Pengfei Li (pl189)
Homework 7, ECE 590 & CS320 Software Reliability.

This paper first introduces some concepts about software aging and the reasons why the software aging will happen in the software. and the concept of software rejuvenation and how to trigger the rejuvenation based on the system state. After giving us some background about the concept, it focuses on the comparative experimental study on the software rejuvenation techniques based on six different rejuvenation techniques. These techniques cover all of studies so far with different levels of granularity. They are (i) physical node reboot, (ii) virtual machine reboot, (iii) OS reboot, (iv) fast OS reboot, (v) standalone application restart, and (vi) application rejuvenation by a hot standby server. And in the experimental, the author analyzes the performance overhead and fragmentation overhead of these methods. This paper also gives us the conclusion about the effectivity of these six methods bring to us. And it gives some shortage about using virtualization by researching the memory fragmentation overhead. Finally, this paper gives us some suggestions and guidelines when we design the rejuvenation scheduling algorithms and select the appropriate rejuvenation mechanism.
Detailed information on software aging and software rejuvenation not mentioned in the paper is given here.
The time-based software rejuvenation approach first assumes that there is already a software aging problem in the system, and under this assumption, the operating state of the system, the distribution of software failures, and the distribution of repair times are assumed, and then the selected model is used to maximize the time available to the system or minimize the software rejuvenation cost, and the software rejuvenation operation is performed at a determined time interval when the software rejuvenation is executed. It can be classified as follows.
1) Markov model and its corresponding extended model
Huang et al modeled the software aging process using a 4-state (normal state, aging state, software rejuvenation state, and failure state) continuous-time Markov model that maximizes the system available time.
2) Petri Nets
Petri net model is a mathematical representation of discrete parallel systems, suitable for describing asynchronous and concurrent computer system models. Petri net model can describe the parallelism, distribution, uncertainty, resource competition, etc. of the system graphically, and in the early anti-fading, this model or stochastic Petri net is usually used (based on Petri net, time parameters and stochastic concept are introduced. assuming that the residence time of the system in a state is a continuous random variable) to describe the software rejuvenation strategy.
3) Other modeling approaches
Eto et al used reinforcement learning methods (which do not require complete knowledge of system failure or performance degradation) to calculate the optimal software rejuvenation timing.
Time-based approaches often make simple assumptions about the system state, load, etc. However, such simple assumptions are often difficult to satisfy in a real application system, and such models often use empirical analysis or use simulated data to verify the validity of the proposed model when validating the proposed model, rather than using

data collected in a real system. The execution of anti-debris at specified time intervals is not suitable for real systems, especially for real-time systems, as the interval is too small to execute anti-debris early and too large to execute anti-debris in time when failure occurs.

The conclusion obtained in this paper is that restarting the application process with a hot standby application server is an optimal choice from the customer's point of view in terms of evaluating the number of failed requests. And the memory fragmentation caused by virtualization technology on the server side introduces a lot of overhead to the rejuvenation technology, which has a negative impact. If we must use virtualization technologies, we must have ways to mitigate this negative impact. On top of that, the article demonstrates that VM reboots cause more memory fragmentation than just rebooting the OS within the VM. And if we use fast OS reboots inside VMs, then if virtualization is required, performing a cold reboot of a physical machine is better than a VM reboot as it will generate fewer failed requests in terms of the number of failed requests on the client side and on the server side.

Specifically for the experimental results at different granularities, at the application granularity level, we observe that the scenario using virtualization generates high overhead in terms of memory fragmentation for APPStandVM and APPHotVM compared to APPStandPH. And these fragmentation events are mainly caused by the system processes responsible for the virtualization and load balancing infrastructure. The fragmentation overhead caused by these processes is significant.

In terms of results at the OS granularity level, in terms of memory fragmentation, we observe that OSRebootPH and OSFastPH show fewer memory fragmentation events compared to OSRebootVM. This result proves that the increased virtualization infrastructure introduces a significant amount of overhead that leads to high levels of memory fragmentation. Ultimately, it proves that OS reboots have an important role in eliminating this type of aging effect.

The virtualized solution had more failed requests than the non-virtualized solution in terms of virtual machine granularity level and physical node level. And the number of slow requests using VM reboots is smaller than the number of slow requests using physical node reboots. This result follows the same pattern as in the OS reboot analysis. The longer downtime increases the number of failed requests and decreases the number of slow requests. Similar results to the previous two granularity experiments were obtained for memory fragmentation.

For the further research, although this paper presents an analysis of the performance of software rejuvenation at different granularities, the software rejuvenation data collected in these or in previous studies are often based on monitoring data in a controlled environment, or even some simulated data, rather than data in a real operating environment, so the use of such data does not reflect the real operation of the system and can cause the problem of too frequent or insufficient software rejuvenation. If the aging is too frequent, the system will frequently perform software rejuvenation operations, which will reduce the user experience of using the system. If there is not enough software rejuvenation, the system state is not guaranteed to be in normal condition, and even service failure may occur, which will also cause the user experience to be degraded. The rejuvenation approach to be used in cloud-based systems may introduce some new problems that have not been encountered. It is also possible that some differences in

user experience may be introduced in the real environment. Therefore, I believe further research is needed for this problem.