Detecting Software Aging in a Cloud Computing Framework by Comparing Development Versions

Felix Langner
Heidelberg University, Germany
Email: felix.langner@uni-heidelberg.de

Artur Andrzejak
Heidelberg University, Germany
Email: artur@uni-hd.de

***Abstract*—Software aging, i.e. degradation of software performance or functionality caused by resource depletion is usually discovered only in the production scenario. This incurs large costs and delays of defect removal and requires provisional solutions such as rejuvenation (controlled restarts). We propose a method for detecting aging problems shortly after their introduction by runtime comparisons of different development versions of the same software. Possible aging issues are discovered by analyzing the differences in runtime traces of selected metrics. The required comparisons are workload-independent which minimizes the additional effort of dedicated stress tests. Consequently, the method requires only minimal changes to the traditional development and testing process. This paves the way to detecting such problems before public releases, greatly reducing the cost of defect fixing. Our study focuses on the memory leaks of Eucalyptus, a popular open source framework for managing cloud computing environments.**

***Index Terms*—Software Aging, Software Development, Cloud Computing**

I. Introduction

Software aging is defined as progressive performance or functionality degradation in software systems [1]. The typical causes are exhaustion of resources due to memory-leaks, unreleased locks, non-terminated threads, or storage fragmentation. Figure 1 shows the cause-effect chain culminating in a system failure due to software aging. Faulty code repeatedly activates software malfunction which (after a certain progress of execution) leads to performance degradation and finally to a failure.

Contrary to the "traditional" software defects which trigger conspicuous and direct failures, aging manifests itself with large latency, usually from hours to weeks. This property complicates severely detection of aging problems during software development and testing, leading to the situation that most of them are discovered during productive deployment. Consequently, the primary technique to combat software aging are controlled restarts known as *software rejuvenation* [2]. The majority of research in the last two decades was targeting modeling of the degradation process and optimizing rejuvenation schedules [3], [4], [5], [6] (see also Sec. IV).

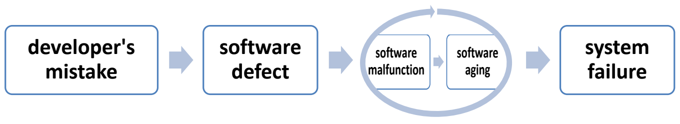

Figure 1. A cause-effect chain culminating in a system failure

Aging issues are likely to be found in any type of software with enough complexity, but it is particularly troublesome in long-running applications. Examples include telecommunication systems, web-servers, web-service middleware [6], or cloud computing infrastructure [7].

In this work we investigate how aging issues can be detected already *during* the software development and testing phase. The key idea is to compare the behavior of suitable runtime metrics across several versions created in the development process. Anomalous behavior of some metrics in the latest version is a likely indication of a new problem. Its detection should trigger more detailed investigation and tests. An essential advantage of this approach is that it can be integrated in the traditional software development process with a relatively small additional effort. This is mostly due to the property that the method is workload-independent and can likely reuse existing unit or performance tests.

Contributions of this work are the following ones:

- We propose an approach for discovering software aging problems via comparisons of runtime metrics between related development builds.
- We suggests a hierarchical method for identifying relevant runtime metrics.
- We evaluate our technique on Eucalyptus, a popular framework for managing cloud computing infrastructures. We show that (i) our approach is able to detect a real (i.e. non fault-injected) aging problem, (ii) it is indeed workload-independent, and (iii) the relevant metrics are correctly identified.

II. Differential Detection Method

This section outlines the proposed approach. The key element is to exploit information obtained via comparisons of runtime behavior of software versions to detect possible aging issues.

A. Assumptions and Approach

We assume a typical scenario where software is developed in a series of minor versions (revisions), each producing an executable build. For each of these, integration and possibly performance tests are performed. Furthermore, it is possible to instrument the project code and/or testing harness in order to collect runtime metrics during the tests.

A software aging defect might be introduced in such a development process. This is likely to change the runtime behavior of the latest built. In order to notice this anomaly, we collect at each build's unit or performance test traces of selected runtime metrics $m_1, \ldots, m_k$. Such metrics can be CPU usage, heap usage in a JVM, and others (see Sec. III-B).

We speculate that if the latest build B has a newly introduced aging problem, the characteristics of at least one of the collected metrics, say m_i, will change. Such a change can be noticed by comparing traces $m_{i,B}$ of metric m_i on B against traces $m_{i,G1}$ of m_i on some previous build, say $G1$. A *difference function* $D = D(m_{i,B}, m_{i,G1})$ shall quantify this change. It is specific to the metric and the system; for example, D can be correlation between traces or the relative difference of the test-final metric values (see Sec. III-C).

To understand which value of D is indeed anomalous we need the traces from yet another build, say $G2$. Both $G1$ and $G2$ are assumed to be free of aging defects ("good"), or at least to have very similar behavior. For each metric m_i we compare the difference $D(m_{i,B}, m_{i,G1})$ (or $D(m_{i,B}, m_{i,G2})$) against $D(m_{i,G1}, m_{i,G2})$. Only if $D(m_{i,B}, m_{i,G1})$ is significantly larger, we consider this as an indication of a potential aging problem in B. This case (for any of the metrics) shall trigger an "alert" leading to further in-depth tests. Note that the (usually small) source code differences between B and $G1$ or $G2$ can give further hints about the location of the defects (see future work).

B. Metric Selection

The above process can be optimized by pre-selecting metrics which are most likely to give meaningful results. For example, some metrics such as CPU usage can be strongly influenced by stochastic "noise" and behavior of other processes while other metrics (e.g. number of open files) might not be influenced by aging issues at all.

We can filter out metrics with high level of noise via the following hierarchical approach. The first, automated step is to compute the pairwise correlations between $m_{i,G1}$ and $m_{i,G2}$ (for two builds $G1$, $G2$ assumed to be aging-free and each $i = 1, \ldots, k$) and remove all metrics whose correlation is below a certain threshold. The remaining metrics are then compared and selected visually as illustrated in Sec. III-B.

It is more involved to eliminate metrics which do not respond to the aging issues. The most reliable approach has turned out to contrast them visually on three builds: both "good" ones $G1$, $G2$ and a build B' known to contain an aging problem. If a considered metric shows similar values for $D(m_{i,B'}, m_{i,G1})$ as for $D(m_{i,G1}, m_{i,G2})$, it can be eliminated. This approach requires a prior knowledge of an aging issue, but the metric selection is performed only once. Furthermore, B' can be created by fault injection. Another option is to keep all metrics (without "noise") as already only one correctly working metric will indicate a problem.

C. Workload-Independence

The selection approach retains only metrics without stochastic "noise". This implies that such metrics show similar response to the same workload patterns across various runs and various "good" builds $G1$, $G2$. Consequently, the difference function D is likely to yield similar values independently of the (non-trivial) workload pattern. This conjecture is indeed confirmed for the Eucalyptus scenario (Sec. III-D).

This *workload-independence* greatly facilitates low-effort integration of the proposed method into traditional software testing processes. Essentially, existing integration/unit tests shall be sufficient to collect meaningful traces, without need for dedicated stress tests. The most time-consuming part of implementing the method are the instrumentation of the built and test harness for metric collection and the creation of system-specific evaluation scripts.

III. Experimental Evaluation

The evaluation of our method has been conducted on an older version the Eucalyptus framework which is known to have software aging issues [7]. The specific problem was the memory depletion of the node controller (NC), one of Eucalyptus' services.

A. System under Study

The experiments were executed on a single virtual machine running Ubuntu 10.10 with kernel 2.6.35.10 using 2 GB RAM main memory and two cores of an Intel i7 CPU.

Three different versions of Eucalyptus were prepared for the experiments. They are based on the changesets (retrieved by us from the project repository) prior to and of the release version 1.6.1. This release corresponds to the revision r946 in which the problem reported in [7] has been fixed. Specifically, we prepared the following versions:

- B: This is the youngest revision (r944) before the fix (i.e. a version "with the bug").
- $G1$: This is the first bug-free version (r946) with fixes introduced in revisions r945 and r946.
- $G2$: This is an older revision (r940) on which we applied the fixes of r945 and r946. This synthetically created version serves as a second version without bug.

We used three different workloads during our experiments. Each lasted about 30 minutes and consisted of basic Eucalyptus commands (*start*, *reboot* and *kill* of a VM):

- $W1$: Accelerated version of the workload from [7] that waits 10 seconds instead of 10 minutes between each function call for one virtual machine (5 iterations, i.e. 5 *start-kill* cycles).
- $W2$: A workload that consists of 10 *start-kill* cycles, followed by 1000 consecutively executed *reboot* calls.
- $W3$: A random workload that consists of *start-kill*, *reboot* and *idle* phases, created by a pseudo-random generator. The probability distribution for the occurrence and the duration of each phase was set as the following: $P(\textit{idle}) = 11/16$, $P(\textit{reboot}) = 1/4$, $P(\textit{start-kill}) = 1/16$.

Table I
CORRELATIONS ϱ OF EACH OF THE SIX METRICS FOR ALL VERSION PAIRS (WORKLOAD $W1$)

Metric / Correlation	ϱ(B,G1)	ϱ(B,G2)	ϱ(G1,G2)
CPU usage	0.39	0.27	0.24
Disk space	0.97	0.99	0.96
Number of open files	0.40	0.30	0.34
Number of threads	0.96	0.95	0.95
Allocated main memory	0.41	0.39	0.52
NC RSS	0.99	0.98	0.98

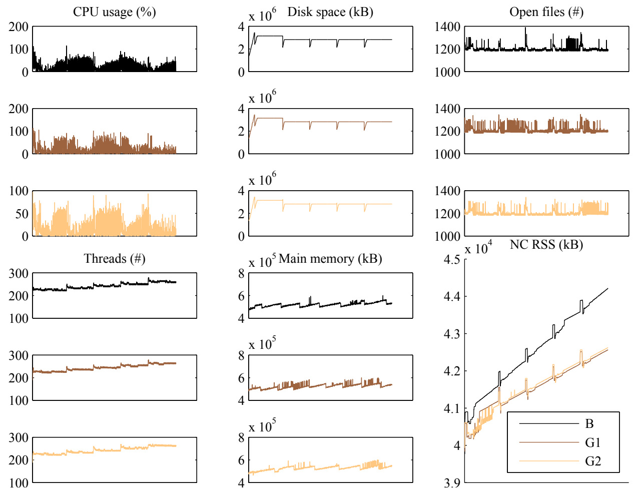

Figure 2. Comparison of the runtime metrics for versions B, $G1$, $G2$

B. Runtime Metrics

We implemented a Python-based framework to collect values of six metrics during each run of an experiment. The metrics represent system resources that typically become affected by resource depletion and are available via basic POSIX system services. The metrics are:

- *CPU usage* (in percent) filtered for all running Eucalyptus processes (acquired via `top`).
- *Disk space* usage for all directories that are known to contain files of Eucalyptus (measured with `du`).
- *Number of open files* for all running Eucalyptus processes (collected via `lsof`).
- *Number of threads* in all Eucalyptus processes (retrieved via `ps`).
- Amount of *allocated main memory* calculated from the resident memory reported by `ps` and memory allocated by the JVM (queried with JMX API).
- *NC RSS,* the resident set (memory) size of the node controller (NC) of Eucalyptus only (measured via `ps`).

Together with the values of the metrics we recorded the corresponding progress of the workload; we synchronized them according to this progress. This was needed as the execution times of Eucalyptus commands vary considerably.

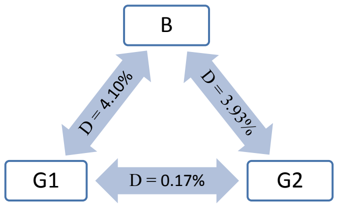

Figure 3. Relative differences D of averaged test-final values of *NC RSS* between Eucalyptus versions B, $G1$ and $G2$ (workload $W1$)

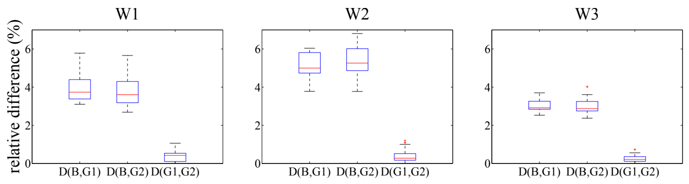

Figure 4. Box plots of the difference value D between versions B, $G1$, $G2$ for workloads $W1$, $W2$, $W3$

Table I and Figure 2 illustrate the hierarchical metric selection described in Sec. II. The correlations in Table I have been computed as Pearson product-moment correlation coefficients ϱ. They show that *CPU usage*, *number open files* and *allocated main memory* have large stochastic "noise". Consequently, they are considered as not significant ("filtered out").

Figure 2 shows traces of all metrics for the software versions $G1$, $G2$ and B (workload W1, after alignment). A visual inspection of the (still significant) metrics indicates that *NC RSS* is the most suitable candidate for detection of aging.

C. Aging Detection

To quantify difference in traces of the metric *NC RSS* we define as the difference function D (Sec. II-A) the relative difference of the last measured *NC RSS* of the traces. That is

$$D(V,V') = D(rss_V, rss_{V'}) = \left| \frac{rss_V - rss_{V'}}{rss_V} \right|,$$

where $rssm_V$ and $rss_{V'}$ are the "test-final" (last) values of the metric *NC RSS* in software version V and V', respectively. To reduce randomness, we have averaged $rssm_V$ and $rss_{V'}$ over five experiment executions.

Figure 3 shows the resulting relative differences for workload $W1$ (averaged over 5 runs). Values of $D(B,G1)$ and $D(B,G2)$ are very similar and about 20 times larger than $D(G1,G2)$. This shows that behavior of *NC RSS* is anomalous in B and indicates a potential aging problem. Also, low value of $D(G1,G2)$ confirms our prior knowledge that both versions behave similarly.

D. Workload-Independence

For each version we run five experiments and computed all 25 combinations for each type of relative difference (i.e. $D(B,G1)$, $D(B,G2)$ and $D(G1,G2)$). This was repeated for each workload $W1$, $W2$, $W3$. Figure 4 shows the resulting box plots. The conclusion for workloads $W2$ and $W3$ is identical as for workload $W1$ (analyzed in Sec. III-C): while $D(G1,G2)$ remain low, relative differences between B

and each of $G1$, $G2$ is much higher. This confirms that in case of Eucalyptus our aging detection method is workload-independent.

IV. Related Work

Most literature on software aging is devoted to modeling performance degradation of software and scheduling rejuvenation actions (i.e. system or service restarts). More recent work focuses on reduction or elimination of availability outages caused by rejuvenation. [8] gives a critical overview of the last 16 years of research in this domain.

Modeling performance can be done via analytic-based and measurement-based approaches. In the former one, various system parameters such as workload and distributions of failure are used to obtain Markov-type models [2], semi-Markov models, and others. The measurement-based approaches use data sampled from the system and creates a performance model via curve-fitting [6], machine learning [9], or time-series analysis [10]. Some works consider combination of both approaches [5].

Eliminating aging via restarts has been first proposed in [2]. Works in this area focus primarily on adaptive rejuvenation. Here time to the complete resource depletion is estimated by above-mentioned models and the schedule of rejuvenation actions is optimized. Consequently, literature is closely related to the modeling [3], [4], [5].

Since restarts cause temporary non-availability, techniques such as recursive (partial) rejuvenation have been developed to shorten the recovery time [11]. Other works use replication of services coupled with virtualization to eliminate the non-availability completely [12], [13].

Detecting and eliminating software aging during development is closely related to debugging and testing. Most mature work here is on tools for identifying memory leaks, for example LeakBot [14] or Memprofiler [15]. Yet aging requires prolonged tests and performance monitoring just to discover problems; only then tools such as LeakBot can be used. A recent work [16] studies how discovery of aging can be accelerated via identifying aging factors and stress testing.

To our knowledge there is no prior work which exploits differences between software versions to detecting aging phenomena. However, this idea has been used to detect traditional software defects, e.g. by analyzing version control commits and corresponding error tickets [17].

V. Conclusion

We have proposed an approach for detecting aging issues by comparing runtime behavior of development versions of software. Experimental evaluation on Eucalyptus framework shows that the method is capable to discover aging issues and it works in a load-independent way. This facilitates implementation of this approach in an existing development and testing process.

Despite of these promising results the approach relies on statistical assumptions and therefore depends on the accuracy and the specificity of the collected measurements. If an aging issue has been introduced prior to any of the "good" versions $G1$, $G2$, such problem will not be discovered. Furthermore, it is not known a priori which metrics can capture aging problems and which are useless. This can cause unnecessary instrumentation and overhead of metrics collection or can lead to omitting relevant metrics. Even false-positives can be induced by differences in the comparison of the runtime behavior of two aging-free versions.

In our future work we will study whether comparing the source code of anomalous software version with previous can help to locate the source of a problem instead of only indicating it. Furthermore, combining our approach with the aging factors and accelerated tests proposed in [16] promises to improve efficiency of discovering aging problems already in the development process.

References

[1] D. L. Parnas, "Software aging," in *Proc. 16th Inter. Conf. on Software Engineering (ICSE '94)*, pp. 279–287, May 1994.

[2] 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.

[3] S. Garg, A. van Moorsel, K. Vaidyanathan, and K. Trivedi, "A methodology for detection and estimation of software aging," in *Proc. 9th Int'l Symposium on Software Reliability Engineering*, pp. 282–292, 1998.

[4] K. Vaidyanathan and K. S. Trivedi, "A measurement-based model for estimation of resource exhaustion in operational software systems," in *Proceedings of 10th IEEE Int'l Symposium on Software Reliability Engineering*, pp. 84–93, November 1999.

[5] K. Vaidyanathan and K. S. Trivedi, "A comprehensive model for software rejuvenation," *IEEE Trans. Dependanble and Secure Computing*, vol. 2, pp. 1–14, April-June 2005.

[6] A. Andrzejak and L. Silva, "Deterministic models of software aging and optimal rejuvenation schedules," in *10th IFIP/IEEE Symposium on Integrated Management (IM 2007)*, (Munich, Germany), May 2007.

[7] J. Araujo, R. Matos, P. Maciel, and R. Matias, "Software aging issues on the eucalyptus cloud computing infrastructure," in *Proc. IEEE Int Systems, Man, and Cybernetics (SMC) Conf*, pp. 1411–1416, 2011.

[8] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, "Software aging and rejuvenation: Where we are and where we are going," in *WoSAR*, pp. 1 –6, Dec 2011.

[9] A. Andrzejak and L. Silva, "Using machine learning for non-intrusive modeling and prediction of software aging," in *IEEE/IFIP NOMS*, (Salvador de Bahia, Brazil), Apr 7–11 2008.

[10] L. Li, K. Vaidyanathan, and K. Trivedi, "An approach for estimation of software aging in a web-server," in *ISESE'02*, pp. 91–102, 2002.

[11] G. Candea and A. Fox, "Recursive restartability: Turning the reboot sledgehammer into a scalpel," in *HotOS*, pp. 125–130, IEEE Computer Society, 2001.

[12] L. M. Silva, J. Alonso, P. Silva, J. Torres, and A. Andrzejak, "Using virtualization to improve software rejuvenation," in *IEEE International Symposium on Network Computing and Applications (IEEE-NCA)*, (Cambridge, MA, USA), July 2007.

[13] A. Andrzejak, M. Moser, and L. Silva, "Managing performance of aging applications via synchronized replica rejuvenation," in *DSOM 2007*, (Silicon Valley, CA, USA), October 2007.

[14] N. Mitchell and G. Sevitsky, "LeakBot: An automated and lightweight tool for diagnosing memory leaks in large Java applications," in *17th ECOOP*, vol. 2743 of *Lecture Notes in Computer Science*, (Darmstadt, Germany), pp. 351–377, Springer-Verlag, June 2003.

[15] Scitech Software, *Memprofiler – .NET Memory Profiler*.

[16] R. Matias, P. A. Barbetta, K. S. Trivedi, and P. J. de Freitas Filho, "Accelerated degradation tests applied to software aging experiments," *IEEE Transactions on Reliability*, vol. 59, no. 1, pp. 102–114, 2010.

[17] S. Kim, E. J. W. Jr., and Y. Zhang, "Classifying software changes: Clean or buggy?," *IEEE Trans. Software Eng*, vol. 34, no. 2, pp. 181–196, 2008.