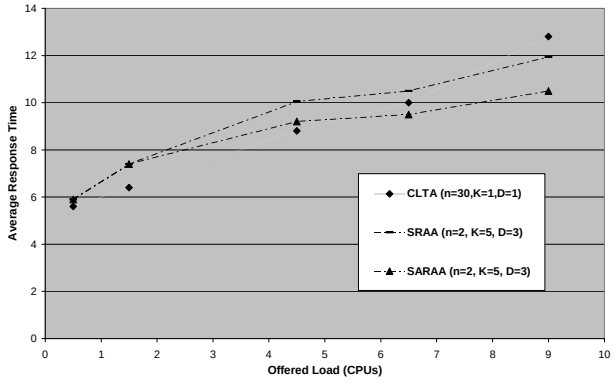

Figure 16. Response time results comparing SRAA, SARAA, CLTA algorithms, $n \cdot K \cdot D$ = 30

For example, for 0.5 CPUs, SRAA and SARAA drop a negligible fraction of transactions while CLTA drops 0.001406, while for 9.0 CPUs, the average is 10.5 seconds for SARAA and 11.94 seconds for SRAA, and 12.8 seconds for CLTA.

6 Conclusions

We have introduced three new software rejuvenation algorithms. We performed empirical studies applying the algorithms to a large e-commerce system for which we had observed significant performance degradation in the field. We used a simulation model of this system to assess the performance of the three algorithms and to study the performance sensitivity to variations on three important algorithm parameters: sample size, number of buckets, and bucket depth. We conclude that optimal performance can be achieved by using the sample size to determine precision of the RT estimation, number of buckets to determine tolerance to burst of arrivals, and bucket depth to accurately detect performance degradation. As a result, configurations that use small values of each of the parameter are better than configurations that invest in only one dimension. For example $(n, K, D) = (2, 5, 3)$ yielded better performance than $(n, K, D) = (30, 1, 1)$. Increasing the number of buckets produced large RTs. Very small numbers of buckets produced measurable rates of transaction loss at low loads. Therefore, the three new algorithms presented are able to control performance by determining appropriate times to perform software rejuvenation. However, care needs to be taken to optimize each algorithm and parameter configuration to the domain of applicability, since for each parameter configuration, a significant range of performance was detected. We have found that adaptive techniques, as employed by SARAA, can provide performance improvement by dynamically matching each algorithm performance to the real-time field conditions. Therefore, we plan to consider statistical estimation techniques to determine optimal algorithm parameters in real-time.

References

[1] A. Avritzer, A. Bondi, and E. J. Weyuker, "Ensuring stable performance for systems that degrade," in *Proc. Fifth International Workshop on Software and Performance*, 2005, pp. 43–51.

[2] ——, "Ensuring system performance for cluster and single server systems," *Journal of Systems and Software*, 2006. (To appear).

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

[4] ——, "The role of modeling in the performance testing of e-commerce application," *IEEE Trans. Software Engineering*, vol. 30, no. 12, pp. 1072–1083, 2004.

[5] A. Bobbio, A. Sereno, and C. Anglano, "Fine grained software degradation models for optimal rejuvenation policies," *Performance Evaluation*, vol. 46, no. 1, pp. 45–62, 2001.

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

[7] D. Gross and C. M. Harris, *Fundamentals of Queueing Theory*, 3rd ed. New York: John Wiley & Sons, 1998.

[8] M. Grottke and K. S. Trivedi, "Software faults, software aging and software rejuvenation," *Journal of the Reliability Engineering Association of Japan*, vol. 27, no. 7, pp. 425–438, 2005.

[9] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton, "Software rejuvenation: Analysis, module and applications," in *Proc. Twenty-fifth International Symp. on Fault-Tolerant Computing*, 1995, pp. 381–390.

[10] M. F. Neuts, *Matrix-Geometric Solutions in Stochastic Models : An Algorithmic Approach*, Baltimore: The Johns Hopkins University Press, 1981.

[11] R. Rajamony and M. Elnozahy, "Measuring client-perceived response times on the WWW," in *Proc. Third USENIX Symp. on Internet Technologies and Systems*, 2001.

[12] R. A. Sahner, K. S. Trivedi, and A. Puliafito, *Performance and Reliability Analysis of Computer Systems: An Example-based Approach Using SHARPE*. Boston: Kluwer Academic Publishers, 1996.

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

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

[15] K. S. Trivedi, K. Vaidyanathan, and K. Goševa-Popstojanova, "Modeling and analysis of software aging and rejuvenation," in *IEEE Annual Simulation Symposium*, 2000, pp. 270–279.