Adaptive on-line software aging prediction based on Machine Learning

Javier Alonso and Jordi Torres
Barcelona Supercomputing Center
Dept. of Computer Architecture
Technical University of Catalonia
{alonso,torres}@ac.upc.edu

Josep Ll. Berral and Ricard Gavaldà
Dept. of Software
Technical University of Catalonia
{jlberral,gavalda}@lsi.upc.edu

Abstract

The growing complexity of software systems is resulting in an increasing number of software faults. According to the literature, software faults are becoming one of the main sources of unplanned system outages, and have an important impact on company benefits and image. For this reason, a lot of techniques (such as clustering, fail-over techniques, or server redundancy) have been proposed to avoid software failures, and yet they still happen. Many software failures are those due to the software aging phenomena. In this work, we present a detailed evaluation of our chosen machine learning prediction algorithm (M5P) in front of dynamic and non-deterministic software aging. We have tested our prediction model on a three-tier web J2EE application achieving acceptable prediction accuracy against complex scenarios with small training data sets. Furthermore, we have found an interesting approach to help to determine the root cause failure: The model generated by machine learning algorithms.

1. Introduction

As the complexity of software systems continues to grow, so increases the difficulty of managing them. Moreover, our current and growing reliance on these software systems to manage critical and ordinary tasks in our lives requires these software systems not only to offer an acceptable performance but continuous availability also. To meet these social needs, more skilled developers and administrators are needed to maintain these complex and heterogeneous software systems, resulting in a large fraction of the total cost of ownership (TCO) of these systems.

Because system complexity is growing day by day, the number of failures due (directly or indirectly) to this complexity has also been growing, resulting in undesirable behaviors, poor levels of service, and even total outages. The need to prevent or gracefully deal with outages of businesses and critical systems is clear, given the industry huge loss due to the downtime per hour. A recent study[1] showed the average downtime or service degradation cost per hour for a typical enterprise is around US$125,000. Moreover, outages have a negative impact on the company image that could affect profits indirectly.

It is now well known that currently, computer system outages are more often due to software, rather than hardware, faults [1, 2]. Although, more sophisticated developing/testing and debugging tools are appearing to help developers avoid software faults [3, 4], the faults still appear and have an important impact over the application availability. In fact, fixing all faults during the testing and debugging phase is a titanic task with an unaffordable cost, because these tools often cannot access third-party modules code. Even more, the transient and intermittent faults are too difficult to fix because it is highly complicated to reproduce them. Design faults too often dormant and they activate only under unknown or rare circumstances becoming in nature transient or intermittent software errors.

Several studies [5, 6] have reported that one of the causes of the unplanned software outages is the software aging phenomena. This term refers to the accumulation of errors, usually provoking resource contention, during long running application executions, like web applications, which normally cause applications/systems hang or crash [7]. Gradual performance degradation could also accompany software aging phenomena. The software aging phenomenon are often related to others, such us memory bloating/leaks, unterminated threads, data corruption, unreleased file-locks and overruns. Software aging has been observed in web-servers [8], spacecraft systems [9], and even military systems [10], with severe consequences such as loss of lives.

For this reason, the applications have to deal with the software aging problem in production stage, making software rejuvenation techniques necessary. Software rejuvenation strategies can be divided into two basic categories:

[1]IDC #31513, July 2004

Time-based and Proactive/Predictive-based strategies. In Time-based strategies, rejuvenation is applied regularly and at predetermined time intervals. In fact, time-based strategies are widely used in real environments, such web servers [11].

In Predictive/Proactive rejuvenation, system metrics are continuously monitored and the rejuvenation action is triggered when a crash or system hang up due to software aging seem to approach. This approach is a better technique, because if we can predict the crash and apply rejuvenation actions only in these cases, we reduce the number of rejuvenation actions with respect to the time-based approach. Software rejuvenation has an impact on the system and potentially, on the revenue of the company as well. Traditionally, software rejuvenation has been based on a restart of the application/system or even a whole machine, but, there are more sophisticated techniques, like micro-reboot [12], i.e. rebooting only the suspicious application component; this reduces the impact of the rejuvenation action.

Predicting the time until resource exhaustion due to software aging is far from easy. Progressive resource consumption over time could be non-linear, or the degradation trend could change along the time. Software aging could be related to the workload, or even the type of the workload. Software aging could also be masked inside periodic resource usage patterns (i.e. leaving a fraction of memory used allocated only after a periodic load peak). Another situation that complicates resource exhaustion prediction is that the phenomenon can look very different if we change the perspective or granularity used to monitor the resources; we will provide specific examples of this later on. This could be relevant, specially, when we are working with virtualized resources, as we do in Section 3. Still another difficulty for software aging prediction is that it can be due to two or more resources simultaneously involved in the service failure [13].

In this paper, we focus on our software aging prediction model based on M5P (a well-known ML algorithm) and its evaluation in front of a varied and complex software aging scenarios. In our preliminary previous work [14], we evaluated three algorithms of Machine Learning (ML) (Linear Regression, Decision Trees, and M5P) to check whether they offered the right capabilities to model software aging phenomena, and M5P offered the best performance. In this paper we have thus decided to use M5P to predict the *resource exhaustion time*, and try it in a new set of experiments, reproducing more complex software aging scenarios. Furthermore, M5P was selected because it has low training and prediction costs and we will eventually want on-line processing, and because it produces models that are easy to interpret by humans. More sophisticated ML techniques (i.e. Support Vector Machines, Neural Networks, Bayesian Nets, Bagging or Boosting) can surely obtain better accuracy, but we believe that M5P offers a good trade-off between accuracy, interpretability, and computational cost.

The rest of the paper is organized as follows: Section 2 describes our prediction modelling and strategy. Section 3 presents the experimental setup. In Section 4 we present the results of our experimental study of the accuracy to predict the time to failure. Section 5 presents the related work; and, finally, Section 6 concludes the paper.

2 Our modelling assumptions and prediction strategy

In a perfect and easy world, resource exhaustion due to software aging or to some other factor would be a linear phenomenon with respect to time. The time until failure could be then predicted easily by:

$$T_{fail} \simeq \frac{Rmax_i - R_{i,t}}{s_i}, \tag{1}$$

Where $Rmax$ is the maximum available amount of resource i, $R_{i,t}$ is the amount of resource i used at instant t, and s_i is the consumption speed per second. However, this is a very simplistic approach. First, in this approach we are assuming consumption rate is constant along the time. Moreover, we assume we know the resource involved on the software aging a priori.

But software aging could have several unknown reasons. It could originate from an interaction of several resources, or we might not be monitoring all the significant metrics from the system, or consumption rate could change over time, and could also depend on the, possibly changing, workload. Or even, the perspective used in monitoring the resource is not the appropriate one; this is mainly true when we are working with virtualized resources. On the other hand, the system actions themselves (such as garbage collection) can mask or mitigate the software aging process, adding more life time to the system.

2.1 Motivating Examples

To understand the complexity and problems when building a resource consumption model, we describe two examples that we found when we tried to model the Java Memory exhaustion of a J2EE application server.

2.1.1 Example 1: Nonlinear Resource Behavior

Given a deterministic and progressive software aging that consumes a resource at constant rate, our first approach to predict the time until resource exhaustion would compute the rate (constant in this case) and apply the formula above. One simple method to automatically obtain this slope is Linear Regression. Linear regression has been used in several works, like [15], to predict the resource consumption under normal circumstances. It is a powerful tool in the area of capacity planning. However, linearity is a strong assumption. Even if the resource consumption is linear a priori, the system may exhibit nonlinear behaviors.

For example, we performed an experiment in which we injected memory leaks at regular rates under a constant workload in a Java Application (a Tomcat web server, as we described in next section). In Figure 1 we can observe the memory actually used during the experiment with a progressive memory consumption and constant workload (dark line). We let the application run until the server fails due to memory exhaustion. In a few words, we observe that even if our memory leak injection has constant rate, the complexity of the underlying system introduces a nonlinear behavior.

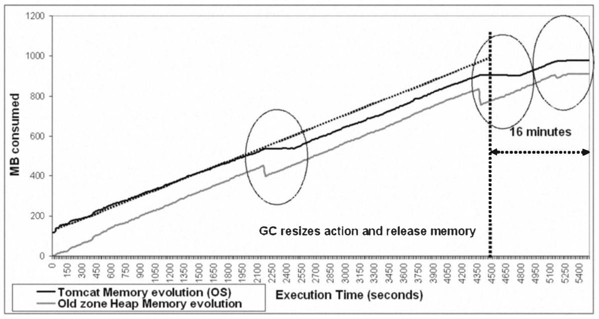

Figure 1. Progressive memory consumption of the Java Application.

In more detail: In Java applications, the Java Heap Memory is divided into three main zones: Young, Old and Permanent. When a Java Object is created, it is stored in the Young zone. When the Young Zone is full, alive objects are moved to the Old zone, that for objects that have been alive for a long time. The default Heap management system defines an initial size for this zone, a fraction of the maximum memory available for the application. When the Old zone is full, the Heap Management System resizes it, allocating more memory to it if available. In Figure 1 we observe this resizing (grey line) in three moments of the execution: at 2150 seconds, at 4350 seconds, and (less visible) at 5150 seconds. At the same time that the Old zone is resized, some objects are moved to the Permanent Zone, or are freed if they are not referenced by others. We can see that the monitoring perspective is crucial here. In the figure, the dark line is the percentage of memory used from the system perspective. The Heap resizes the Old zone and releases some part of the memory allocated by the application. The perspective from the system level perspective (dark line) is that the Java application is using a constant amount of memory for a while, but that is only part of the truth: the application has released a part of memory (grey line) but the system does not yet see that fact. We can observe that fact only if we monitor the Heap internal behavior, hence obtaining a more accurate system description. We come back to this interesting fact in the next example.

In the example, the normal Heap behavior allows the application to run for about 16 extra minutes, over what we would predict from the initial consumption rate. Furthermore, this figure is strongly dependent on the aggressiveness of the memory leak and the workload; less aggressive leaks or lighter loads increase this extra time, hence the prediction error of a naive prediction strategy.

2.1.2 Example 2: Different Viewpoints on a Resource

As discussed in the previous example, resource behavior can look quite different depending on our monitoring strategy. This is most dramatic when we are working with virtualized resources, such as (in some sense) the Java Heap Memory. Memory usage by a Java application looks quite different if we monitor at the operating system (OS) level or the Java Virtual Machine (JVM) level.

We conducted a simple experiment to show this duality. We run a web application (presented in Section 4 in detail) under a constant workload. We have modified the application to force three different phases in the application execution. Every phase lasts 20 minutes. The application has a normal behavior for 20 minutes, after which it consumes memory abnormally during 20 minutes, and after which the application releases the memory acquired during the previous phase, returning to the initial state. We repeat this periodic behavior/pattern every hour during 5 hours. However, this periodic and constant workload is not shown by a simple monitoring over the memory used by the Java application from the OS perspective. In Figure 2 we present both perspectives of the same resource during the same experiment. The waves (the grey line) represent the sum of the memory used by the Young and Old zones. The Permanent zone is not depicted because it is constant during the experiment. We can observe how the memory allocated by the Java application looks constant from the OS perspective (dark line). In a Linux system, when an application frees up some memory, the system does not recover this memory automatically: it only recovers it when required by other applications. Due this behavior, if we monitor the OS memory consumed by an application it may look constant along time, but if we observe the Java Heap Memory, the application is releasing and consuming memory.

These examples indicate the difficulty of building an accurate prediction system. We need detailed and sophisticated monitoring tools to obtain detailed information on the resources. Even having all the metrics, we need a lot of human expertise to decide which are the relevant ones, and to actually build the model taking into account the system's

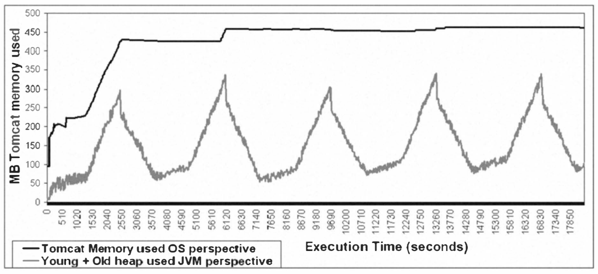

Figure 2. The Tomcat memory consumption from system and heap perspectives

complex behavior. This is probably unaffordable in most cases, and anyway useless in ever-changing environments where hardware and software are updated frequently. For this reason, ML and Data Mining seems as an alternative (or at least, a complement) to explicit expert modeling. ML can be used to automatically build models of complex systems from a set of (possibly tens or hundreds of) apparently independent metrics.

2.2 Prediction Assumptions

Our proposal is thus to use ML to predict time until failure due to software aging. Due to the complexity of modeling these growing complex environments and with low knowledge a priori about them, we decide to use ML to build automatically the model from a set of metrics easily available in any system like CPU utilization, system memory, application memory, Java memory, threads, users, jobs, etc. Thanks to the potential of ML to learn from previous executions what are the most important variables to take into account to build the model. In fact, software aging could be related with other reasons, besides resource exhaustion. The technique used in this paper could predict the crash due to software aging if we collect the metrics (from resources or not) related with the software aging.

Note that our approach may be valid when an approaching failure is somehow foreseeable from the system metrics, not for sudden crashes that happen with no warning. These would require completely different techniques, such as static analysis of the code to reveal dangerous logical conditions, which we do not address here.

Among many ML algorithms and models available, we have chosen to use the one called M5P [16], included in the popular WEKA [17] ML and data mining package, at this stage of our work. An M5P model consists of a binary *decision tree* whose inner nodes are labelled with tests of the form "variable $<$ value?", and each leaf is labelled with a linear regression model (possibly using all variables). The rationale is that while a global behavior may be highly nonlinear, it may be composed (or approximate by) a reasonable number of linear patches, i.e., it may be piecewise linear. This may well be the case for many system behaviors of the kind we want to analyze, where the system may be in one of a relatively small number of phases, each of which is essentially linear. In [14] we performed a preliminary comparison of M5P with linear regression alone, and decision trees alone, in a simple failure prediction scenario and concluded that it indeed performed much better.

Our model will be trained using failure executions and tested using different executions from the training set to validate the accuracy of the model build by ML. Our idea is that our model predicts time until failure if the state of the system (including workload) does not vary in the future. However, if the situation changes (the consumption speed changes) the model has to be able to recalculate the time until failure under the new circumstances. The M5P training process does not really take into account the absolute value of the times, but the number of checkpoints (instances) we collect. If the resource degrades 100 times slower than in other environment, but we still measure the same number of checkpoints, the measurements of the degrading resource will be the same, only spaced apart 100 more times, M5P will build the same model, and the predictions will be equally good.

For this reason, we added a set of derived metrics as a variables to achieve a more accurate prediction. The most important variable we add is the consumption speed from every resource under monitoring (threads, system memory, web application memory and every Java Heap zone: Young and Old). To calculate the consumption speed, we decided to use an *average speed* to avoid too much fluctuations in the measure. We decided to use a sliding window average (also called moving average). The sliding window average collects the last X speed observations from the resource and calculates their average, so as to smooth out noise and fluctuation. The choice of X is a certain trade-off: a long window is more noise tolerant, but also makes the method slower to reflect changes in the input. It must be set by considering the expected noise and the frequency of change in our scenario.

Using this sliding window and M5P we have conducted a set of experiments to evaluate the effectiveness of our approach for complex software aging scenarios. We have used Mean Absolute Error (MAE) to measure our prediction accuracy: this is the average of the absolute difference between true values and predicted values. So, we are using absolute errors. An error of 200 seconds over a time-to-failure of 1000 seconds is not equivalent to an error of 2 minutes over 10 minutes. However, predicting exactly the time until failure is probably too hard, even as a baseline. We have used another measure called the Soft Mean Absolute Error (S-MAE): We decided that if the model predicts

Table 1. Machine Description

	Clients and DBServer	App. Servers
Hardware	2-way Intel XEON 2.4 GHz with 2 GB RAM	4-way Intel XEON 1.4 GHz with 2 GB RAM
Operating System	Linux 2.6.8-3-686	Linux 2.6.15
JVM	-	jdk1.5 with 1GB heap
Software	TPC-W Clients/MySQL 5.0.67	Tomcat 5.5.26

a time until crash within a margin of 10% of the real time until crash (named *security margin*), we count it as zero error. For example, if the real time until crash is 10 minutes, we assume 0 error if the model predicts between 11 minutes and 9 minutes. If the system predicts say 13 (or 7) minutes, we would count a 2-minute error (the absolute error). Of course, thresholds other than 10% are possible. It is clear that S-MAE is always smaller than MAE.

Finally, we have trained our model to be more accurate when the crash is coming. For this reason, we have calculated the MAE for the last 10 minutes of every experiment (POST-MAE) and for the rest of experiment (PRE-MAE). The idea is that our approach has to have lower MAE in the last 10 minutes than the rest of experiment, showing that the prediction becomes more accurate when it is more needed.

3 Experimental Setup

In this section we describe the experimental setup used in all experiments presented below, whose main goal is to evaluate the effectiveness of the prediction approach. The experimental environment simulates a real web environment, composed by the web application server, the database server and the clients machine. The analysis subsystem and planning subsystem are in an external centralized machine; in a real environment, however, the best option would be to have the analysis subsystem distributed among nodes and only the planning subsystem centralized to make decisions using the information from all nodes. Finally, to simulate the client workload we have a machine with client simulator installed.

In our experiments, we have used a multi-tier e-commerce site that simulates an on-line book store, following the standard configuration of TPC-W benchmark [18]. We have used the Java version developed using servlets and using as a MySQL [19] as database server. As application server, we have used Apache Tomcat [20]. TPC-W allows us to run different experiments using different parameters and under a controlled environment. These capabilities allow us to conduct the evaluation of our approach to predict the time until failure. Details of machine characteristics are given in Table 1.

TPC-W clients, called Emulated Browsers (EBs), access the web site (simulating an on-line book store) in sessions. A session is a sequence of logically connected (from the EB point of view) requests. Between two consecutive requests from the same EB, TPC-W computes a thinking time, representing the time between the user receiving a web page s/he requested and deciding the next request. In all of our experiments we have used the default configuration of TPC-W. Moreover, following the TPC-W specification, the number of concurrent EBs is kept constant during the experiment.

Table 2. Variables used in every experiment to build the model

	Exp 4.1	Exp 4.2	Exp 4.3	Exp 4.4
Throughput(TH)	X	X	X-X [a]	X
Workload	X	X	X-X	X
Response Time	X	X	X-X	X
System Load	X	X	X-X	X
Disk Used	X	X	X-X	X
Swap Free	X	X	X-X	X
Num. Processes	X	X	X-X	X
Sys. Memory Used	X	X	X-	X
Tomcat Memory Used	X	X	X-	X
Num. Threads	X	X	X-X	X
Num. Http Connections	X	X	X-X	X
Num. Mysql Connections	X	X	X-X	X
Max. MB Young/Old (2)[b]		X	X-X	X
MB Young/Old Used (2)		X	X-X	X
% Used Young/Old Used (2)		X	X-X	X
SWA[c] Young/Old variation(2)		X	X-X	X
SWA variation (3)[d]	X	X	X-[h]	X
SWA variation /TH (2)[e]	X	X	X-	X
SWA variation /TH (2)[f]		X	X-X	X
1/SWA (3)[d]	X	X	X-[h]	X
1/SWA (3)[f]		X	X-X	X
Young/Old Used/SWA (2)		X	X-X	X
Resource Used(R)/SWA (3)[d]	X	X	X-[h]	X
(1/SWA variation)/TH (2)[e]	X	X	X-	X
(1/SWA variation)/TH (2)[f]		X	X-X	X
(R/SWA variation)/TH (2)[e]	X	X	X-	X
(R/SWA variation)/TH (2)[f]		X	X-X	X
SWA Resource Used (4)[g]	X	X	X-X	X
Time to Failure	X	X	X-X	X

[a] Exp. 4.3 Complete-Exp. 4.3 Feature Selection
[b] (X) number of variables represented
[c] Sliding Window Average (SWA)
[d] For Num. Threads, Tomcat Mem. Used and System Mem. Used
[e] For Tomcat Memory Used and System Memory Used
[f] For Young Zone Used and Old Zone Used
[g] For Response Time, Throughput, System Memory Used and Tomcat Memory Used
[h] Removed only Tomcat Memory Used and System Memory Used variables related

To simulate the aging-related errors consuming resources until their exhaustion, we have modified the TPC-W implementation. In our experiments we have played with two different resources: Threads and Memory, individually or merged. To simulate a random memory consumption we have modified a servlet (*TPCW_search_request_servlet*) which computes a random number between 0 and N. This number determines how many requests use the servlet before the next memory consumption is injected. Therefore, the variation of memory consumption depends of the number of clients and the frequency of servlet visits. According to the TPC-W specification, this frequency depends on the workload chosen. This makes that with high workload our servlet injects quickly memory leaks, however with low workload, the consumption is lower too. But, again, the average consumption rate would depend on the average of this random variable, with fluctuations that become less relevant when averaged over time. Therefore, we could thus simulate this effect by varying N, and we have decided to stick to only one relevant parameter, N. On the other hand, to simulate a thread consumption in the servlet we use two parameters: T and M. At every injection, the system injects a random number of threads between 0 and M, and determines how much time occurs until the next injection, a random number (in seconds) between 0 and T. Thread injection is independent of the workload (since injection occurs independently of the running applications), while memory injection is workload dependent (because it occurs when a certain application component is executed). These two errors help us to validate our hypothesis under different scenarios. TPC-W has three types of workload (Browsing, Shopping and Ordering). In our case, we have conducted all of our experiments using shopping distribution.

4. Prediction Experiment Results

Table 2 presents the variables used to build every model used in every experiment conducted. In every experiment, we indicate the size (number of nodes and leafs) of the model and the number of instances used to train the model. Moreover, [21] links to the training and test datasets used in our experiments.

4.1 Deterministic Software Aging

Our first approach was to evaluate M5P to predict the time until failure due to deterministic software aging. We decided to inject a 1MB of memory leak with $N = 30$ (see experimental setup). We trained our model, generated using M5P, with previous 4 executions with 25 EBs, 50EBs, 100EBs and 200EBs, becoming 2776 instances. The M5P model generated was composed by a tree with 33 Leafs and 30 inner nodes, using 10 instances to build every leaf. In this experiment, we did not add the heap information. The four training experiments were executed until the crash of Tomcat, to let the M5P to learn the behavior of the system under a deterministic software aging. Finally, to evaluate the accuracy of the model, we evaluated the model built with these four experiments using two new experiments with different workload (75EBs and 150EBs).

Table 3. MAEs obtained in Exp. 4.1

	Lin. Reg	M5P
75EBs MAE	19 min 35 secs	15 min 14 secs
75EBs S-MAE	14 min 17 secs	9 min 34 secs
150EBs MAE	20 min 24 sec	5 min 46 secs
150EBs S-MAE	17 min 24 secs	2 min 52 secs
75EBs PRE-MAE	21 min 13 secs	16 min 22 secs
75EBs POST-MAE	5 min 11 secs	2 min 20 secs
150EBs PRE-MAE	19 min 40 secs	6 min 18 secs
150EBs POST-MAE	24 min 14 secs	2 min 57 secs

In Table 3 we present the results obtained. We can observe how M5P obtains better results than simple linear regression due because it handles better the trend changes due to the Heap Memory Management actions, even when we do not add the specific information, as in the examples in Section 3.

4.2 Dynamic and Variable Software Aging

Our next experiment was to evaluate our model to predict progressive but dynamic software aging under constant workload. We trained the model with 4 executions (1710 instances): one hour execution where we did not inject any memory leak and three executions where we injected 1MB memory leak with constant ratio ($N = 15$,$N = 30$ and $N = 75$ in every respective execution, see the experimental setup). Using these only 4 executions we trained the model. The model generated was composed by 36 leafs and 35 inner nodes, using 10 instances to build every leaf. The model was trained to determinate as an infinite time until crash as 3 hours (10800 secs) using the training data set without injection. This model was tested over an experiment where we changed the ratio every 20 minutes. During the first 20 minutes we did not inject any memory leak. After that, during the next 20 minutes we injected a memory leak following a ratio of $N = 30$. After that, we increase the injection to $N = 15$ and finally, 20 minutes later, we reduced the software aging following a $N = 75$ and we left constant this ratio until crash. In order to compute the MAE and S-MAE, we fix the current injection rate and then simulate the system until a crash occurs. So, we wanted to evaluate the capability of the model to overcome software aging trend changes and react before it, so if the consumption speed goes down, the time until exhaustion increases and vice versa.

Figure 3 presents the predicted time (dark line) vs. Tomcat memory evolution (grey line) during the execution. First, we have trained our model to declare that the time until crash is 3 hours (standing for "very long" or "infinite") when there is no aging; indeed, during the first 20 minutes

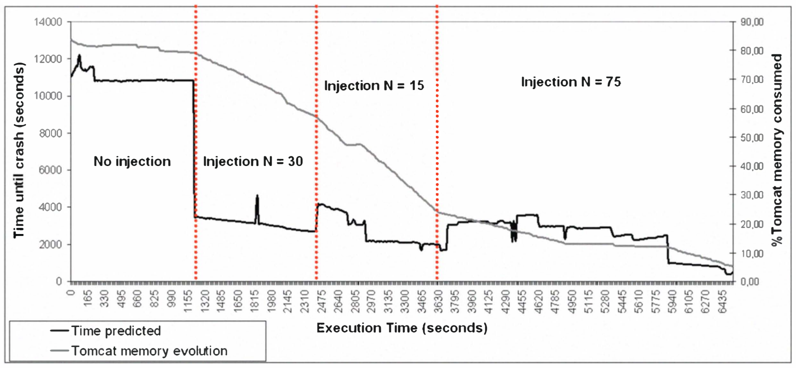

Figure 3. Time predicted vs. Tomcat Memory Evolution

the model predicts a 3-hour time-to-failure, meaning that no aging is occurring. After 20 minutes, we start injecting; we can see that the Tomcat memory starts decreasing gradually but predicted time until crash decreases drastically. The reason to this drastic adaptation is due to the construction of M5P. The starting injection phase provokes a change of branch of the tree from non-aging ($>$ 3 hours to crash) to aging branch, based on aging trend. After another 20 minutes, the injection rate increases. The sliding window introduces some delay in detecting that (in this case, 12 marks, so 12 marks * 15 seconds per mark, 180 seconds). But a more important fact happens in the beginning of the third phase of the experiment: A flat zone provoked by the Heap Management process, as discussed before.

Because of this, the model starts to predict more time that the real time, because the model is seeing a lower-than-real consumption rate, which translates to a larger-than-real prediction. However, when the flat region ends (as visible in the the Tomcat memory plot), the model reacts quickly to adapt the time until crash close to the real value (around 2850 seconds after the start of the experiment). In the fourth phase, we reduce the rate and the model quickly adapts the time predicted to the new circumstances, increasing the time until crash. Prediction in this region is not quite accurate, though: as the injection rate is so slow, the model has trouble detecting it, and it keeps the prediction almost constant. Furthermore, we can observe a second resizing (from 4850 seconds to 5900 seconds). During this phase, again the model is not accurate (as it does see a constant Tomcat memory usage), but when the Tomcat memory is out of the flat zone, again the model reacts reducing the time until crash. This behavior shows the adaptability to changes of M5P. The MAE and S-MAE obtained by M5P in this scenario was 16 min. 26 secs. and 13 min. 3 secs. respectively, which we believe is a quite reasonable accuracy; on the other hand, Linear Regression has a really unacceptable MAE. On the other hand the PRE-MAE and POST-MAE were, respectively, 17 min. 15 secs. and 8 min. 14 secs. The experiment was running for 1 hour and 47 minutes.

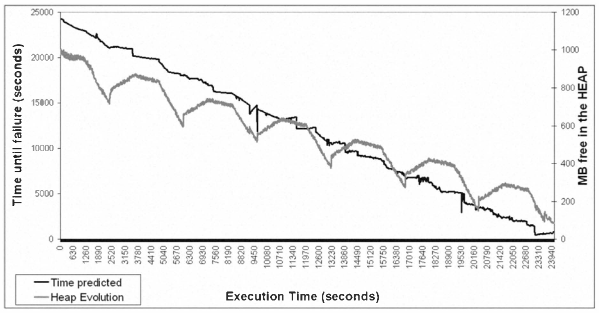

Figure 4. Time predicted vs. Java Heap Tomcat Memory Evolution

4.3 Software Aging Hidden within Periodic Pattern Resource Behavior

After evaluating the effectiveness of M5P to predict dynamic software aging, our next step was to evaluate the model in front of a deterministic software aging masked by a periodic pattern of memory acquisition followed by memory release. Aging here means that not all memory allocated during the memory acquisition phase is later released, so memory leaks accumulate over time. Our experiment was the same we conducted in the second motivating example. But now we modify the release phase to guarantee that after these 20 minutes some memory was retained, so a crash was bound to happen after several periodic phases. The workload was constant with 100EBs. As we can observe, the memory leak in fact is quite constant but we introduce a periodic behavior pattern introducing some noise (the released phases and non-injection phases). The injection phase follows $N = 30$ and release phase follows $N = 75$, allocating and releasing 1MB each time. We trained the model using the same training set as in Section 5.2. So, the train-

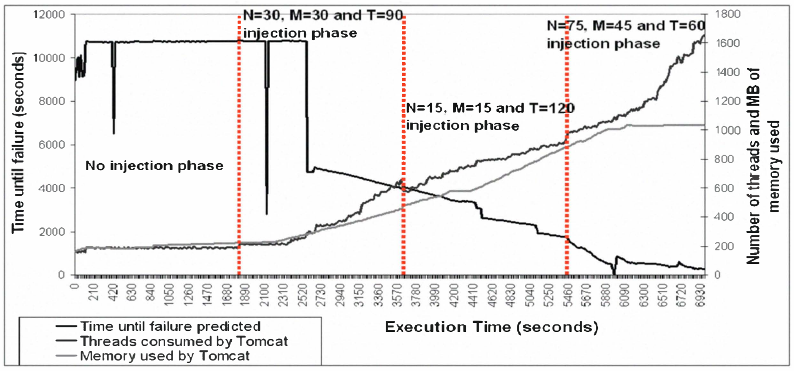

Figure 5. Time predicted and resource evolution during the two-resource experiment

Table 4. MAEs obtained in Exp 4.3

	Lin Reg	M5P
MAE	15 min 57 secs	3 min 34 secs
S-MAE	4 min 53 secs	21 secs
PRE-MAE	16 min 10 secs	3 min 31 secs
POST-MAE	8 min 14 secs	5 min 29 secs

ing set does not have any execution with release phase or periodic patterns. Our idea is that M5P can manage the periodic pattern and extract from that, the real trend (the random injection between 0 to N). However, we obtained poor results in our first approach, and after some inspection we noted that the model was paying too much attention to irrelevant attributed. We decided to apply an expert feature/variable selection following the conclusion extracted in [22], where the authors concluded that selection of a good subset of variables increases the prediction accuracy. We decided to re-train the model only with the variables related with the Java Heap evolution. The new model was formed by 17 inner nodes and 18 leafs. In Figure 4 we can observe clearly how the time until failure is linear and how the Java Heap memory is being consumed and released in every phase until exhaust the memory resource. Table 4 shows the MAE and S-MAE, PRE-MAE and POST-MAE obtained by M5P and the Linear Regression. We note that M5P can manage the periodic pattern, unlike Linear Regression which obtains much worse results even after variable selection. However, we have to notice that, in this case, M5P has problems to be accurate in the last 10 minutes of the experiment.

4.4 Dynamic Software Aging due to Two Resources

After evaluating our approach in front of aging due to a single resource, our next step was to consider aging caused by two resources simultaneously. The two resources involved in the experiment were Memory and Threads, and every phase was around 30 minutes. The experiment was conducted as follows: we have a first phase with no injection (both resources), after that we start to inject both resources: the memory rate injection was $N = 30$, while the thread rate injection was $M = 30$ and $T = 90$. After that, we increased the memory rate injection to $N = 15$ and the thread rate injection was reduced to $M = 15$ and $T = 120$. After that, in the last phase, we change again both rates, reducing the memory rate ($N = 75$) and increasing the thread rate injection ($M = 45$ and $T = 60$).

One important point in this experiment is that when a Java Thread, a system thread is assigned to the Java thread until it dies. Other important point is that every Java Thread has an impact over the Tomcat Memory, because the Java thread consumes Java memory by itself. So, although the two causes of aging are unrelated on the surface, they are related after all; we believe this may be a common situation, and that it may easily go unnoticed even to expert eyes.

In Figure 5 we can observe the thread consumption and memory consumption evolution during the experiment. We can observe clearly the four phases of the experiment. The MAE and S-MAE, PRE-MEA and POST-MAE obtained by M5P in this experiment was: 16min. 52secs., 13min. 22secs, 18min. 16secs. and 2min. 5 secs. respectively: this is about 10% error (MAE and S-MAE), given that the experiment took 1 hour and 55 minutes until crash. We can observe how the M5P is able to predict with great accuracy the time to crash, when it is near. Most importantly, *the model never was trained using executions where both resources were injecting errors simultaneously*. We trained the model with several executions at different (constant) workload and different (constant) injection rates ($N = 15, 30, 75$, $M = 15, 30, 45$ and $T = 60, 90, 120$, so 6 executions, 2752 instances), but in all of them only one resource involved in the execution. The model generated was composed by 35 inner nodes and 36 leafs. These results show the promising adaptability of this approach to new situations not seen during previous training.

Finally, we want to remark another interesting point. After conducted all of the experiments presented before, we decided to inspect the models generated by M5P in every experiment. We observed the model could give clues to determine the root cause of failure. For example, in the last experiment, we observed the tree built by M5P, where the root node contains the system memory attribute if the used system memory is over 1306 MB, the second variable inspected is the number of threads in the system. But if system memory is below 1306 MB, the most relevant variable is Tomcat Memory and the only third is the number of threads. Only with the first two levels of the tree we can observe how memory usage and the threads are important variables, which gives administrators or developers a clue on the root cause of the failure due to software aging. So, interpreting the models generated via ML models has an additional interest besides prediction.

5. Related Work

The idea of modeling resource consumption and forecast system performance from here is far from new. A lot of effort along this line has been concerned with capacity planning. In [23], an off-line framework is presented to develop performance analysis and post-mortem analysis of the causes of Service Level Objective (SLO) violations. It proposes the use of TANs (Tree Augmented Naive Bayesian Networks), a simplified version of Bayesian Networks (BN), to determine which resources are most correlated to performance behavior. In [24], Linear Regression is used to build an analytic model for capacity planning of multi-tier applications. They show how Linear Regression offers successful results for capacity planning and resource provisioning, even under variable workloads.

Other works such as [25, 26] present different techniques to predict resource exhaustion due to a workload in a system that suffers software aging. In these two works they present two different approaches: in [25], authors use a semi-Markov reward model using the workload and resource usage data collected from the system to predict resource exhaustion in time. In [26], authors use time-series ARMA models from the system data to estimate the resource exhaustion due to workload received by the system. However, these works assume a general trend of the software aging, not a software aging that could change with time. Other important difference with our work is that they apply the model over the resource involved with the software aging, but our approach is more generic because we allow ML to learn, by itself, what is the resource (or resources) involved in the software aging.

Other interesting approach was presented in [27], the authors present an evaluation of three well-known ML algorithms: Naive Bayes, decision trees and support vector machines to evaluate their effectiveness to model and predict deterministic software aging. However, the authors didn't test their approach against a dynamic and more than one resource involved in order to evaluate the effectiveness of their approach.

Some interesting works have addressed the important task of predicting failures and critical events specifically in computer systems. In [28], the authors present a framework to predict critical events in large-scale clusters. They compare different time-series analysis methods and rule-based classification algorithms to evaluate their effectiveness when predicting different types of critical events and system metrics. Their conclusion is that different predictive methods are needed according to the element that we want to predict. But, as we showed M5P could obtain good results, even with more than one resource involved. [15] presents an on-line framework that determines whether a system is suffering an anomaly, a workload change, or a software change. The idea is to divide the sequence of recorded data into several segments using the Linear Regression error. If for some period it is impossible to obtain any Linear Regressions with acceptable error at all, the conclusion is that the system is suffering some type of anomaly during that period. Their approach is complementary with ours, because the underlying assumption is that, except on transient anomalies and between software changes, the system admits a static model, one that depends on the workload only and does not degrade or drift over time. On the other hand, we concentrate on systems that can degrade, i.e., for which a model valid now will not be valid soon, even under the same workload.

6. Conclusions

We have proposed a ML approach to build automatically models from system metrics available in any system. We decided to use ML due to the complexity of the resources behavior and the complexity of the environments. We have based our approach in feeding our model with a set of derived variables where the most important variable is the consumption speed of every resource, which is smoothed out using averaging over a sliding window of recent instantaneous measurements. We have evaluated our approach to predict the effect of software aging errors which gradually consume resources until its exhaustion, in a way that cannot be attributed to excess load. We have conducted a set of experiments to evaluate the M5P in different and complex software aging scenarios. Our experiments show how M5P obtains acceptable accuracy in a variety of dynamic scenarios. Moreover, M5P has showed its ability of adapting scenarios never seen during training. Finally, we have suggested a new potential approach to help to determine the root cause software aging, interpreting the models generated by M5P. On the other hand, in [29], we present an extended version of this paper, presenting a framework for predicting in real time the time-to-crash of web applications which suffer from software aging, using ML techniques. Our framework allows recovery of the potentially crashing server using a clean automatic recovery and avoiding losses of new and on-going user requests.

Several issues and challenges are still open. We want to work on to know what is the best moment to apply the recovery action, which has the responsibility to understand the prediction information and take the most effectiveness action to maximize the server utilization. In the future, we also we want to investigate further Artificial Intelligence techniques to combine human expertise with information from the predictors. Anther idea we want to evaluate is to build a prediction board with a set of prediction models to reach a consensus to increase the prediction accuracy.

Acknowledgment

This research work has been supported by the Spanish Ministry of Education and Science (projects TIN2007-60625 and TIN2008-06582-C03-01), EU PASCAL2 Network of Excellence, and by the Generalitat de Catalunya (2009-SGR-980 and 2009-SGR-1428).

References

[1] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. *Why do internet services fail, and what can be done about it?*.In 4th USENIX Symposium on Internet Technologies and Systems (USITS'03), 2003.

[2] S. Peret and P. Narasimham. *Causes of Failure in Web applications*. TR CMU-PDL-05-109, Carnegie Mellon Univ, 2005.

[3] MemProfiler *http://memprofiler.com/*

[4] Parasoft Insure++ *http://www.parasoft.com*

[5] K. S. Trivedi, k. Vaidyanathan and K. Goseva-Popstojanova. *Modeling and Analysis of Software aging and Rejuvenation*. IEEE Annual Simulation Symposium, April 2000.

[6] T. Dohi, K. Goseva-Popstojanova and K. S. Trivedi. *Analysis of Software Cost Models with Rejuvenation*. IEEE Intl. Symposium on High Assurance Systems Engineering (HASE 2000).

[7] M. Grottke, R. Matias Jr. and K.S. Trivedi *The Fundamentals of Software aging* In Proc. 1st Int. Workshop on Software Aging and Rejuvenation. 19th Int. Symp. on Software Reliability Engineering, 2008.

[8] Apache *http://httpd.apache.org/docs/*

[9] A.Tai, S.Chau, L.Alkalaj and H.Hecht. *On-board Preventive Maintenance: Analysis of Effectiveness an Optimal Duty Period*. Proc. 3rd Workshop on Object-Oriented Real-Time Dependable Systems, 1997.

[10] E. Marshall *Fatal Error: How Patriot Overlooked Scud*. Science, p. 1347, Mar.1992

[11] K.Vaidyanathan and K.Trivedi *A Comprehensive Model for Software Rejuvenation*. IEEE Trans. On Dependable and Secure Computing, Vol, 2, No 2, April- 2005

[12] G.Candea, E.Kiciman, S.Zhang and A.Fox *JAGR: An Autonomous Self-Recovering Application Server*. Proc. 5th Int Workshop on Active Middleware Services, Seattle, June 2003

[13] K.Cassidy, K.Gross and A.Malekpour. *Advanced Pattern Recognition for Detection of Complex Software Aging Phenomena in Online Transaction Processing Servers*. Proc. of the Int. Conf. on Dependable Systems and Networks, DSN-2002.

[14] J. Alonso, R. Gavaldà, and J. Torres *Predicting web server crashes: A case study in comparing prediction algorithms*. Procs. Fifth Intl. Conf. on Autonomic and Autonomous Systems (ICAS 2009), April 20-25, Valencia, Spain, 2009.

[15] L. Cherkasova, K. Ozonat, N. Mi, J. Symons, E. Smirni *Anomaly? Application Change? or Workload Change? Towards Automated Detection of Application Performance Anomaly and Change*. Procs. 38th Annual IEEE/IFIP Conf. on Dependable Systems and Networks, DSN'2008, June 24-27.

[16] Y. Wang and I. H. Witten *Inducing Model Trees for Continuous Classes*. In Proc. of the 9th European Conf. on Machine Learning Poster Papers, 1997.

[17] Weka 3.5.8 *http://www.cs.waikato.ac.nz/ml/weka/*.

[18] TPC-W Java Version *http://www.ece.wisc.edu/~pharm/*.

[19] MySQL Data Base server *http://www.mysql.com/*.

[20] Apache Tomcat Server *http://tomcat.apache.org/*

[21] Training and Test Datasets in WEKA format *http://alonso.site.upc.edu/research/DSN-2010results.html*

[22] G.A. Hoffmann, K.S. Trivedi, and M. Malek *A best practice guide to resource forecasting for the Apache Webserver*. IEEE Transactions on Reliability 56, 4 (Dec 2007), 615-628.

[23] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. S. Chase *Correlating instrumentation data to system states: A building block for automated diagnosis and control*. In Proc. 6th USENIX OSDI, San Francisco, CA, Dec. 2004.

[24] Q. Zhang, L. Cherkasova, N. Mi, and E. Smirni *A regression-based analytic model for capacity planning of multi-tier applications*. Cluster Computing (2008), vol 11: 197-211.

[25] K. Vaidyanathan and K.S. Trivedi *A Measurement-Based Model for Estimation of Resource Exhaustion in Operational Software Systems*. In Proc. of the 10th Intl. Symp. on Software Reliability Engineering, 1999.

[26] L. Li, K. Vaidyanathan, and K. S. Trivedi *An Approach for Estimation of Software Aging in a Web Server*. In Proc of Intl. Symp. Empirical Software Engineering, 2002.

[27] A. Andrzejak and L.M. Silva *Using Machine Learning for Non-Intrusive Modeling and Prediction of Software Aging*. In Procs of IEEE Network Operations and Management Symposium, 2008. NOMS 2008.

[28] R. K. Sahoo, A. J. Oliner, I. Rish, M. Gupta, J. E. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam *Critical Event Prediction for Proactive Management in Large-scale Computer Clusters*. In KDD, pages 426435, August 2003

[29] J. Alonso, J-Ll. Berral, R. Gavaldà and J. Torres *Adaptive on-line software aging prediction based on Machine Learning*. TR UPC-DAC-RR-CAP-2010-4, http://alonso.site.upc.edu/, 2010.