Using Machine Learning for Non-Intrusive Modeling and Prediction of Software Aging

Artur Andrzejak
Zuse Institute Berlin (ZIB)
Takustraße 7, 14195 Berlin
Germany
Email: andrzejak@zib.de

Luis Silva
Dep. Engenharia Informática
Univ. Coimbra
Portugal
Email: luis@dai.uc.pt

***Abstract*—The wide-spread phenomenon of software (running image) aging is known to cause performance degradation, transient failures or even crashes of applications. In this work we describe first a method for monitoring and modeling of performance degradation in SOA applications, particularly application servers. This method works for a large class of the aging processes caused by resource depletion (e.g. memory leaks). It can be deployed non-intrusively in a production environment, under arbitrary service request distributions. Based on this schema we investigate in the second part of the paper how machine learning (classification) algorithms can be used for proactive detection of performance degradation or sudden drops caused by aging. We leverage the predictive power of these algorithms with several techniques to make the measurement-based aging models more adaptive and more robust against transient failures. We evaluate several state-of-the-art classification methods for their accuracy and computational efficiency in this scenario. The studies are performed on a data set generated by a TPC-W benchmark instrumented with a memory leak injector. The results show that the probing method yields accurate aging models with low overhead and the machine learning approach gives statistically significant short-term predictions of degrading application performance. Both approaches can be used directly to fight aging via adaptive software rejuvenation (restart of the application), for operator alerting, or for short-term capacity planning.**

I. Introduction

Software (running image) aging is defined as the gradual performance degradation of running software images due to unreleased resources, accumulation of numerical errors, and file system degradation [10]. This undesired phenomenon is especially visible in long-running software such as SOA applications (web and application servers) and always-on applications - software deployed frequently in enterprise and utility computing environments. The management costs caused by this problem are considerable as it might be hard to pinpoint this problem in complex systems, and even if identified, it cannot be removed due to "black box" nature of the applications [3].

While the optimal solution to this problem is to fix the software bugs, in practice this can be rarely applied due to application complexity, lack of source code, or budget constraints. A common solution termed *software rejuvenation* is to restart the software after certain time, at a specific performance degradation level, or by other criteria [14], [3]. Recently this approach has been investigated in context of application replication in order to avoid service outages [2].

To deploy rejuvenation effectively, models of the aging process are needed. They allow to estimate the current or future progress of the performance degradation, and facilitate scheduling of optimal rejuvenation times. This approach is known as the *adaptive* or *proactive* software rejuvenation [6], [8]. Moreover, such models can be useful in a variety of management tasks, such as alerting of operators of anticipated crash, or short-term capacity planning.

In this work we make two contributions concerning modeling of aging processes. First, we consider the problem of obtaining the performance measurements or a complete performance model of the an application or web server in a production environment, i.e. without any dedicated testing setup. We understand as performance the maximum number of requests which a server can process per second. In a production environment the service request rates are usually lower, and so this performance is "hidden" until it drops to an intolerable level. We propose an approach based on adaptive probing which allows for non-intrusive monitoring and creation of accurate performance models as a function of performed work (served requests). We evaluate this technique and show that it yields accurate performance models without significant acceleration of aging.

In the second part of the paper we study the usage of classification algorithms for predicting application performance. The selected algorithms include Naive Bayes, decision trees, and Support Vector Machines [18]. They are computationally efficient and can be trained on-line and incrementally. This allows for an early deployment of the predictors (i.e. with relatively little historical data) and facilitates an automated model adaptation. Furthermore, by using certain inputs we allow the algorithm to adapt to transient changes of the performance levels.

The proposed prediction methods are evaluated via extensive studies on a data set generated by a TPC-W benchmark instrumented with a memory leak injector. The errors (memory leaks) are injected with each request, yet we inject non-deterministically additional leaks in order to test the adaptation to transient failures. The results show that the classification algorithms make accurate and statistically significant short-

978-1-4244-2066-7/08/$25.00 ©2008 IEEE

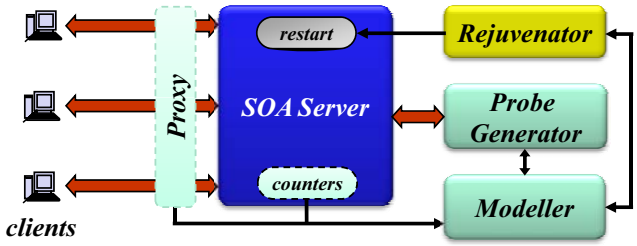


Figure 1. Architecture of the modeling and rejuvenation framework

term predictions of the synthetic aging processes in TPC-W. Moreover, the predictions are accurate already with a small training sets. This allows for more frequent classifier training and so higher adaptability.

Summarizing, our findings show that the presented methods are useful and practical for modeling and prediction of aging under production settings.

The paper is organized as follows. Section II describes the probing approach for measuring server performance under production settings. In Section III we present the method for classification-based prediction of aging processes. Section IV discusses the experimental evaluation of both approaches. In Section V we outline the related work, and finally we state conclusions in Section VI.

II. Performance Monitoring and Modeling for SOA Applications

The central term of this paper is *performance* of an application server: the maximum number of requests it could serve at a given moment. We denote it as P and express it in number of requests per second. The related variables are: *request rate* (number of incoming requests per second) and the *service rate* (number of actually served requests per second). Of course, the latter is never larger than the performance while the request rate might be above it.

For applications with degrading or variable performance, it is useful to have a model or a predictor of the performance due to the reasons discussed above. Note that the current service rate might be quite unrelated to the current performance (from being much lower to almost equal), and so it gives little clues for estimating server performance or the progress of the aging.

There are several approaches for measurement-based modeling of aging processes, including Markov chains [17], time-based models [14], and work-based models [3]. The later estimate the aging indicator as a function of the number of work done since restart, which is in case of application servers the number of served requests. Such models have several advantages: they describe accurately a wide class of aging processes caused by resource depletion such as memory leaks. Furthermore, they are independent of the request or load distribution, and so more universal than time-dependent models. Finally, in SOA applications the work done can be easily approximated. In the following we understand as an aging (or a performance) model a function which approximates the server performance depending on the number of served requests.

Obtaining aging models under real-world conditions can be tedious and costly. In case of work-based models, two alternatives exist. The first option is to put an application under a stress test by using a request rated exceeding its performance, and measure the performance change [3]. Although this process shortens considerable time until a complete aging cycle is recorded, it requires a dedicated environment and time, increases management costs, and might not accurately reflect the production scenario. While we have used this method in laboratory conditions in order to estimate the quality of our approach, it is rarely feasible in a production setting.

The second option is to use the "natural" client request traffic in a production setting to model performance. The essential difficulty is that it gives the opportunity to learn the true performance only if it contains load which exceeds the server capacity. This is very rare, uncontrollable and occurs usually only under abnormal conditions. Our initial experiments have shown that the performance models obtained in this way are very inaccurate, and describe the aging profile only shortly before the complete crash, thus providing little predictive power. Furthermore, the approach requires additional instrumentation to count the number of dropped *client* requests.

A. Determining performance by probing

To overcome these issues of the modeling in the production setting we propose the following approach. We repeatedly probe the server via artificially created request groups (called *probes*) with high request rate exceeding expected performance. By measuring the number of requests serviced and dropped by the application we can accurately determine the momentary performance. The probing requests are considered as dropped if they are not serviced at all or if the response latency exceeds a tolerable threshold.

Special attention must be given to generating such probing requests. The most promising way seems to be the replaying the previously recorded client interactions, especially if these interactions are more involved. Such request groups can be generated from simulated browsers like those in the TPC-W benchmark [13]. Note that such probes must be issued and measured externally to the application, so that monitoring techniques like DTrace [5] are of limited use.

In order to update the above-mentioned performance model, the knowledge of the number of served requests since the last rejuvenation is necessary. This can be implemented either by a proxy which counts the number of served client requests (or, as approximation, the number of incoming client requests), or by direct querying of the application statistics, see Figure 1. The second method is more intrusive yet incurs less overhead.

The *accuracy* of the obtained models determines the efficiency of the above-discussed management solutions. For example, a model understating the true performance might trigger too frequent healing or rejuvenation actions, while

overestimating the true performance could lead to an unexpected crash. To quantify this variable in a laboratory setup we use the mean squared error (MSE) and the mean absolute error (MAE) of the deviations between the true performance (measured under burst load, see Section IV-A) and the model generated by our method. Prior to the computation of these variables, both models are gridded to obtain equal number of sampling points.

B. Probing overhead

Under the assumption that the performance degradation depends on the number of requests our probing will accelerate the aging. The magnitude of this undesirable effect can be measured by the following variable called *overhead* (requests are counted over the whole rejuvenation cycle):

$$\%\ Oh \quad = 100\,\frac{\#\text{ served probing requests}}{\#\text{ all served requests}}. \qquad (1)$$

Clearly, the more accurate model is requested, the more frequent probing is necessary, and so the overhead increases. However, for a fixed accuracy level the overhead can be minimized by the following measures.

First, the probe triggering policy can be tweaked. A naive policy is to issue probes in regular time intervals. This *time-based* triggering policy creates unnecessary overhead if the client request rate is low (since a large share of the served requests might come from the probes). A better solution is to issue a probe after every fixed number of served requests. We call it a *request-based* triggering policy. Under our aging process assumptions this schema should yield good performance models even if the time interval between probes is large: a small number of client requests served between probes implies small performance change. In the experimental part we evaluate the relation of this overhead and the model accuracy for both policies and their parameters.

Secondly, the probing demand spikes should be long enough to create dropped requests but as short as possible. A suitable duration might depend on many factors, such as maximum tolerable response latency threshold, application stress response patterns (it might store not served requests and respond to them with larger latency), current client request rate, and others. As a consequence, application-specific experimental testing is needed.

As another undesirable side effect, client requests might be dropped during the probing. The solutions here include triggering of the probing only during periods of lower than average requests rates, or delaying the request via a proxy, and replaying them after the probing phase is finished.

C. Simulation details

We have used simulations based on real data to evaluate the approach. Each simulation consisted of the following phases.

One. The performance of an application server under the TPC-W benchmark with a memory-leak injector is recorded under a request rate exceeding maximum performance (see Section IV-A). The sampled values are stored in a model "container" which returns performance values for a specified number of requests. For later performance queries, the values between original samples are obtained by linear interpolation.

Two. We generate a sequence of requests (over simulated time) and emulate the behavior of the application server. Essentially, for each simulation step we determine its time duration and the number of the requests to be served. The implied request rate is compared with the current server performance, and the number of served and the number of dropped requests is recorded. Also the sum of served requests (of clients and of probes) is updated. The request generation is done by interweaving simulation steps for the probing requests or for the client requests. The "probing steps" have fixed time length and number of requests, while their triggering is done according to the time-based or request-based policy. The "client steps" are equidistant in time, yet have different number of requests per step according to the distribution type and its parameters.

Three. Finally use the proposed method to reconstruct the original performance model. For each time step of the trace from the previous phase we determine whether any requests have been dropped. If this is the case, we can measure the server performance, and update the reconstructed performance model. For each step the sum of the requests served since rejuvenation is recorded. The reconstruction ends when the ratio of dropped requests to served requests is larger than 0.99 which indicates that the server is crashing.

For the phase two we have used two types of request generators. The first, "*tpc-w*" is taken directly from the TPC-W benchmark. Here a set of b emulated browsers issues page requests independently. After each request, a browser waits a "think time" which is governed by the exponential distribution with a mean of 7 (seconds). This generator is considered as a good approximation of real web server load [1]. The second generator named "*real*" replays (scaled) request rate of the campus-wide web server at the University of Saskatchewan [4] (7 months starting 6/01/1995). This trace features daily and weekly fluctuations (see left figure of Figure 3), while the "tpc-w" generator is more uniform on larger time scales.

III. Predicting Performance by Machine Learning

The aging models based on the method from Section II assume that the aging process is "stable" and looks similar in each rejuvenation cycle. While this applies to a large number of aging processes caused by resource depletion [3], such models might fail if the dependencies are more complex or transient failures occur. In a such scenario, we propose use machine learning classification methods to predict performance in the near future. Such algorithms are capable to model implicitly complex relationships, and become more accurate with growing amount of the historical data. In general, this method

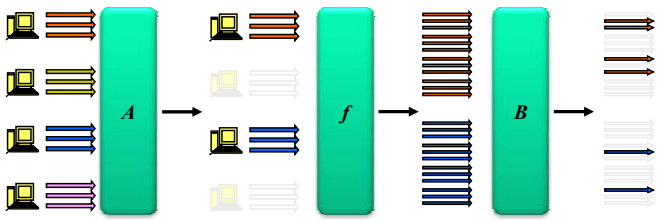


Figure 2. Attribute selection process for a single prediction target

is capable to predict the server performance or to predict the value of any other *aging indicator*: a system variable which most accurately describes the progress of running image aging.

A. Key ideas

We assume a scenario of a long-running application such as web service, web server, or an enterprise always-on application. We further assume that the application runs in an environment which allows for measuring its performance (by an approach presented in Section II or some other) and possibly other other system statistics. The collected data is fed periodically to an aging and rejuvenation manager which creates a model and initiates rejuvenation actions, see Figure 1.

We use a data mining classification algorithm such as Naive Bayes [18] to obtain an implicit model of the aging process. This representation can be used directly to predict a (discretized) performance value. The advantage of classification algorithms is that we can use simultaneous multiple data inputs for the model. This can increase accuracy and allow for incorporating potential correlations between different applications, OS components, and even different hosts.

One of the algorithm inputs is the instantaneous value of the performance to be predicted, and some averages thereof. This helps the classifier to accommodate for transient performance changes which might be not reflected by other inputs. For example, assume that the performance drops suddenly by 50% due to a very rare bug and this change is not visible from other inputs like number of served requests or CPU usage. Including the current performance as an input is likely to help the classifier to "override" the learned aging curve and predict a performance level adjusted to the currently observed one.

The training of the prediction algorithm takes place incrementally and on-line. In this scenario, the classifier inputs and the performance indicator (prediction target) are recorded on-line (during deployment) and added incrementally as further training examples. In this way, the prediction algorithm does not need any special off-line training, it can be used productively shortly after the system deployment, and its accuracy increases with time. Furthermore, it is possible to use only the training data from some recent time window in order to adapt to changes in the aging profile.

B. Prediction via classification algorithms

We first describe the terminology and background on the classification algorithms. A *classifier* is a function $f : V \to W$ which assigns to a vector of values $v \in V$ a label $w \in W$ [18]. Here vector v has usually length larger one, and each of its columns corresponds to a specific type of an input scalar called *attribute*. In the setting of this paper, the labels are discretized levels of the performance P measured at a certain time t'. For example, if performance P at time t' is discretized into 20 levels, label $w = 1$ covers P values between 0% and 5% of the maximum, and label $w = 20$ corresponds to P values between 95% and 100% of the maximum performance. The attributes in our scenario are application metrics (and functions thereof) collected at some time $t < t'$. In this paper we used as attributes the application performance at time $t < t'$, time passed since last rejuvenation (measured at t), and functions of both (such as moving averages of different lengths and their differences). Of course, also additional inputs such as CPU, memory and disk utilization could be used, however our application instrumentation does not allow to collect them at this stage.

Before predictions can take place, classifier f is presented a set of examples (v, w) (attribute vectors with correct labels) from which it attempts to build a model of relationships between vector values and labels. After this so-called *training* or *fitting phase* predictions are performed: a vector v with an unknown label is given, and then the label $f(v)$ is computed. Note that in each training example (w, v) there is some (usually fixed) offset between the times when inputs for w and those for v are collected. This offset is called a *lag* and it describes how far into the future we wish to predict.

It is usually unclear which functions of the raw inputs should constitute the attribute vectors. An approach is to generate many such functions and then prune them until only the significant and mutually uncorrelated ones remain. This so-called *attribute selection* is an essential step preceding the training process. Due to the phenomenon called "curse of dimensionality", too many attributes in relation to the number of examples can easily lead to overfitting. We have developed a two-stage process which first selects the relevant (correlated) inputs/resources (phase A), then computes a pool of functions on traces of these inputs (phase f), and finally selects the final attributes from the pool of functions (phase B), see Figure 2. Since phase A is essentially a trace correlation analysis with low running time, we can specify a large set of inputs/resources (on the order of 100) as a potential input. As a by-product of this phase a ranked list of inputs potentially influencing the behavior of the target is produced. In phase B we apply the standard mechanism which prefers attributes with high correlation to the target yet low redundancy [9]. As noted above, we use currently only two inputs but this scalable selection approach will become essential when additional system metrics are collected.

The process of attribute selection is usually time consuming but needs to be performed only once for a prediction target.

The subsequent classifier training and testing require few seconds to minutes depending on the classification technique and size of the input. We have used in our study three major classification algorithms and a "primitive" zero-rule algorithm from the WEKA library of data mining algorithms [18]. We have used default WEKA options. The algorithms are specifically:

- J48 - the classical algorithm for generation of a decision tree, in this case a pruned or unpruned C4.5 decision tree [16]
- NaiveBayes - a simple yet computationally efficient probabilistic classifier based on applying Bayes' Theorem with the ("naive") assumption that the attributes have independent probability distributions [11]
- SMO - a Support Vector Machine based on John Platt's sequential minimal optimization algorithm [15]
- ZeroR - the 0-R classifier predicts in our case the mode of the label indices (approximately the mean of discretized performance given by the training examples) [18]. This classifier has been included for the purpose of base comparison against an unsophisticated prediction schema.

As the above algorithms are well-known and extensively described in the machine learning literature, we omit the details due to space constraints.

C. The prediction process

In our study the raw data is a series of samples, each composed of the recorded application performance and the time since last rejuvenation at the sampling time (see Section IV-A). During a prediction run this data is read in and for each sample a vector of attribute values is computed, together with the discretized (and time shifted) target value. The attributes are functions of the raw inputs determined by selection process described above. For example, the attributes for the prediction process with a lag of 50 samples and the samples 1 to 2246 as training examples had 29 attributes, including simple moving averages (SMA) of lengths $2, 5, 10, 20, 40, 120$ of the performance, SMAs of the time since rejuvenation of lengths $2, 5, 10, 20, 40, 60, 120$, as well as several difference functions of two SMAs on the same input (for example, SMA of length 120 minus SMA of length 2).

The prediction process consists of the alternating training and prediction phases. At first some set of the first S samples is used to build an initial model. Then R predictions are performed for samples $S+1$ till $S+R$. Here a new model is build which uses at most T samples prior to sample $S+R+1$. This model is used for the following R predictions for samples $S+R+1$ till $S+2R$ etc. This sequence is repeated until the end of the series. This procedure is known as *walk-forward testing*. It ensures adaptability to new data due to retraining every R samples, and prevents overfitting since each prediction is made for a sample which has not been previously used for training. The number of conducted predictions is total number of samples less S.

IV. Experimental Evaluation

A. Experimental setup

To evaluate the methods described above we are using data obtained from industry-typical web service applications. The data set comes from experiments with a Java implementation of the TPC-W benchmark. This benchmark resembles a on-line book store and uses Tomcat as the container and MySQL as the database. Since the original TPC-W implementation does not show any visible aging problem we have implemented a small fault-injector that works as a resource parasite: it consumes system resources in competition with the application [8]. The memory leak of size 1024 bytes is injected each time a request is served. In addition, we inject non-deterministically memory leaks memory leaks of size 1 to 100 kilobytes in order to simulate transient errors and non-deterministic aging effects. The rejuvenation time for the TPC-W software ranged between 12 and 15 seconds.

To speed-up the occurrence of software aging we deployed a multi-client tool called QUAKE. This tool permits the launching of simultaneous multiple clients that execute requests in a server under-test. All together, we have used a cluster of 12 machines: 10 running the client benchmark application, one database server and another server running the aging application. All the machines are interconnected with a 100Mbps Ethernet switch.

In these studies, we used the burst distribution to test the maximum performance of the application and sampled this every 30 seconds. After the request rate slow down to below 1 requests/s (which took place after approximately 2 hours and 15 minutes) we restarted the application (i.e. performed rejuvenation). This experiment has been repeated 25 times. The concatenated performance plot is shown in Figure 4 (left). While the performance curves are similar in the first 60% of each cycle, they differ in the remaining part of the cycle due to non-deterministic memory leaks.

B. Reconstructing aging models

In this section we evaluate quantitatively results of the simulations described in Section II-C. We have used for the original performance model data from the first of the 25 rejuvenation cycles described in Section IV-A (other datasets behaved similarly). The goal of the evaluation was to compare the two approaches for probe triggering and find in each case parameter values which incur low overhead yet yield highly accurate models.

To this aim, we first performed experiments with the time-based probe triggering method, and used as the interval T after which a new probe is inserted the values 5s, 10s, 30s, 1min, 2min, 5min, 10min, 30min, 1h, and 2h. In the request-based triggering mode a new probe has been inserted every $D =$ 50, 125, 250, 500, 1000, 2000, 4000, 8000, 16000, and 32000 served requests. The number of requests per probe was 100, and the probe duration 1s. We set 30s as the resolution of the time trace from phase two.

To compensate for the effects of a particular request rate distribution in the phase two of the simulation (Section II-C)

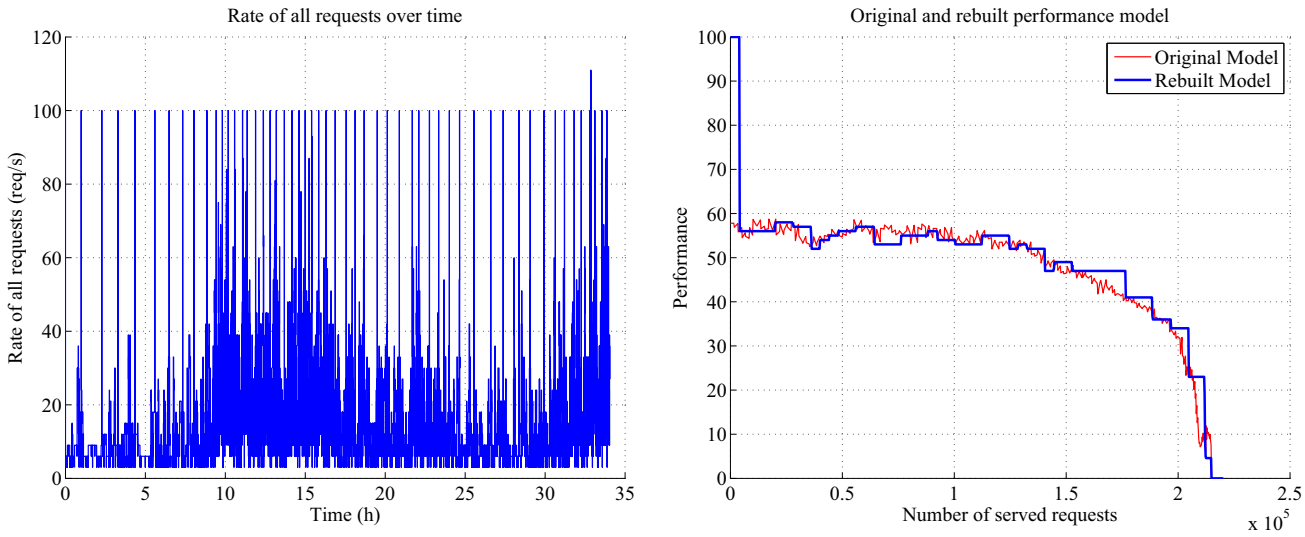


Figure 3. Request rate history of a simulation (left); original and rebuilt performance model (right)

we have performed for each of the above cases 11 simulations with different distributions. The first 6 simulations were done with the "tpc-w" distribution and the following number of emulated browsers: 1, 5, 10, 15, 20, 25. For the "real" request rate distribution we used different multiplication factors for the original request rate, an used the values $1, \ldots, 5$. These values have been chosen such that the request rates cover the full range from almost 0 to the maximum server performance. In each simulation the mean squared error (MSE) and the mean absolute error (MAE) of the of the reconstruction accuracy has been recorded, as well as the overhead Oh (Equation (1)). Then we averaged these results over 11 cases corresponding to the same T or D values.

Table I states these averaged results. As expected, there is a trade-off between accuracy and overhead. Moreover, the request-based probe triggering method allows for slightly higher model accuracy (lower MSE or MAE) than the time-based method at the same overhead level. For example, at about 2.5% overhead level the time-based method has an MSE of almost 17, while the MSE of the request-based method is less than 9.

Figure 3 illustrates a typical request rate history (of the phase two) and the corresponding original and reconstructed models. Here request-based probe triggering is used with $D = 4000$, and the "real" request rate distribution with a scaling factor of 3. The left figure shows the time plot of the request rate (client and probing requests), while the right figure shows the original and the reconstructed performance models. As visible in the plot, the rebuilt model overestimates initially the performance (with the default value of 100 requests/second) until the first probe is issued. The results here are: MSE = 9.32, MAE = 0.60, and Oh = 1.22%. In general, the results show that our probing approach yields sufficiently accurate performance models with a negligible overhead.

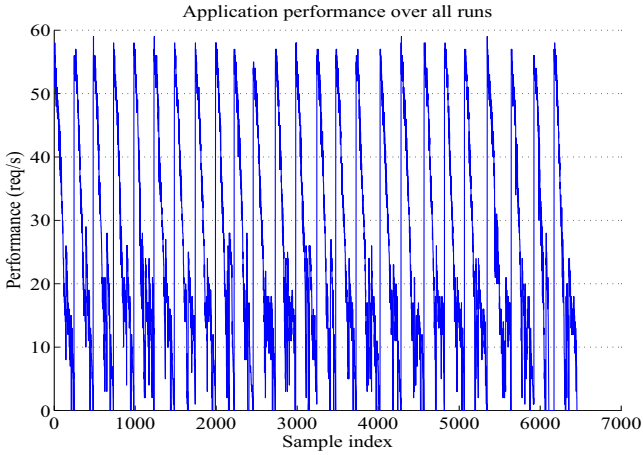
T	MSE	MAE	% Oh
5s	0.11	0.13	76.01
10s	0.12	0.13	76.08
30s	0.71	0.16	51.46
1min	1.64	0.20	36.38
2min	2.03	0.22	23.90
5min	3.30	0.28	12.29
10min	5.00	0.36	6.92
30min	**16.92**	**0.71**	**2.56**
1h	39.84	1.31	1.32
2h	36.04	1.32	0.68

D	MSE	MAE	% Oh
50	0.35	0.15	87.08
125	2.01	0.19	35.87
250	2.34	0.23	18.60
500	3.02	0.28	9.50
1000	3.93	0.34	4.81
2000	**8.55**	**0.51**	**2.42**
4000	18.64	0.79	1.22
8000	31.27	1.22	0.62
16e3	72.94	2.29	0.32
32e3	136.87	3.75	0.16

Table I
AVERAGED ACCURACY AND OVERHEAD OF THE MODEL RECONSTRUCTION (TOP: TIME-BASED PROBE TRIGGERING, BOTTOM: REQUEST-BASED PROBE TRIGGERING)

C. Prediction experiments and their evaluation

We have used different combinations of algorithms and settings to conduct all together 96 prediction runs. The total processing time (including attribute selection) was in 55 minutes using Java 1.5.0_10 on a single core of a 2.16 GHz Intel Core Duo (T2600) laptop running under Windows XP. With approximately 5250 predictions per experiment the amortized time per prediction is on average 6.5 milliseconds. However, SMO algorithms need a multiple of the time taken by Naive Bayes or J48, and so the latter algorithms are faster than this average.



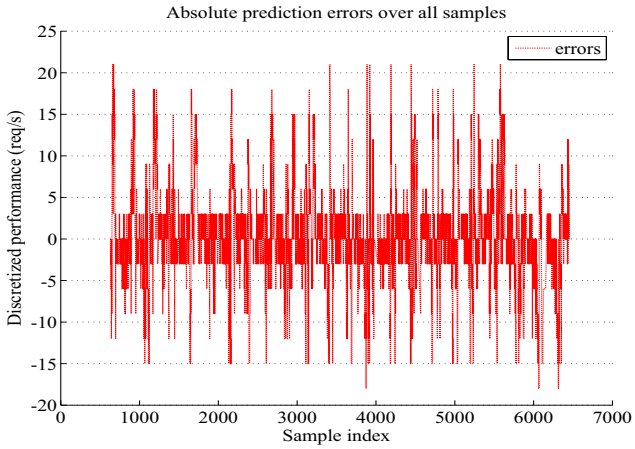
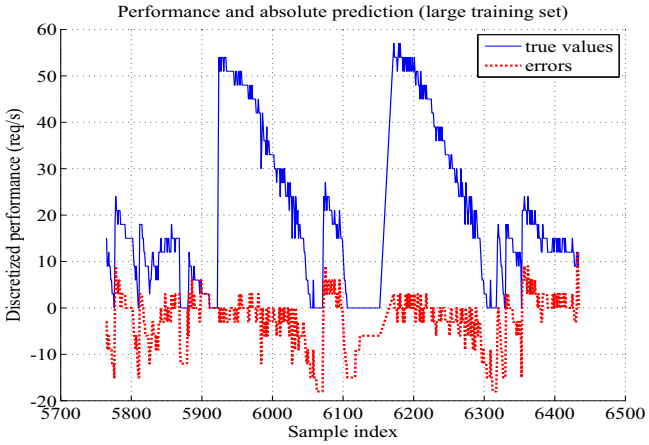


Figure 4. Application performance (left) and prediction errors over all 25 rejuvenation cycles (right, classifier = SMO, lag = 5min)



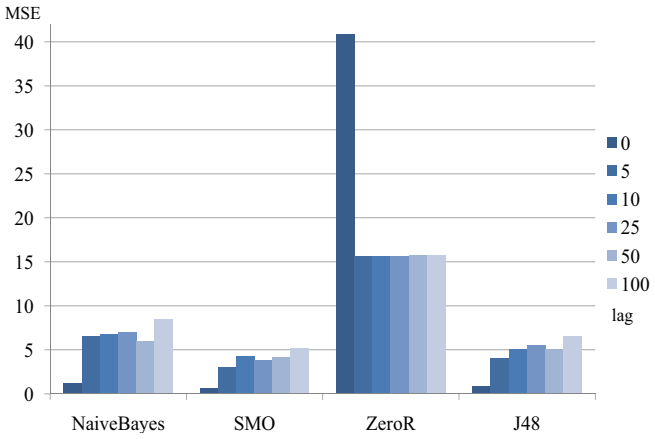


Figure 5. Left: prediction errors for final rejuvenation cycles (SMO, $\ell = 10$); right: MSE averages for different classifiers and lags (lag in 30s units)

We tested each of the four algorithms described in Section III-B with the combinations of following parameters expressed in the number of samples (i.e. 30 seconds units): lag $\ell = 0$, 5, 10, 25, 50, 100; maximum training interval length $T = 3000, 6000$; model update interval length $R = 1000, 3000$. No predictions have been made before sample number 622 (i.e. S defined in Section III-C equals 622) to ensure that at least one whole rejuvenation cycle is used for classifier training before the first prediction. We have used 20 discretization levels over the interval $[0, 60]$ as the classifier output labels, i.e. label 1 corresponds to performance between 0 and 3, and label 20 to performance in $[57, 60]$.

Figure 4 (right) shows the absolute prediction errors (true label index minus predicted label index) for the SMO classifier ($\ell = 10$, $T = 6000$, $R = 1000$) over the whole prediction range. The shown algorithm produces larger errors at the ends of the rejuvenation cycles since the original data is less deterministic here. Figure 5 (left) shows the final part of the above prediction run (together with the performance data). We obtain a similar figure for initial rejuvenation cycles. This confirms that the prediction accuracy is high already for small training sets.

D. Comparing prediction algorithms and their settings

To evaluate prediction errors for a whole series we also used the Mean Squared Error (MSE) [18] measure applied to differences between the predicted and correct (discretized) performance values. In Figure 5 (right) we show the MSE averages for different lag values (0, 5, 10, 25, 50 and 100 samples) and all four classification methods. Each bar represents an average of MSE values over four prediction runs corresponding to all combinations of $R = 1000$, 3000 and $T = 3000, 6000$. As the variance of the correct discretized performance values ranged between 31.95 and 32.33 (depending on the lag), most cases yield non-trivial predictions (a random guessing approach would have given an MSE value around 32).

Figure 5 (right) shows also that - as expected - the primitive ZeroR predictor has much smaller predictive power, and performs badly for all lag values. Interestingly, for $\ell = 0$ (i.e. "predictions" for the current moment) none of the classifiers is sophisticated enough to perform error-free as it would

be possible. As the input values are not discretized, the MSEs for this case represent possibly an incorrectly "learned" discretization procedure.

As expected, the algorithms perform better for shorter update intervals $R = 1000$ (more frequent adaptation) and shorter maximum training interval $T = 6000$ (adaptation to more recent data). However, the differences are less pronounced than for different lags.

V. Related Work

The primary method to fight aging is software rejuvenation, i.e. a restart of the aging application periodically or adaptively. The latter approach takes into account the progress of aging and the effects of transient errors in order to find rejuvenation schedules which maximize the overall application availability and performance [6]. It has obvious advantages over the periodical rejuvenation schema yet it requires models of aging / performance of the investigated application.

There are two basic approaches to apply proactive software rejuvenation: (i) Analytic-based approach; (ii) Measurement-based approach. The first approach uses analytic modeling of a system, assuming some distributions for failure, workload and repair-time. A survey about papers that follow this approach can be found in [17].

The measurement-based approach is simpler yet usually more accurate. Here the goal is to collect some data from the system and then quantify and validate the effect of aging in system resources [7], [6], [12]. Our previous work [3] falls into this category. Here we used a spline-based description of the aging profiles and a statistical test to verify its correctness. In [2] we extended this method to the scenario of replicating the aging application.

The above approaches have some drawbacks: they require an initial data collection over many rejuvenation cycles to establish a prediction model, they are not robust against transient failures which invalidate the aging model established under error-free conditions, and the computation of the aging model might be expensive, especially in the case of ARMA-based [12] models. The classification-based aging process prediction partially eliminates these disadvantages.

VI. Conclusions

In this paper we have described two techniques related to modeling of aging processes. The first one uses measuring of server performance by artificial probes with high request rates to obtain an aging model. The evaluation via simulation based on real aging traces and realistic request distributions confirmed that it is possible to create accurate aging models in a production setting with negligible overhead.

In the second part we studied usage of classification algorithms for predicting software aging processes in presence of partial non-determinism and transient failures. The algorithms yielded statistically significant predictions and achieved good accuracy even with small training sets, which allows for high adaptability. Furthermore, the amortized computational cost of a prediction turned out to be low and so predictions can be performed even with sub-second periodicity. The study also has shown that the major families of classifiers - decision trees (J48), Bayesian methods (NaiveBayes) and Support Vector Machines (SMO) perform comparably well.

VII. Acknowledgments

This research work is carried out in part under the FP6 Network of Excellence CoreGRID and the SELFMAN project, both funded by the European Commission. The authors would like to thank Mehmet Ceyran (ZIB) and Paulo J. F. Silva (UCO) for help with the experimental work.

References

[1] Cristiana Amza, Anupam Ch, Alan L. Cox, Sameh Elnikety, Romer Gil, Karthick Rajamani, Emmanuel Cecchet, and Julie Marguerite. Specification and implementation of dynamic web site benchmarks. In *Fifth Annual IEEE International Workshop on Workload Characterization (WWC-5)*, Oct 2002.

[2] Artur Andrzejak, Monika Moser, and Luis Silva. Managing performance of aging applications via synchronized replica rejuvenation. In *18th IFIP/IEEE Distributed Systems: Operations and Management (DSOM 2007)*, Silicon Valley, CA, USA, October 2007.

[3] Artur Andrzejak and Luis Silva. Deterministic models of software aging and optimal rejuvenation schedules. In *10th IFIP/IEEE Symposium on Integrated Management (IM 2007)*, Munich, Germany, May 2007.

[4] Martin F. Arlitt and Carey L. Williamson. Web server workload characterization: The search for invariants. In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pages 126–137, May 1996.

[5] Bryan Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference*, pages 15–28. USENIX, 2004.

[6] V. Castelli, R. Harper, P. Heidelberg, S. Hunter, K. Trivedi, K. Vaidyanathan, and W. Zeggert. Proactive management of software aging. *IBM Journal Research & Development*, 45(2), March 2001.

[7] S. Garg, A. van Moorsel, K. Vaidyanathan, and K. Trivedi. A methodology for detection and estimation of software aging. In *Proceedings of the 9th Int'l Symposium on Software Reliability Engineering*, pages 282–292, 1998.

[8] K. Gross, V. Bhardwai, and R. Bickford. Proactive detection of software aging mechanisms in performance critical computers. In *Proceedings of 27th Anual IEEE/NASA Software Engineering Symposium*, December 2002.

[9] M. Hall and L. Smith. Practical feature subset selection for machine learning. In *Proceedings 21st Australasian Computer Science Conference*, University of Western Australia, Perth, Australia, February 1996.

[10] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton. Software rejuvenation: Analysis, module and applications. In *Proceedings of Fault-Tolerant Computing Symposium FTCS-25*, June 1995.

[11] George H. John and Pat Langley. Estimating continuous distributions in Bayesian classifiers. In *Proc. 11th Conference on Uncertainty in Artificial Intelligence*, pages 338–345. Morgan Kaufmann, 1995.

[12] L. Li, K. Vaidyanathan, and K. Trivedi. An approach for estimation of software aging in a web-server. In *Proceedings of the 2002 International Symposium on Empirical Software Engineering (ISESE'02)*, 2002.

[13] Amit Manjhi. *TPC-W in Java on Tomcat and MySQL*. Carnegie Mellon University, 2005.

[14] András Pfening, Sachin Garg, Antonio Puliafito, Miklós Telek, and Kishor S. Trivedi. Optimal software rejuvenation for tolerating soft failures. *Perform. Eval*, 27/28(4):491–506, 1996.

[15] J. Platt. Fast training of support vector machines using sequential minimal optimization. In B. Schölkopf, C. J. C. Burges, and A. J. Smola, editors, *Advances in Kernel Methods — Support Vector Learning*, pages 185–208, Cambridge, MA, 1999. MIT Press.

[16] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA, 1993.

[17] K. Vaidyanathan and K. S. Trivedi. A comprehensive model for software rejuvenation. *IEEE Trans. Dependanble and Secure Computing*, 2(2):1–14, April-June 2005.

[18] Ian H. Witten and Eibe Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, San Francisco, 2nd edition edition, 2005.