Proactive Software Rejuvenation Based on Machine Learning Techniques

Dimitar Simeonov[1] and D.R. Avresky[2]

[1] IRIANC
simeonov.dimitar@gmail.com
[2] IRIANC
autonomic@irianc.com

Abstract. This work presents a framework for detecting anomalies in servers leading to crash such as memory leaks in aging systems and proactively rejuvenating them.

Proactive VM-rejuvenation framework has been extended with machine learning techniques. Utilization of the framework is allowing the effect of software failures virtually to be reduced to zero downtime. It can be applied against internal anomalies like memory leaks in the web servers and external as Denial of Service Attacks. The framework has been implemented with virtual machines and a machine learning algorithm has been realized for successfully determining a decision rule for proactively initiating the system rejuvenation. The proposed framework has been theoretically justified and experimentally validated.

Keywords: proactive rejuvenation, virtualisation, machine learning techniques, feature selection, sparsity, software aging (memory leaks), validation.

1 Introduction

All computer systems may fail after some amount of time and usage. This is especially true for web servers. The availability is one of the most important characteristic of the web servers. Computer systems, which are prone to failures and crashes, can be realized with a higher availability if their mission critical parts are replicated. There are many practical examples of such systems - RAID, e-mail servers, computing farms. In this paper, it is shown how the software replication and rejuvenation can be used for increasing the availability of a software application with a critical workload. Software replication and rejuvenation can be performed by virtual machines easily, cheaply and effectively. The virtualization allows us to create a layer of an abstraction between software and hardware, which provides some independence of the underlying hardware.

Any anomalies, with a similar behavior that are leading to a system's crash, can be effectively predicted by a machine learning algorithm. For example, memory leaks exhibit a similar behavior every time they occur, and therefore, such behavior can be predicted with a high accuracy. With an accurate prediction and an efficient recovery mechanism, the software system's availability can be increased significantly.

2 Related Work

Different methods and models have been presented for estimating software aging in web servers and resource exhaustion in operational software systems in [17], [18] and [19]. Software rejuvenation has been introduced as an efficient technique for dealing with this problem in [14] and further developed in [23]. Virtualization has been effectively used in [1] for improving software rejuvenation. Virtual machines are widely used for increasing the availability of web servers [16]. In [5], [11] and [24] different techniques for increasing availability of complex computing systems have been introduced. Recently, a comprehensive model for software rejuvenation has been developed for proactive detection and management of software aging ([15], [17], [20], [21] and [22]). Different techniques for analyzing the application performance due to anomalies for enterprise services are presented in [6], [10], [12] and [13].

In this paper a comprehensive method for a proactive software rejuvenation for avoiding system crashes due to anomalies, such as memory leaks, is presented. It is theoretically justified and experimentally validated. Based on the training data, obtained by the proposed framework, a close predictor of the actual remaining time to crash of a system has been accurately estimated. Such prediction has been used as a decision rule for initiating software rejuvenation.

3 Proactive VM-Rejuvenation Framework

The VM-REJUV framework has been developed in [1] in attempt to solve the problem of aging and crashing web servers. Current paper proposes an extension to the VM-REJUV framework that allows to predict the right time for activating the rejuvenation mechanism.

The VM-REJUV framework consists of three virtual machines called for simplicity VM1 (VM-master), VM2(VM-slave) and VM3(VM-slave). VM1 contains the controlling mechanism of the application. VM2 and VM3 are identical and contain the application susceptible to anomalies. VM1 is like a mini-server to which VM2 and VM3 are connected. They regularly send information about their parameters to VM1. This information is analyzed and only one of VM2 and VM3 is allowed to be active. VM1 activates the spare copy to become active and to start handling the workload when the active machine will be crashing soon or stop reporting data.

The VM-REJUV framework can be extended into Proactive VM-rejuvenation framework to contain an arbitrary number of virtual machines with the functionality of VM2 and VM3. Figure 1 shows the organization of Proactive VM-rejuvenation framework.

3.1 VM-Master and VM-Slave Components and Communication

VM-master needs to be always on. It creates a local server to which VM-slaves are connected. Each VM-slave can be in one of the possible states: starting up, ready, active and rejuvenating. All virtual machines have the following properties:

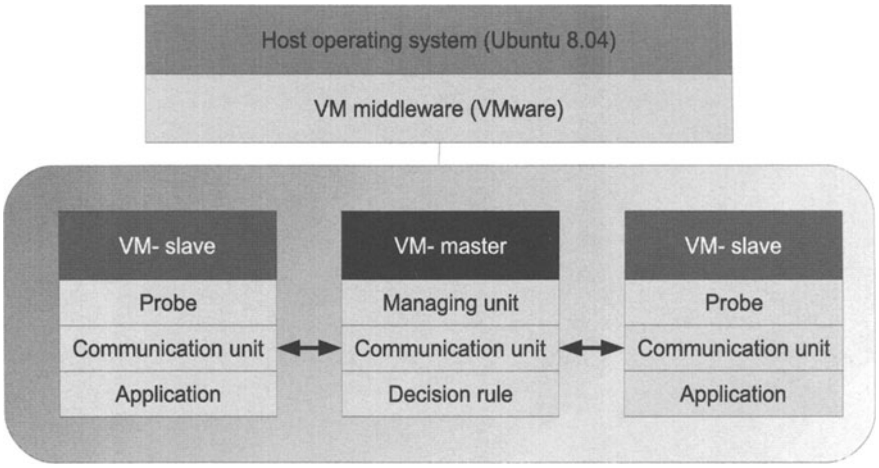

Fig. 1. Proactive VM-rejuvenation framework

- There is at least one active VM-slave (if possible.)
- All VM-slaves are functioning according to same rules.
- If the VM-master decides that the active VM-slave will crash soon it sends a control message to a ready VM-slave to become active. When the new VM-slave becomes active the old one is forced to rejuvenate.

3.2 VM-Master Components

Decision rule
The decision rule is a function from the history of parameters of a VM-slave to a binary value `YES/NO`. It is obtained off-line by the developed machine learning technique and is hard-coded in the VM-master. If the value is `YES` then the corresponding VM-slave needs to be rejuvenated.

Managing unit
The managing unit holds information about which VM-slaves are currently connected and what is their most recent status. When the Decision Rule decides that a VM-slave needs rejuvenation and informs the Managing unit, it starts rejuvenation at a suitable moment.

Communication unit
The communication unit is responsible for receiving VM-slave parameters and responding with simple commands for activating the application in a VM-slave. The communication can be performed using either TCP-IP or VMCI protocols (provided by VMware.)

3.3 VM-Slave Components

Probe
The probe collects system parameters of the VM-slave such as but not limited to a memory distribution and a CPU load.

Communication unit
The communication unit receives orders about the execution of the application from the VM-master and follows them. This way it serves as a managing unit as well. Another duty of the communication unit is to report the system parameters that has been collected by the probe.

Application
The application can be virtually any legacy code. It can be an Apache web server, a protein folding simulation or any other program.

4 Machine Learning Framework

The VM-REJUV framework presented in [1] relies simply on selecting a level of the current CPU utilization of a VM-slave to decide whether it needs to be rejuvenated. This has been shown to be effective for detecting memory leaks but has some limitations and drawbacks ([10]).

First, it discards a lot of the parameters of the VM-slave system, which may be used for further refining the decision rule. Therefore, there is no warranty that any empirically chosen level will be good for all scenarios. Some attacks and exploits may be keeping the CPU utilization high enough to prevent the rejuvenation of the VM-machine.

Second, it doesn't keep any track of previous times. Some attacks are recognizable only if one considers several consecutive moments in time combined.

The proposed solution in this paper is eliminating these drawbacks. The machine learning technique for deriving an adequate decision rule that has been developed in this paper is extending the capabilities of the Proactive VM-rejuvenation framework to predict anomalies leading to the system crash. It is presented in Figure 2 and consists of five steps.

1. Training Data Collection
   To be able to detect anomalies (memory leaks) in advance, the system needs to have information about the symptoms of such anomalies. Such data can be obtained by exposing the system to the anomalies several times and recording the system parameters through the time.
2. Data Labeling
   The system parameters record needs to be tagged with the remaining time to the crash. This means for an every moment in time, in which the system parameters are recorded, an additional parameter is added i. e., the time remaining to crash. Note that this value cannot be known in advance. The goal of this framework is to be able to extract a good prediction for the time to crash from the rest of the parameters. Such prediction can be used in the decision rule.
3. Data Aggregation
   The system parameters for a certain period of time are collected and combined in what is called an aggregated datapoint. To such datapoint are added additional parameters, which describe the dynamics of the parameters during the time period. For example, the average slope of each parameter is

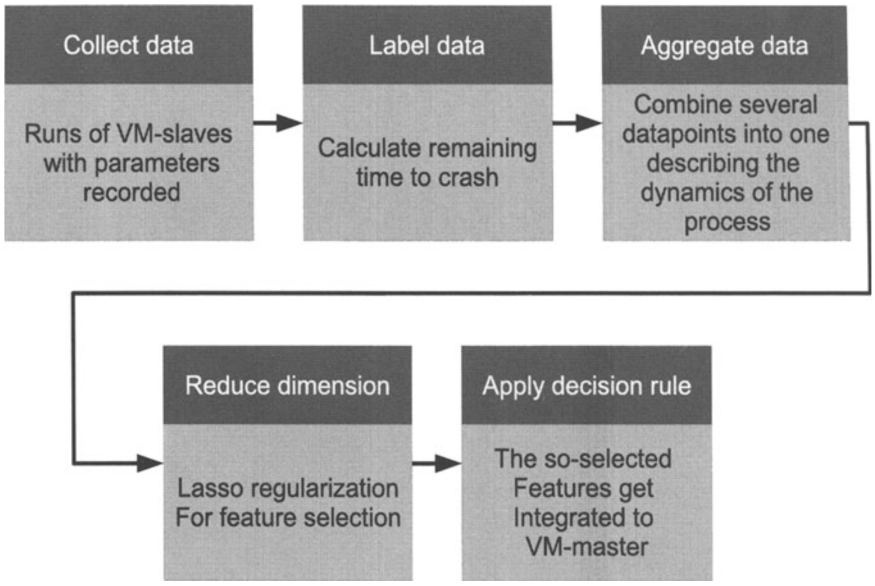
Collect data
Runs of VM-slaves with parameters recorded
Label data
Calculate remaining time to crash
Aggregate data
Combine several datapoints into one describing the dynamics of the process
Reduce dimension
Lasso regularization For feature selection
Apply decision rule
The so-selected Features get Integrated to VM-master

Fig. 2. ML framework

an aggregated datapoint. This aggregation increases the number of parameters to consider many-fold, and each parameter constitutes an additional dimension in the representation of the problem. Considering all of them is not the most efficient approach as some of them may be irrelevant to a certain anomaly. Also to provide convergence guarantees for a decision rule in a certain dimension, the higher dimension, the higher number of training points is required. By reducing the dimension of aggregated datapoints the convergence becomes possible and tractable.

4. Feature selection
A sparse regression, also known as Lasso regularization([9]), is performed to reduce the number of important parameters to a certain number, which can be controlled. Lasso regularization is explained further in the paper.
5. Decision rule application
The solution of a Lasso regularization is a parse set of weights of the parameters in the aggregated datapoint. Application of the decision rule can be implemented by calculating aggregated datapoint on the fly and taking the dot product of it and the weights obtained by Lasso regularization.
More sophisticated machine learning methods with higher degree kernels can be applied to the reduced dimensionality datapoints. These could be Support Vector Machines (SVM) and Regularized Least Squares (RLS) ([7]). This step might not be necessary in some cases but in other it might further boost the efficiency of the decision rule. Because Lasso regularization only tries to find a linear regression, this step might be necessary for some problems and anomalies that might have a non-linear behavior.

5 Lasso Regularization

A machine learning task is equivalent to learning a function or a close approximation to it, given the values of the function at some points ([3],[4]). These values will be called training data. There could be many functions, which satisfy the training data or have a small difference. A measure of how well a function matches the training data is the Empirical Risk([2]). Therefore, a function that minimizes the Empirical Risk might look like a good candidate function. However, such functions have the drawback that they overfit the training data i. e., these functions adjust themselves to the training data for the cost of making themselves more complicated, which leads to them having uncontrollable and hard to predict behavior if evaluated at other points. Therefore, a machine learning tries to regularize such functions by assigning some penalty to their complexity i. e., the more complicated the function, the higher is the penalty.

The most common and widely known regularization technique is Tikhonov regularization([8]). It selects the function to be learned by the following rule:

$$f = \arg\min_{f \in H} \frac{1}{m} \sum_{k=1}^{m} V(f(X_k), Y_k) + \lambda ||f||_H \tag{1}$$

In this formula H is the space of all functions that are considered (usually some Hilbert space with a defined norm, usually L2 norm), m is the size of the training data, (X_k, Y_k) is the format of the training data - X_k is a vector of parameters and Y_k is a scalar or a vector of values that somehow depend on the parameters (in this paper Y_k is the remaining time to crash), V is a loss function that penalizes empirical errors. λ is a parameter, which controls how much to regularize and how important is minimizing the empirical risk. Usually, the best value for λ is selected through a cross-validation.

Lasso Regularization differs slightly from Tikhonov regularization and the difference is that the norm on the function is not given by the Hilbert Space the function is in, but is the L1 norm. The function selection rule takes the form:

$$f(x) = <\beta, x> \tag{2}$$

where x can be any vector variable of parameters. The vector β is derived by:

$$\beta = \arg \min_{\beta \in \mathbb{R}^{dim(X_k)}} \frac{1}{m} \sum_{k=1}^{m} (<\beta, X_k> - Y_k)^2 + \lambda ||\beta||_{L1} \tag{3}$$

The functions that Lasso regularization considers are restricted to linear functions but it has the property that the selected weight vector β is sparse, i.e. the majority of its coordinates are zeros. An intuition about this can be observed in Figure 3:

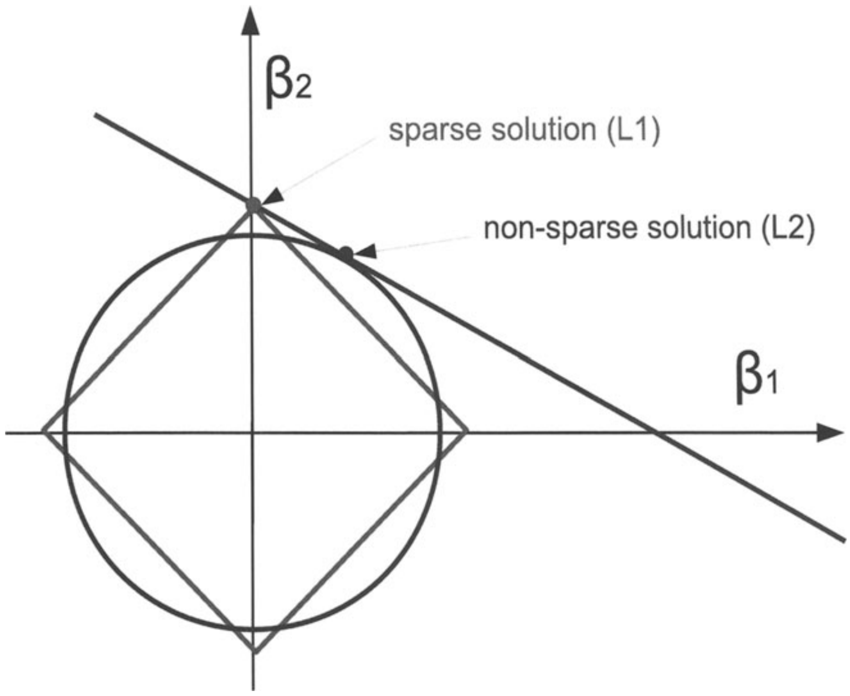

Fig. 3. Sparsity of Lasso regularization

At Figure 3 β_1 and β_2 represent the different coordinates of β. The sloped violet line represents space of solutions with equal empirical risk. Then, among them, one needs to chose the solution that minimizes the regularization penalty. For Tikhonov regularization (red) the regularization penalty is the L2 distance between a solution and the origin of the coordinate system. Therefore, the best solution is at a tangent point between a circle centered at the origin and the sloped line. For Lasso regularization the penalty is the L1 distance between a solution and the origin. Therefore, the best solution is at a tangent point of L1-ball (green rhomboid) and the line, which will happen to be at a some subset of the axes i. e., therefore, it will be sparse. Similar arguments in higher dimensions justify the sparsity of Lasso regularization in general.

6 Experimental Setup

Two laptops Dell M1530 with 4GB RAM and 2GHz Core Duo processor have been used for performing the experiments and the Proactive VM-Rejuvenation Framework has been installed on each of them. The operating system was Linux (Ubuntu 8.04). The virtual machines were created and maintained with VMWare Workstation 6.5, but this is not necessary - they could be managed with any other virtualization software. There was one VM-master and two VM-slaves. They were communicating to each other via VMCI protocol, but of course other forms of communication such as TCP-IP are possible. All the software in VM-master and VM-slaves was self-written or built-in in Ubuntu.

In order to demonstrate the scalability of the proposed Proactive VM-rejuvenation framework, as shown in Fig 1, it has been has implemented with a possibility to introduce multiple VM machines, independent of the available hardware. This approach also demonstrates the minimal hardware requirements. Still, the Proactive VM-rejuvenation framework can scale horizontally, to many physical machines. VM-masters and VM-slaves could be replicated multiple times, if the VM-masters can synchronize their actions, for example with a common database.

The Managing unit in the VM-master, the Communication units in the VM-master and VM-slaves, the Probe and a sample Application were self-written and are in the range of few thousands lines of C code. For the decision rule were used some freely available libraries of implementations of Lasso regularization.

The Probe collects parameters about a VM-slave, combines them in a strictly defined form and sends the data to the VM-master on a regular interval. In the experiment performed this interval was set to one second. The form of the data is the following:

```
Datapoint:
Memory: 515580 497916 17664 0 17056 268692
Swap: 409616 0 409616
CPU: 52.380001 0.070000 3.090000 0.260000 0.000000 44.200001
```

Such datapoint contains information about the memory distribution and CPU activity.

The Application in the VM-slaves had the capability to produce memory leaks. Its only task was to accumulate them.

The Communication units were responsible for transmitting data between the Probe and the Communication unit in the VM-slave and for transmitting the commands from the Communication unit of the VM-master to the Communication unit of the VM-slave.

Besides communication with the VM-master, the Communication unit of the VM-slave is responsible for only executing simple commands like `START` and `STOP` the application.

The data collection per each laptop has been conducted for 63 runs, each of them consisted of approximately 15-30 minutes of parameter history recorded every second. That data was aggregated and labeled with a simple self-written Python script. The Lasso regularization was performed using freely available implementations of Lasso regularization.

Rejuvenation

For rejuvenation was used a restart of the virtual machine. Another approach would be to simply restart the process of the application. However, this would not completely restore the original state of the system when the application was started. For example if the application has used the swap space this would not be cleared after a process restart but would be after a virtual machine restart. The only way that it can be guaranteed that the system parameters will be the same at the start of the application is through a virtual machine restart.

7 Results

Figure 4 shows some of the values of the parameters combined in the aggregation step, change with a respect to the time before crash for one particular run. These parameters describe the memory distribution, the swap memory distribution (on the left) and the CPU load distribution (on the right). These are presented for one of 30 instances used for the aggregation, correspondingly at time 15 seconds. The values of the parameters are in parameter units. For example, for memory parameters the units are KB and for CPU parameters the units are % (percent).

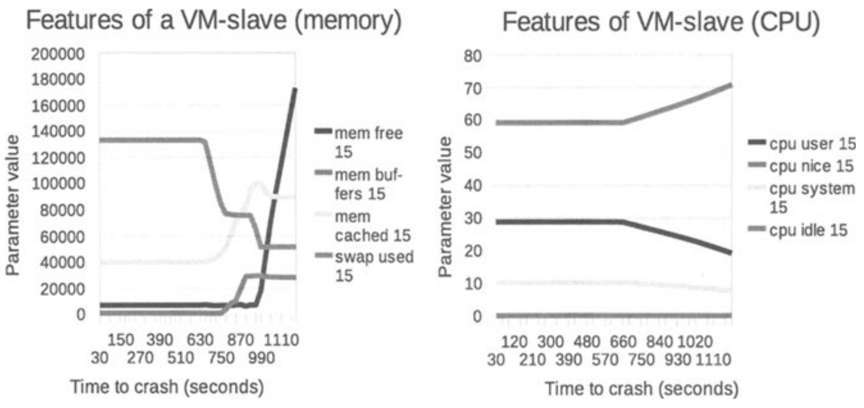

Fig. 4. Variation of all parameters over time

Figure 5 shows some additional parameters(the average slopes) that were calculated for aggregation. For Figure 5 the values of the parameters are shown in parameter units per time. For example, for memory parameters the units are KB/s and for CPU parameters the units are %/s (percent per second).

However, some problems with the probes have been observed in the cases when a certain level of memory leaks have been reached. Unfortunately, this holds for all runs and consists of repeating the old system parameters without a change. It can be observed at figures 4 and 5 as flattening of all parameter plots approximately 600 seconds before the actual crash. However, this outrage of the Probe module does not change the effectiveness of the machine learning method. This is explained later in the paper at Figure 8.

Another specific of Lasso regularization is that the algorithm is not guaranteed to converge to the global minimum for β, but may end up with a local minimum solution. This is due to the fact that Lasso regularization is a convex relaxation

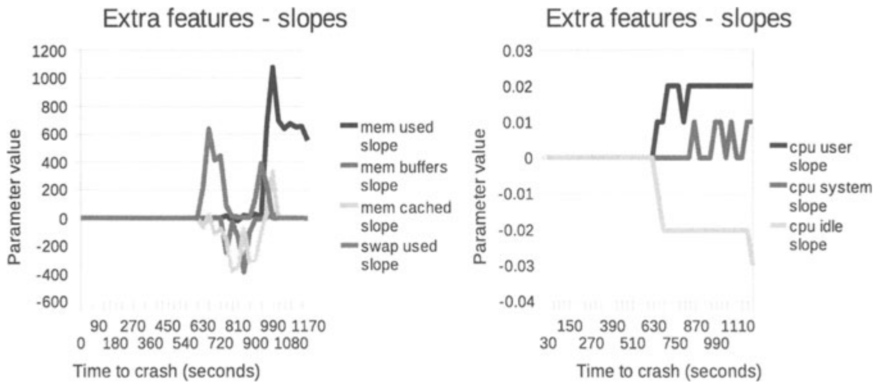

Fig. 5. Variation of extra calculated parameters (slopes) over time

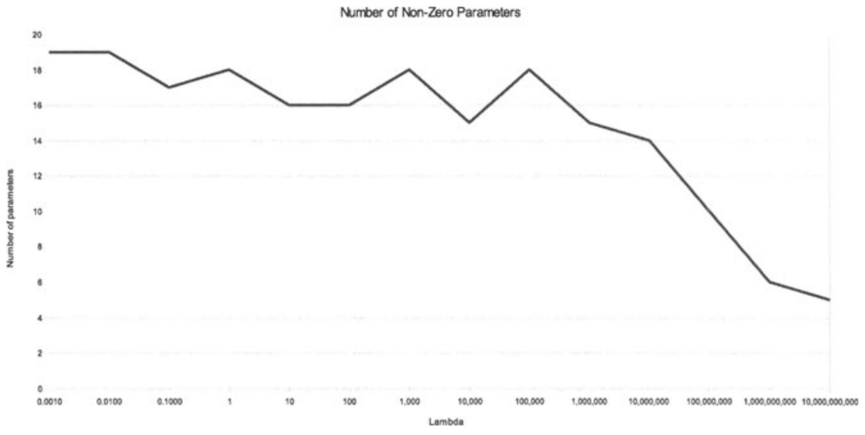

Fig. 6. Variation of the number of non-zero parameters with respect to λ

of a NP-hard problem. Yet, the solution that the algorithm provides is good enough in the sense that it exhibits important properties such as a sparsity and a good regression solution. This is illustrated in Figure 6, by showing the number of the parameters in the sparse solution with respect to lambda. The general trend is to decrease the number of parameters, even though this doesn't happen strictly monotonously. After aggregating the datapoints Lasso regularization was performed on them, and the weights selected for the parameters for few values of λ are presented in Figure 7. Many of the parameter weights are zeros, which is expected since the method provides a sparse solution. The sparsity of the solution can be adjusted by the value of λ.

For example, in the case $\lambda = 10$, only 5 out of 39 parameters were given high non-zero weights. All other parameters had weights smaller than `0.01`. These five parameters are shown in Table 1:

Table 1. Most important parameters after feature selection for $\lambda = 10$

Parameter name	Weight
`mem_used_slope`	-0.70
`swap_used_slope`	0.89
`cpu_user_15sec`	12.01
`cpu_idle_15sec`	17.52
`cpu_user_30sec`	9.12

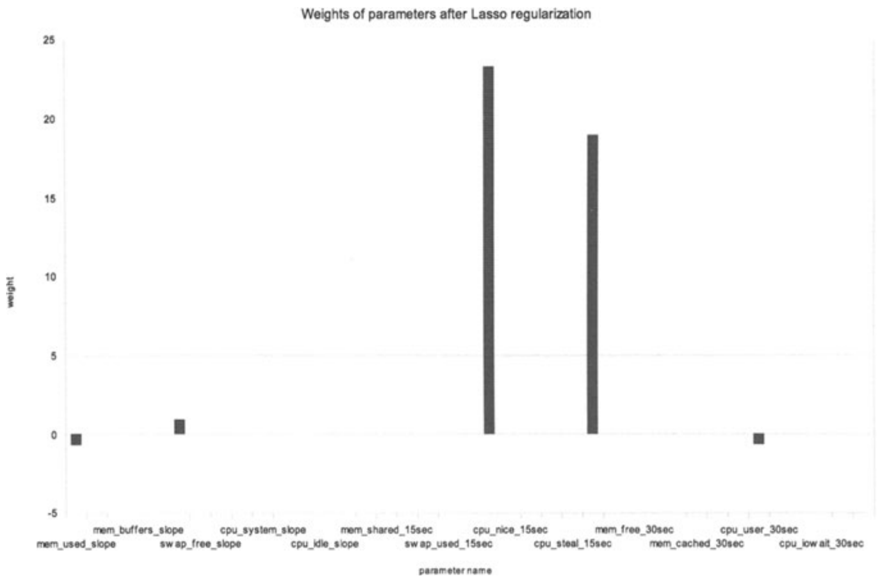

Fig. 7. Selected weights for the parameters after Lasso regularization for several values of λ

Decision Rule
When the weights of the parameters were multiplied to the values of the parameters at each datapoint and summed the result is a close predictor of the actual remaining time to crash. For that datapoint, the calculated remaining time to the crash is incorporated in a decision rule. Figure 8 is an example of the correspondence between a predicted and actual remaining time on one of the runs. The training was done over all runs, and the figure presents only one of the runs. The ground truth is the dashed line called "Actual time", and the predicted remaining time for various values of λ is described by the other lines in Figure 8. The predicted times were calculated by using parameter weights, in the format shown in Figure 7, multiplied to the parameter values in the format shown in Figures 4 and 5 to obtain time to crash value and then summed up. This is equivalent to taking the dot product between the weights vector $\boldsymbol{w}$ and the parameters vector $\boldsymbol{p}$.

$$\boldsymbol{w}.\boldsymbol{p} = t_{predicted} \tag{4}$$

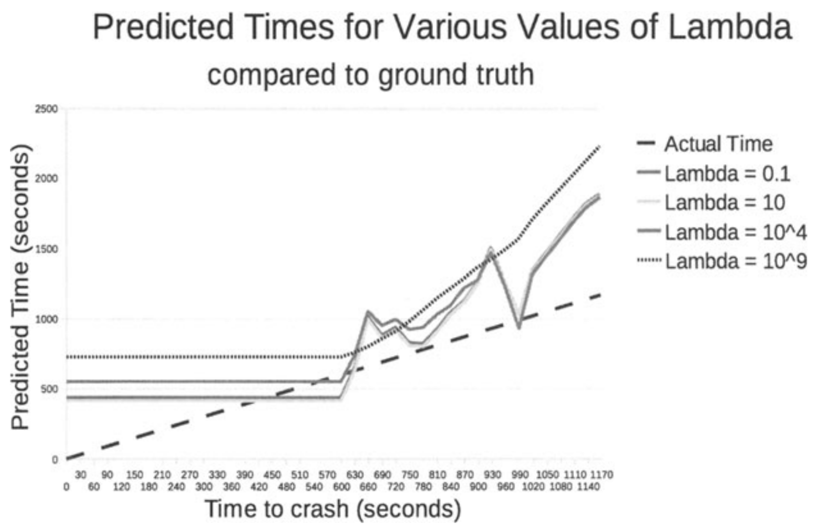

Fig. 8. Comparison between actual remaining time and prediction based on the machine learning algorithm for various values of lambda

The results are presented in Figure 8, which shows that the predicted time for all values of λ is a good approximation of the ground truth. The abscissa shows the remaining time to crash, and the ordinate shows the predicted time in seconds.

Usually, the best value of λ is selected through cross-validation. However, in this case, another property of a good solution is its sparsity. Hence, the value of lambda can be varied to achieve a small number of parameters, which would lead to efficiency from implementation point of view. As can be observed in Figure 8 the quality of the solution doesn't vary greatly as λ varies. The predicted remaining times are for values of λ with multiplicative difference in the order of 10^{13}.

Such predictor was used as a decision rule. If the predicted remaining time is under some safe limit (1000 seconds - more than the minimal predicted time), as in Figure 8, the decision rule is activated and it informs the managing unit of the VM-master that the corresponding VM-slave needs to be rejuvenated. The decision rule was hard coded, since all the learning was done off-line, as it requires the data labeling step of the ML framework, which can be performed only after the data is once collected.

The framework with one VM-master and two VM-slaves, with properly set a decision rule and a bug-free implementation was able to continue changing the load from one VM-slave to another without a server crash. The Proactive VM-rejuvenation framework with a properly devised decision rule flawlessly was able to run for a couple of weeks and switch the activity of VM-slaves every 15-30 minutes. Additional difficulties to that aim were the varying rejuvenation times. Many times all that was needed for the rejuvenation was simply a restart of the virtual machine. However, in some cases was necessary the OS to perform a hard-disk check and this required an additional time to be taken into account during the rejuvenation process.

8 Conclusion

Proactive VM-rejuvenation framework for selecting critical parameters for detecting anomalies in web servers has been presented in the paper. The ability to add arbitrary number of backup virtual machines and reliably to predict the remaining time to crash with the use of machine learning techniques is described. An algorithm for a feature selection, based on machine learning for reducing the complexity and dimensionality of the problem, has been developed. The framework has been implemented with virtual machines and a machine learning algorithm has been realized for successfully determining a decision rule for proactively initiating the system rejuvenation. The proposed framework has been theoretically justified and experimentally validated. These are real problems for the Internet today and the future cyber infrastructure. The proposed machine learning method is general and can be applied for a wide range of anomalies.

9 Future Work

One opportunity for extension is to apply other machine learning techniques on the top of Lasso Regularization. Such techniques could be Regularized Least Squares (RLS) or Support Vector Machines (SVM). They could be used with a non-linear kernel and learn more complicated behavior. This would reduce the number of false positives and false negatives of the decision rule significantly.

Another opportunity for example is to learn to defend against more than one type of anomaly. If decision rules against memory leaks and denial of service attack can be learned, both of them can be used simultaneously. In this case, whenever any anomaly occurs, the rejuvenation of the VM-slave will be initiated.

Each virtual machine can implement a simplified version of the proposed framework that includes the embedded decision rule and the probe for monitoring the parameters in a real time. These virtual machines can be provided to the clients on demand across the network.

References

1. Silva, L., Alonso, J., Silva, P., Torres, J., Andrzejak, A.: Using Virtualization to Improve Software Rejuvenation. In: IEEE Network Computing and Applications, Cambridge, USA (July 2007)
2. Bousquet, O., Boucheron, S., Lugosi, G.: Introduction to Statistical Learning Theory. In: Bousquet, O., von Luxburg, U., Rätsch, G. (eds.) Machine Learning 2003. LNCS (LNAI), vol. 3176, pp. 169–207. Springer, Heidelberg (2004)
3. Poggio, T., Smale, S.: The Mathematics of Learning: Dealing with Data. Notices of the AMS (2003)
4. Bishop, C.: Pattern Recognition and Machine Learning. Springer, Heidelberg (2007)

5. Chen, M., Accardi, A., Kiciman, E., Lloyd, J., Patterson, D., Fox, A., Brewer, E.: Path-based failure and evolution management. In: Proc. of the 1st Symposium NSDI 2004 (2004)
6. Cherkasova, L., Fu, Y., Tang, W., Vahdat, A.: Measuring and Characterizing End-to-End Internet Service Performance. Journal ACM/IEEE Transactions on Internet Technology, TOIT (November 2003)
7. Evgeniou, T., Pontil, M., Poggio, T.: Regularization Networks and Support Vector Machines. Advances in Computational Mathematics (2000)
8. Cucker, F., Smale, S.: On the mathematical foundations of learning. Bulletin of the American Mathematical Society (2002)
9. Tibshirani, R.: Regression selection and shrinkage via the lasso. J. R. Stat. Soc. Ser. B 58, 267–288 (1996)
10. Cherkasova, L., Ozonat, K., Mi, N., Symons, J., Smirni, E.: Towards Automated Detection of Application Performance Anomaly and Change. HPlabs 79 (2008)
11. Cohen, I., Zhang, S., Goldszmidt, M., Symons, J., Kelly, T., Fox, A.: Capturing, Indexing, Clustering, and Retrieving System History. In: Proc. of the 20th ACM Symposium SOSP 2005 (2005)
12. Mi, N., Cherkasova, L., Ozonat, K., Symons, J., Smirni, E.: Analysis of Application Performance and Its Change via Representative Application Signatures. In: NOMS 2008 (2008)
13. Zhang, Q., Cherkasova, L., Mathews, G., Greene, W., Smirni, E.: R-Capriccio: A Capacity Planning and Anomaly Detection Tool for Enterprise Services with LiveWorkloads. In: Cerqueira, R., Campbell, R.H. (eds.) Middleware 2007. LNCS, vol. 4834, pp. 244–265. Springer, Heidelberg (2007)
14. Huang, Y., Kintala, C., Kolettis, N., Fulton, N.: Software Rejuvenation: Analysis, Module and Applications. In: Proceedings of Fault-Tolerant Computing Symposium, FTCS-25 (June 1995)
15. Castelli, V., Harper, R., Heidelberg, P., Hunter, S., Trivedi, K., Vaidyanathan, K., Zeggert, W.: Proactive Management of Software Aging. IBM Journal Research & Development 45(2) (March 2001)
16. Rosenblum, M., Garfinkel, T.: Virtual Machine Monitors: Current Technology and Future Trends. IEEE Internet Computing 38(5) (May 2005)
17. Vaidyanathan, K., Trivedi, K.: A Comprehensive Model for Software Rejuvenation. IEEE Trans. on Dependable and Secure Computing 2(2) (April 2005)
18. Vaidyanathan, K., Trivedi, K.S.: A Measurement- Based Model for Estimation of Resource Exhaustion in Operational Software Systems. In: Proc. 10th IEEE Int. Symp. Software Reliability Eng., pp. 84–93 (1999)
19. Li, L., Vaidyanathan, K., Trivedi, K.: An Approach for Estimation of Software Aging in a Web-Server. In: Proc. of the 2002 International Symposium on Empirical Software Engineering, ISESE 2002 (2002)
20. Gross, K., Bhardwaj, V., Bickford, R.: Proactive Detection of Software Aging Mechanisms in Performance Critical Computers. In: Proc. 27th Annual IEEE/NASA Software Engineering Symposium (2002)
21. Kaidyanathan, K., Gross, K.: Proactive Detection of Software Anomalies through MSET. In: Workshop on Predictive Software Models (PSM 2004) (September 2004)
22. Gross, K., Lu, W.: Early Detection of Signal and Process Anomalies in Enterprise Computing Systems. In: Proc. 2002 IEEE Int. Conf. on Machine Learning and Applications, ICMLA (June 2002)

23. Silva, L., Madeira, H., Silva, J.G.: Software Aging and Rejuvenation in a SOAP-based Server. In: IEEE-NCA: Network Computing and Applications, Cambridge USA (July 2006)
24. Candea, G., Brown, A., Fox, A., Patterson, D.: Recovery Oriented Computing: Building Multi-Tier Dependability. IEEE Computer 37(11) (November 2004)
25. Oppenheimer, D., Brown, A., Beck, J., Hettena, D., Kuroda, J., Treuhaft, N., Patterson, D.A., Yellick, K.: ROC-1: Harware Support for Recovery-Oriented Computing. IEEE Transactions on Computers 51(2) (2002); Special issue on Fault Tolerant - Embedded Systems, Avresky, D., Johnson, B.W., Lombardi, F. (Guest eds.)