On the Aging Effects due to Concurrency Bugs: a Case Study on MySQL

Antonio Bovenzi*, Domenico Cotroneo*, Roberto Pietrantuono*, Stefano Russo*†,
*Dipartimento di Informatica e Sistemistica, Universitá di Napoli Federico II, Via Claudio 21, 80125, Naples, Italy.
†Laboratorio CINI-ITEM "Carlo Savy", Complesso Universitario Monte Sant'Angelo, Via Cinthia, 80126, Naples, Italy.
{antonio.bovenzi, cotroneo, roberto.pietrantuono, stefano.russo}@unina.it

***Abstract*—This study investigates software aging effects caused by the activation of concurrency bugs in a well-known database management system (DBMS), namely MySQL. Experiments with different workloads are performed in order to reproduce the most likely conditions for concurrency bugs activation. Besides the typical aging effects observed in many operational systems (i.e., a gradual degradation over time), results highlight that both available resources and DBMS performance (e.g. service rate, service time, and connection latency) can decrease with time in a hard-to-predict way. We observed that, due to the activation of concurrency bug, the DBMS enters a degraded state in which: *i)* the estimation of Time-To-Failure (TTF) by means of memory depletion trend analysis is highly inaccurate, and *ii)* the failure rate does not depend on the instantaneous and/or mean accumulated work. Results suggest that, in such cases, finer-grained indicators and/or different techniques need to be taken into account for properly preventing failures.**

***Keywords*-Software Aging, Concurrency Bug, Database**

I. Introduction

Current software applications are often built by integrating legacy and third-party components, by relying on intermediate middleware layers, and/or by extensively exploiting multithreading features. This trend, even bringing valuable advantages, leads to large and complex systems whose behavior is difficult to predict. A well-known phenomenon often observed in such systems is software aging. It mainly causes a continued and growing degradation of software internal state during its operational life, which can lead to progressive performance loss or even to system crash and hang. Aging has been experienced intensively in long-running applications such as web-servers [1], operating systems [2], spacecraft systems [3], and, by now, the phenomenon is recognized as a systematic non-negligible problem of mission- and safety-critical systems (e.g., memory leaks in NASA project[1]). Software aging is due to the activation of a particular class of bugs, known as *Aging-related bugs* (ARB), which have the peculiarity to cause an *increasing failure rate and/or degraded performance* once activated [3], [4]. Examples are those bugs causing resource leakage (e.g., unreleased file/locks, unterminated threads, and memory leaks), numerical round-off errors, and fragmentation problems (e.g., file system and physical memory).

[1]International space station (ISS) reboot: on-orbit status 03-26-11. www.nasa.gov/directorates/heo/reports/iss_reports/2011/03262011_prt.htm

ARBs are difficult to reproduce in a systematic way during the testing phase, because they require a long time to manifest their effects. For these characteristics, they are considered as a subclass of *Mandelbugs* [4]. Mandelbugs are those software faults that are difficult to isolate, and whose activation and/or error propagation are considered "complex" because: *i)* they depend on indirect factors (e.g., timing and/or sequencing of operations or inputs, interactions with system-internal environment), *and/or ii)* there is a time lag between the fault activation and the failure manifestation.

Due to the intrinsic difficulty in detecting ARBs during development, most of the literature on software aging proposes methods to best counteract the phenomenon during operation, thus trying to contain the *effect* of ARBs activation. The most studied and adopted method to this aim is software rejuvenation, which is a preventive maintenance activity aiming at restoring a clean state of the system before the failure occurrence.

However, the effects of ARBs activation might vary depending on the type of ARB, and this aspect is often neglected. From a recent survey [5], we observed that most of studies conducted so far analyze aging effects without addressing the nature of aging bugs. This leads to analyses in which ARBs are systematically activated at every execution with any workload (e.g., [6], [7]). These bugs, like any Mandelbug, are certainly hard to reproduce during testing, because their manifestation requires long time (i.e., the *propagation* is "complex"). Nevertheless, when the bugs *activation* is easy to reproduce, long-running experiments tend to regularly activate them, leading to observe linear or piecewise-linear degradation trends that researchers seamlessly use to estimate the time to failure. We still do not know if similar trends hold also for more complex software bugs.

This paper investigates software aging effects caused by the activation of concurrency bugs in Database Management System (DBMS). Concurrency bugs are a valuable example of such "complex" bugs; they represent a significant share of Mandelbugs in operational software systems [3] and a widely-studied class of bugs [8], [9]. The unpredictable activation of a concurrency bug could lead to complex and unknown propagation patterns able to cause aging effects for which current countermeasures may fail. Thus, analyzing their impact on aging phenomena would help in adopting the most proper action to avoid failures. Such an analysis is not trivial, because concurrency bug activation may depend on many factors: the number of threads competing for shared resources; the type of operation to perform on the resource (e.g. write or read); and the thread interleavings. It requires a comprehensive experimental plan in terms of: *i)* type of applied workload, *ii)* number and duration of experiments, and *iii)* monitoring ability, since more suitable aging indicators may be required in this case, along with the traditional ones (e.g., available free memory and response time).

By performing a set of measurements-based analyses, this work examines the impact of the activation of concurrency bugs on software aging. To this aim, an experimental campaign is designed and executed on the MySQL DBMS[2], which is a multi-threading software application deployed in many of the most popular Web Sites including Google, Facebook and YouTube, as well as in several enterprise applications [10]. The workload-based experiments were aimed at stressing MySQL extensively with various settings, so as to increase the likelihood of activating concurrency bugs and then investigate the aging effect.

Preliminary results highlighted, in any experiment, an improper memory consumption and a progressive performance degradation after many hours of execution, detecting the presence of aging in MySQL. However, besides these memory-related aging trends, we also observed atypical performance degradation in some experiments, which leads to the DBMS failure. We further investigated this behavior by inspecting the MySQL bug report (http://bugs.mysql.com/). We found that we reproduced a concurrency bug, which was ascertained to cause performance degradation (more details are in Section IV).

Starting from these results, further experiments have been performed to replicate the observed behavior and then analyze aging when such bugs are activated. To this aim, we have measured resource depletion and performance degradation, by taking into account several system-wide (coarse-grain) and application-specific (fine-grain) indicators. The analysis revealed that in such cases:
1) the estimation of the TTF by means of trend analysis is highly inaccurate; trends in resource depletion/performance degradation are not trivially linear or quasi-linear. Therefore rejuvenation techniques based on trend analysis, e.g., [2], [11], [12], are no longer effective to avoid the occurrence of failures;
2) the failure rate is not depending on the instantaneous and/or mean accumulated work; in other words, to heavier load does not necessarily correspond a greater failure rate. In such a case, traditional models capturing the failure rate as workload-dependent, e.g., [13], [14], [15], are not accurate in the TTF estimation.

[2]http://www.mysql.com/

In the rest of the paper we first present the existing literature on software aging (Section II); then, in Section III, we illustrate the case study, whereas in Section IV we describe the preliminary experiments for studying MySQL aging dynamics. Section V details the reproduction of Mandelbugs and its effects on performance degradation; in Section VI we discuss the key findings of the work and future direction.

II. Related Work

Recent studies analyzed ARBs, and recognized that they are a subclass of Mandelbugs [3], [4]. Aging factors, namely the combination of events causing the activation of ARB, were recognized to be complex since they can depend on: *i)* the environment, e.g, the hardware [16], the OS [2], but also on *ii)* the user behavior [17].

In the literature on software aging, besides these few studies on Bohrbug/Mandelbug classification, the nature of ARBs is not much addressed; authors mainly focused on conceiving optimal rejuvenation strategies once an ARB is activated. Since the first studies, the focus was: *i)* on determining the best rejuvenation scheduling, i.e., maximizing system availability, solving analytical system models, known as *Model-based* approach (e.g.,[13], [18]), or *ii)* on predicting the time to resource exhaustion by adopting statistical or machine learning techniques on data coming from system execution, known as *Measurements-based* approach [1], [6], [7], [19]. Remarkable attempts have been made to combine the benefits of both the previous approaches; hence, describing the phenomenon analytically, and determining the model's parameters through measurement (e.g., [14], [20], [21]). Since aging has been shown to be clearly correlated with workload variation, several authors also accounted for its impact, highlighting its dependency on aging effects and thus on the failure rate. Examples are in [2], [14], [22], where the estimate of the time-to-aging-failure at a given time varies in function of the workload actually experienced by the system.

Attempts to study aging from a different perspective are in the works by Matias et. *al.* [23], where authors try to accelerate the activation of aging bugs by means of accelerated life tests (ALT), and in [7], [24], where authors emulated aging effect by means of memory leak injection. Although these studies aimed at reducing the time for observing aging, they also deal with aging bugs whose activation conditions are not complex, e.g., bugs activated regularly by requesting a dynamic page to the web server [23], or by serving a specific number of requests in a servlet [7]: their effects on considered indicators (e.g., physical or process memory) were therefore linear or piecewise linear.

All the mentioned work did not deal with the underlying ARBs, but mostly with the mitigation and/or acceleration of aging effects. Differently from the past, this work studies the aging effects when a complex class of bugs, i.e., concurrency ones (such as race conditions, deadlocks or atomicity violation), are activated, in order to figure out if current countermeasures are effective in aging treatment and in avoiding system failures.

III. Case Study

MySQL is a multi-threading DBMS that manages every new connection by means of a separate thread, which contends for access to different shared data structures. At the startup, MySQL creates a pool of threads and keeps some of them in a thread cache. Since we focus on aging due to concurrency bugs, MySQL lends itself well to this type of analysis. The experimental testbed consists of a server, in which MySQL stable version 5.1 is executed, and two client machines where we launch the traffic generators that are in charge of making requests to the DBMS. Both clients and server are connected through a dedicated Gigabit Ethernet; thus there is no extraneous traffic generated during experiments. All the machines are equipped with Intel Core 2, Quad CPU Q8200, 2.33 GHz clock, 6GB of RAM, and 1 TB HD disk, running Scientific Linux Operating System 64 bit, kernel version 2.6.18-11 (http://www.scientificlinux.org/). The server has been configured to work with a minimal set of services, such as *ssh, ftp and vncserver*, so as to minimize the resource consumption not related to MySQL. As for the database structure, the load and the queries we have implemented the TPC-E benchmark specifications (http://www.tpc.org/tpce/default.asp), to model a brokerage firm with customers generating requests related to trades, account inquiries, and market researches.

IV. Preliminary Aging Analysis

As past studies demonstrated, resource depletion and performance degradation are the most important aging effects [6], [19], [25]. They can be measured by using several indicators. In particular, to measure memory depletion, the free physical memory (M_F) is commonly sampled. As for performance degradation, it can be measured at client side by sampling the success rate of requests (S_R). S_R accounts for the correct answers received over the number of requests submitted to the server in a time unit. In a perfect scenario, this indicator is always 1. It exhibits a negative trend if the server performance is degrading. To take the workload influence into account, different parameters have been considered to analyze MySQL aging. Following the procedure proposed in our previous study [22], we characterize MySQL workload with the following parameters:

- *Intensity*. It represents the stress level of the server. In absolute value, it is measured as number of queries executed per seconds.
- *Size*. It indicates the amount of data exchanged between the server and the clients (e.g., the amount of byte inserted or updated by a query, or data returned through a select);
- *Type*. It indicates the type of query that is processed by the server (e.g., select, join, update, insert); this may affect the different parts of the application code that can be exercised during experiments.
- *Variation*. This parameter allows to control if MySQL threads are concurrently serving different type of requests. For instance, in experiments with variation the threads that are serving different queries (e.g. a select or an update) are also competing for acquiring shared resources.

These parameters are used to plan the experiments with the aim of stressing the system in a broad way (i.e., with various workload configurations).

A. Workload-based Experiments

The objective of this initial analysis is to detect and to assess aging trends in the MySQL DBMS. The experiment planning is carried out by means of the Design of Experiments (DoE) technique [26], so as to have a set of orthogonal experiments with respect to the mentioned workload parameters. Two levels are defined for the considered workload parameters: *Low* and *High*. They respectively represent the unloaded and loaded operational modes, except for *Variation* (the level are in this case expressed by “yes” or “no”, indicating “*variation*” and “*no variation*”).

Given factors and levels, we generated the plan with the support of DoE aiding tool. Typically, a full factorial design, which accounts for the effects of all factors and all interactions on response variables, requires a large number of treatments; hence designers choose to reduce this number by ignoring the analysis of interactions among factors, and relying on tools guaranteeing the statistical significance of response [26]. Design reduced in such a way is referred to as fractional factorial design. We chose to ignore 3-factors interaction and obtained a fractional factorial design with a total of 8 treatments. Table I summarizes the considered experimental plan.

Table I: Experimental plan. L=Low, H=High

Exp	*Intensity*	*Size*	*Type*	*Variation*
1	L	H	L	Y
2	H	L	H	N
3	L	H	H	N
4	H	H	L	N
5	L	L	L	N
6	L	L	H	Y
7	H	L	L	Y
8	H	H	H	Y

Before executing the experiments, we need to “translate” *High* and *Low* levels of the parameters into concrete values. Moreover, to avoid failures due to excessive stressful workload (which is not related to aging failures), we also need to determine what is the maximum capacity for the system under test. To these aims, *capacity tests* are performed, according to the procedure described in our previous work [22], in order to define the limits of MySQL server deployed in our testbed. To evaluate the actual limit (i.e., maximum number of requests correctly served in a time unit), we need to take into account the levels of other workload parameters (such as request types and exchanged data size), because different limits can be reached depending on them. As for the *Size* parameter, as suggested in preliminary experimental analyses [26], we simply choose two values sufficiently distant to observe any impact of the workload parameter on aging indicators.

Table II: Results of Capacity tests, with *Size* parameter being Low and High, respectively

Query Type	**Intensity**	
	Size Low	*Size* High
Select	150	80
Update	150	25
Nested Select	150	30
Join	150	22

As for the request types, we consider four types of request, namely: *select*, *update*, *nested select*, *join*. The results of the capacity tests with these configurations are summarized in Table II, which reports, for each type of query and amount of data exchanged, the maximum number of requests served. Hence, depending on the configuration of request *Type* and *Size* parameters, the value of the *Intensity* of requests has to be chosen properly. The measures of the *Intensity* parameter is defined in [22] as the percentage of the maximum system's capacity. For High level we use the 80% of the limit, while the 10% of the limit is used for the Low level.

Finally, since we consider two levels (*Low* and *High*) for each parameter, we collapsed the four request types into two levels, i.e., a "loaded" and "unloaded" levels of the *Type* parameter. To assess wether a request type belongs to loaded or unloaded level, we sample, for each type, the average success rate, S_R, the average time elapsed to establish a connection, L_C, and the average time required for processing the request and send data back to the client, S_T. By sampling these values in a series of experiments, and clustering the results into two groups, we evaluate the request *complexity* as *High* or *Low*. To apply this procedure, we executed 4 short experiments (one for each type of request), each one repeated ten times. In such experiments, the selected type of request was submitted to the server for 5 minutes, by setting the levels of other parameters to *High*. We evaluated the average and the standard deviation of S_R, L_C, and S_T. By applying the *k-means* algorithm [27], with $k = 2$, on the resulting 40 samples, 2 clusters were obtained. In the former one, namely the unloaded cluster (*Low* level), there are the majority of *Select* and *Update* queries; in the latter (namely, the *High* level), *Nested select* and *Join* queries.

As for the *Variation* parameter, for the *No* level we randomly select one type of queries (namely, between *Select* or *Update* for the unloaded cluster, and between *Join* or *Nested select* for the loaded one). On the contrary, for the *Yes* level all the queries in a cluster were randomly executed. Based on these preliminary tests, we instantiated and then executed the treatments reported in Table I. The duration time of each experiment was set to 24 hours (for a total amount of 192 hours), since we observed a memory depletion trend within that time in the experiment with the least stressful workload (i.e., exp 5). Therefore, by assuming that the more intensive workload, the higher aging trends are [22], 24 hours suffice to observe aging effects in all the experiments.

B. Analysis of Preliminary Experiments

Aging indicators samples were collected each 30 seconds, then stored in *.csv* files. We use the Mann-Kendall test and the non-parametric Sen's procedure [28] to verify the hypothesis H1: *there is a trend in data*, and, if confirmed, to estimate the trend. We choose this procedure because, even if it is computationally heavy, it does not assume normally distributed measurement errors, and it is not sensitive to outliers.

Table III: Results of trend evaluation for each experiment

	M_F (KB/h)	S_R (Req/hour)	TTF (days)
1	$-1.44E + 04$	-	16
2	$-4.64E + 03$	$-7.88E - 03$	50
3	$-1.12E + 04$	$-9.81E - 04$	21
4	$-5.20E + 03$	$-2.46E - 02$	45
5	$-4.74E + 02$	-	492
6	$-2.97E + 04$	-	8
7	$-1.45E + 04$	-	16
8	$-1.12E + 04$	-	21
Avg	$-1.14E + 04$	$-4.18E - 03$	20

Table III lists significative trends for the M_F and S_R indicators for each experiment. Figure 1 and 2 show snapshots of some resource depletion and performance degradation trends, which were observed during the experiments. In particular, Figure 1 shows M_F for experiments 5, 6 and 7; Figure 2, plots the average throughput for the experiment 4. Results show that the most relevant trends occurred for memory depletion. The phenomenon is not negligible, since the TTF may reach even 8 days (see exp 6 of Table III). As for performance-related indicator, we also found some trends (in experiments 2,3,4) but not in all the experiments. The most severe performance degradation was experienced in the treatment number 4, with a trend of performance loss of about 2.5E-02 requests per hour. In this experiment, we also observed a DBMS failure; the server stopped serving requests even if the MySQL process did not crash.

Observing the behavior in experiments 2, 3 and 4, performance loss is apparently in contrast with resource depletion; server rate losses are observed together with the less relevant memory depletion trends, and always with a *Variation* level low (whereas the most relevant resource depletion trend was revealed in experiments with High *Variation*). To explain this behavior, as well as the failure observed in experiment 4, and to verify if it is caused by complex bugs, we inspected the MySQL bug report. As discussed by MySQL developers, we found that the observed degradation, and the resulting failure, are caused by the activation of the bug #22868, which has been classified as a concurrency bug with server impact on the system performance. In fact, this fault is activated under *hard-to-reproduce* conditions related to heavy I/O-bound workloads (more than 50 threads concurrently serving queries) such as the ones recreated in experiment 4. To better analyze aging effects due to this type of bugs, further experiments have been planned. These are discussed in the next section.

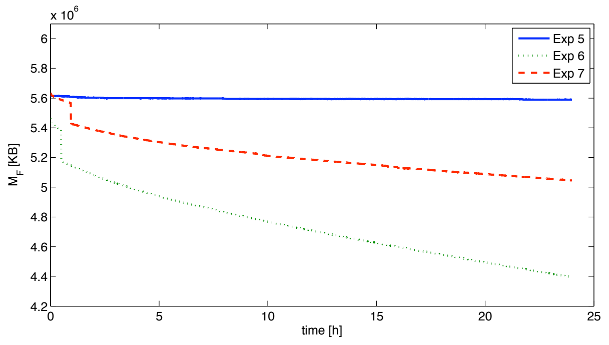

Figure 1: Snapshot of experiments with some relevant memory depletion trends

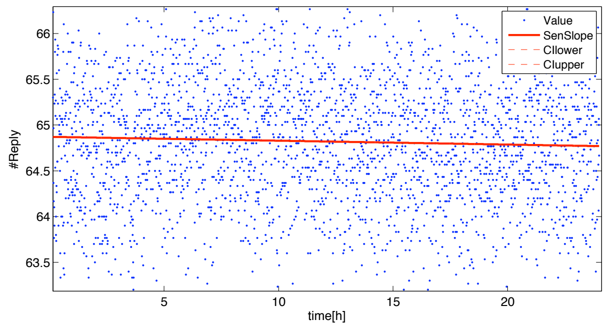

Figure 2: Snapshot of experiments 4, which has the most relevant performance degradation trend

V. Aging Analysis due to Complex Bug

The previous experiments, other than revealing the presence of aging in the memory depletion trends, show that in some cases also performance degradation was experienced. In one experiment we also observed a DBMS failure. Data analysis and bug reports revealed that the failure is due to a concurrency bug. In this Section, we exploit these first results to reproduce more likely conditions activating this concurrency bug: the goal is to analyze the effect of its activation on aging trends. First, we identify what are the potential indicators most suitable to detect the bug activation and the aging effects in this case. Then, new experiments are planned and executed trying to replicate the Mandelbug activation; finally, a cross-analysis is carried out with respect to the selected indicators.

A. Preparation phase

1) Aging indicators: to better understand the observed behavior we monitor other indicators during system execution, which could be useful to detect the bug activation and its effect on the system. In particular, we monitor indicators related to resources consumed by the MySQL process. Examples of such indicators include: virtual memory size, memory actually in RAM, current number of allocated threads, number of file handlers, shared memory segments, semaphores. As for performance indicators, we monitor: the success rate of requests (S_R); the average response time to establish a connection (LC); the average response time to process the request and send data back to the client (ST). L_C and S_T, which regard both response time, are accounted separately in order to distinguish the following two cases: *i)* response times are high because the DBMS is still serving old accepted requests (i.e., the DBMS is busy); *ii)* response times are high because the DBMS, even being not busy, is not using available resources in the best manner. The exhaustive list of memory and performance related indicators is reported in Table IV.

Table IV: Monitored variables during system operation

Name	Description	Name	Description
VmPeak	Virtual memory Peak	Wrtback	Memory written back to the disk
VmSize	Virtual memory	AllocFH	#allocated file handler
VmLck	Locked memory	Proc-fd	#file descriptors
VmHWM	Peak resident set size	Slab	In-kernel data structures cache
VmRSS	Resident set size	PgTab	Amount of memory dedicated to the lowest level of page
VmLib	Shared library code size	CommAs	Memory allocated, but not used
VmPTE	Page table entries size	Shm	#shared memory segments
Threads	#Threads in the process	Sem	#semaphores
M_F	Available Memory in RAM	Queue	#queues
Buffers	Relatively temporary storage for raw disk blocks	AnonPg	Non-file backed pages mapped into userspace page tables
Cached	In-memory cache for files read from the disk	Mapped	Files which have been mapped such as libraries
Swap Cached	Mem. swapped back in but still in the swapfile	L_C	Time elapsed to accept a new connection
Active	Most recently used memory	S_T	Time for processing the request and send data
Inactive	Least recently used memory	S_R	Percentage of requests that are correctly served
Dirty	Non-file backed pages mapped into userspace page tables	Errors	#errors returned to the clients

2) Experiments: to increase the probability of reproducing concurrency bugs, we increase the duration of the experiments, and recreate the experienced conditions of activation. Therefore, we planned longer experiments, lasting each one 60 hours, and set the level of workload parameters according to the results of the previous analysis. Specifically, Table III in the previous Section highlighted that the most severe performance degradation is experienced in the treatment number 4, whose parameters configuration was: *Intensity*, *Size* to high, *Type* to low, and no *Variation*. In the following two experiments, number 9 and 10, we adopted the same setting. To verify if the performance degradation also persists under a lighter workload, we act as follows: if we detect a degradation in the first 30 hours of the experiment, then, at the 30^{th} hour, we decreased the load by setting the *Intensity* and *Size* to the low level. Hence we measure performance degradation again during this second interval of observation. Moreover, to verify if the activation of concurrency bugs is due just to the heavy workload, we also performed one more long running experiment with a "light" load, in which we set *Intensity*, *Size* and *Type* to low, and no *Variation* (experiment number 11).

Summarizing, experiments 9 and 10 are performed to activate concurrency bugs, having the following parameter setting: intensity H, size H, type *select* and variation *No*. Instead the experiment 11 is performed to verify if concurrency bugs can be activated with a lighter workload, namely intensity L, size L, type *select* and variation *No*.

B. Analysis of Results

By performing the new experiments we observed again i) the activation of the concurrency bug ii) memory depletion and performance degradation trends, and iii) database failures (the database cannot serve any incoming requests). Memory depletion trends for experiment 9, 10 and 11 are respectively: -1.51E+03, -1.06E+04 and -1.62E+03; instead performance degradation trends, in terms of request per hour, are: -2.32E-02, -1.03E-02 and -8.26E-07. It is worth noting that S_R trends were computed considering either the end of the experiment or the system failure. Figure 3 shows that: *i)* in the experiment 9, although the load was drastically decreased at its 30^{th} hour of operations, the DBMS unexpectedly failed at about the 35^{th} hour of execution: the concurrency bug activation caused the DBMS entering a state in which the workload is no longer influential to determine the degradation trend; *ii)* in the experiment 10, we observed a failure earlier, at the 7^{th} hour of operation; *iii)* in the experiment with light load (i.e., number 11) the DBMS failed at its 32^{nd} hour of operation. The latter failure was transient, because the DBMS restarted serving requests again after few minutes (there is a spike in the throughput, see Figure 3).

Results of trend analysis for these treatments is summarized in Table V; indeed, they highlight that TTFs estimated

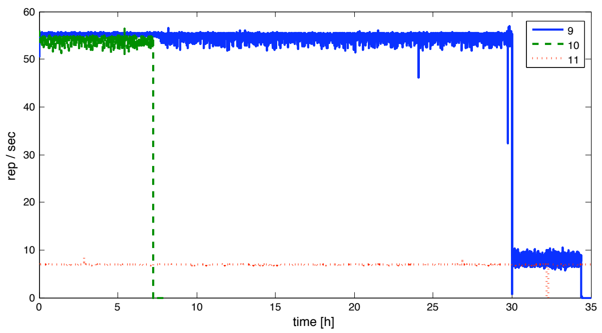

Figure 3: Service rate in long running experiments. Failures can be observed at about the 8^{th}, 32^{th} and 35^{th} hours

Table V: Trends of memory depletion and performance degradation, estimated TTF and observed failure time

Exp	M_F (KB/h)	S_R (Req/hour)	Estimated TTF (days)	Failure Time (h)
9	$-1.51E+03$	-2.32E-02	154	35
10	$-1.06E+04$	$-1.03E-02$	22	7
11	$-1.62E+03$	$-8.26E-07$	143	32

by means of memory depletion trend analysis are highly inaccurate when this kind of bugs are activated. Therefore different indicators and techniques should be taken into account for preventing DBMS failures. To figure out what are the most suitable indicators to detect this behavior among the monitored ones (see the previous Section, Table IV), we perform both correlation and graphical analyses; then, we examine collected data with the selected indicators. Correlations between monitored indicators, M_F, and S_R is carried out by computing Pearson and Spearman coefficients [29] in each treatment experiencing performance degradation, i.e., number 2, 3, 4, 9, 10, 11. Table VI lists for each experiment the top ten correlations, which are significant at $\alpha = 0.05$ (i.e., 95% confidence level), between M_F and the collected indicators. Results show that memory-related variables were highly correlated to M_F. Namely, if $Cached$, $Buffers$, $Slab$ and $VmHWM$ increased, available free memory decreased (negative correlation). This correlation applies almost for each experiment. Such a behavior is expected because it is consistent with the way Linux manages the available memory. Trends in the cache variables are not related with aging, because the operating system stores data read recently from the disk in the main memory (i.e., the cache) to speed up successive accesses.

As for performance degradation (see Table VII), correlation coefficients turns out to be much lower than the ones for memory. However, some of the indicators correlating with M_F are also correlated with S_R, such as $VmHWM$ and $Threads$. Thus we focus the attention on these indicators. Figure 4 shows the behavior of the peak of MySQL memory,

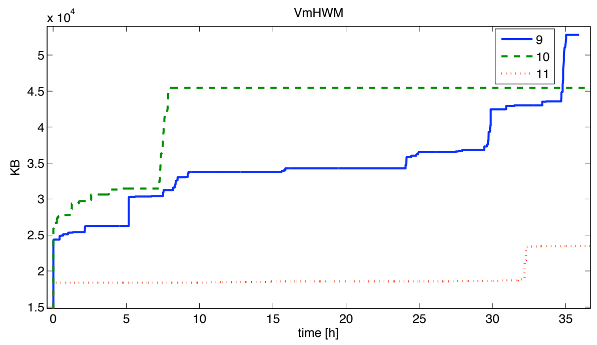

Figure 4: VmHWM in long running experiments. Failures can be observed at about the 8^{th}, 32^{th} and 35^th hours

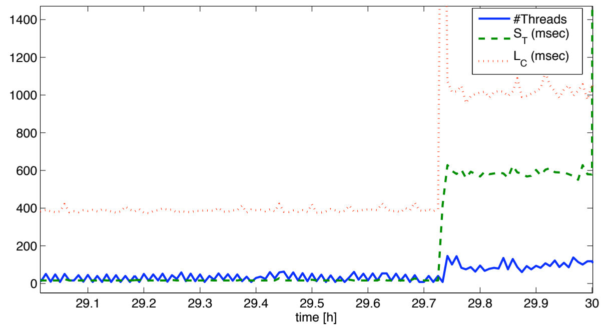

Figure 5: Snapshot of experiment 9 S_T and L_C vs. $Threads$

i.e. $VmHWM$, during the experiments. It can be noticed that there was a sudden increase just few minutes before every failure. At the same time, the other indicator, i.e., $Threads$, had an abnormally increase, hence the process memory increment was due to the data structures allocated by the creation of novel threads. Analyzing the number of threads against the two performance indicators, S_T and L_C (Figure 5), it can be noted that as soon as the connection latency and the service time increased, the very same behavior is noticed for threads, which started increasing in order to augment the DBMS ability in serving requests. However, the system service rate (S_R) did not get any benefit form this thread increasing (see Figure 6); instead, it became even worse.

Figure 6 shows the normalized number of threads and the throughput (i.e., S_R multiplied for the number of queries requested) for experiment 9. At about the 29^{th} hour of operation, the number of MySQL threads suddenly increased (the non-normalized value is about 150), and we observed a performance loss in terms of throughput. As explained before, at the 30^{th} hour the number of client requests per seconds was decreased from 56 to 8 and the size of exchanged data from $100Kb$ to $1Kb$ for each query. Although the applied workload was lighter, the L_C and S_T indicators show a positive trend, respectively, 9.89E01 and

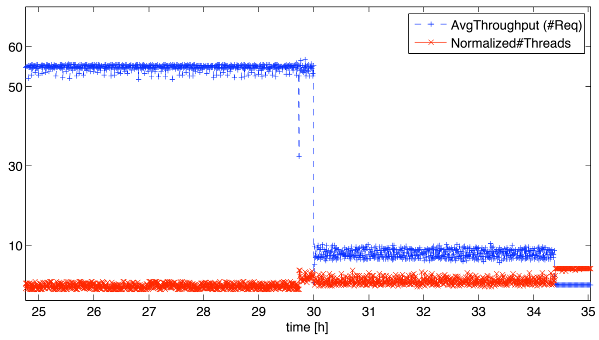

Figure 6: Snapshot of experiment 9: S_R vs. $Threads$

6.02E01 $msec$ per hour, hence confirming that the system performance are degrading. Moreover, when the number of threads increased again around the 35^{th} hour, the DBMS stopped serving incoming requests. A similar behavior was also observed in the other experiments.

The multiple indicators analysis let us conclude that the observed performance degradation, which can be reproduced by increasing client requests and by using I/O bound workloads, is due to the activation of the discussed concurrency bug: when some particular conditions occur (e.g., several threads are serving I/O requests and the connections/s to MySQL exceeds the thread cache) and MySQL cannot satisfy the incoming requests, the server increases the number of threads to satisfy the requests. However performance continues decreasing with time, independently from the applied workload, because lots of threads perform some CPU processing, wait for I/O, and then, after request processing is completed, block again waiting for a new request.

As a further step, we evaluate M_F trends. In particular, we select experiments in the number of threads increment and the performance degradation were observed, i.e., $4,\ 9,\ 10$. Hence, the analysis is performed on two disjoint intervals, namely before and after the observed increment of threads. Results in Table VIII show that, when memory depletion trends were higher (in absolute value), the system entered in the degraded state earlier. However, in contrast to what one might expect, during performance degradation, namely after thread increasing, we did not observed a greater memory depletion trend, except for one experiment. This confirms that the analysis of memory depletion trends by itself does not suffice to avoid the failure of the DBMS.

To conclude the analysis, we conducted a further experiment in which we try to remove/mitigate the aging effect. In this experiment the DBMS failed because of the excessive increment of threads (about 150); however, when we restarted the network interface the number of threads suddenly decreased and the DBMS restarted serving requests. This is because the network restart causes the termination of all MySQL threads and the re-initialization of internal data structures of MySQL kernel, which manages client requests.

Table VI: Top ten Pearson and Spearman coefficients between M_F and indicators

	2		3		4		9		10		11	
Pearson	Buffers	-1,00	Active	-0,99	Slab	-0,99	Cached	-0,94	Buffers	-0,98	Buffers	-0,94
	Slab	-0,99	Buffers	-0,98	Cached	-0,98	Active	-0,94	Active	-0,98	Slab	-0,91
	Active	-0,95	Slab	-0,84	Inactive	-0,98	Buffers	-0,92	Inactive	0,97	Active	-0,89
	Cached	-0,81	Inactive	-0,72	Active	-0,93	Slab	-0,81	Cached	-0,93	Cached	-0,88
	Mapped	0,60	Mapped	0,68	Buffers	-0,76	VmPTE	-0,77	S_R	0,77	VmPTE	-0,70
	PgTab	0,56	PgTab	0,65	VmSize	-0,54	VmSize	-0,76	L_C	0,77	VmPeak	-0,69
	VmPTE	-0,54	VmHWM	-0,65	CommAS	-0,54	VmPeak	-0,76	Slab	0,73	VmSize	-0,69
	VmSize	-0,52	VmPeak	-0,62	VmPeak	-0,52	VmHWM	-0,73	S_T	-0,63	CommAS	-0,68
	VmRSS	-0,34	Cached	-0,50	AnonPg	-0,48	VmRSS	-0,72	Proc-fd	0,54	VmHWM	-0,68
	AnonPg	-0,31	VmPTE	-0,47	VmPTE	-0,45	CommAS	-0,70	Threads	0,51	VmRSS	-0,61
Spearman	Active	-1,00	Active	-0,99	Cached	-1,00	Active	-0,99	Active	-0,98	Active	-1,00
	Buffers	-1,00	Buffers	-0,99	Buffers	-1,00	Buffers	-0,98	Buffers	-0,98	Buffers	-1,00
	Cached	-1,00	Slab	-0,98	Active	-1,00	VmHWM	-0,96	Inactive	0,98	Errors	0,66
	Slab	-0,98	VmHWM	-0,76	VmHWM	-0,98	VmRSS	-0,96	Cached	-0,98	Wrtback	-0,65
	VmSize	-0,62	VmPeak	-0,76	VmPeak	-0,95	VmSize	-0,96	S_R	0,60	Slab	-0,63
	VmPTE	-0,59	Inactive	-0,40	Slab	-0,95	VmPTE	-0,96	AnonPg	-0,60	S_R	0,61
	VmRSS	-0,41	Mapped	0,37	Mapped	-0,89	VmPeak	-0,96	Slab	0,52	L_C	0,57
	AnonPg	-0,37	VmPTE	-0,36	VmPTE	-0,87	Cached	-0,88	Proc-fd	0,51	VmHWM	-0,49
	CommAS	-0,32	Cached	-0,20	VmSize	-0,85	AnonPg	-0,87	L_C	0,50	Inactive	0,47
	Mapped	-0,20	PgTab	0,17	VmRSS	-0,80	CommAS	-0,86	Threads	0,49	VmRSS	-0,44

Table VII: Top ten Pearson and Spearman coefficients between S_R and indicators

	2		3		4		9		10		11	
Pearson	L_C	-0,80	S_T	-0,51	VmRSS	-0,83	VmHWM	-0,77	L_C	1,00	Proc-fd	-0,68
	VmHWM	-0,52	L_C	0,29	AnonPg	-0,77	AnonPg	-0,75	Slab	0,97	Threads	-0,65
	VmPeak	-0,52	Dirty	0,17	S_T	-0,76	VmRSS	-0,74	Buffers	-0,85	Active	-0,28
	Errors	0,52	Cached	0,16	L_C	-0,75	VmPTE	-0,67	Active	-0,84	Buffers	-0,24
	Cached	-0,36	Inactive	0,15	VmHWM	-0,72	VmSize	-0,65	Inactive	0,84	VmRSS	-0,23
	S_T	0,29	Errors	0,11	VmSize	-0,68	CommAS	-0,65	S_T	-0,81	VmPTE	-0,23
	VmSize	-0,28	VmPeak	-0,09	VmPTE	-0,65	VmPeak	-0,64	M_F	0,77	VmSize	-0,23
	VmPTE	-0,28	VmHWM	-0,09	Proc-fd	-0,63	L_C	-0,61	Cached	0,77	AnonPg	-0,23
	AnonPg	-0,25	Threads	-0,07	VmPeak	-0,62	Threads	-0,49	Proc-fd	-0,76	CommAS	-0,21
	VmRSS	-0,25	Proc-fd	-0,06	Threads	-0,60	Slab	-0,47	Threads	0,70	M_F	0,19
Spearman	L_C	-0,70	S_T	-0,49	VmPTE	-0,59	Slab	-0,51	Slab	0,66	Errors	0,84
	S_T	-0,11	Cached	0,35	VmSize	-0,58	Wrtback	-0,50	Buffers	-0,65	Wrtback	-0,83
	Inactive	0,08	L_C	0,33	VmPeak	-0,54	Cached	-0,47	Inactive	0,65	Buffers	-0,64
	VmPeak	-0,06	Inactive	0,27	VmHWM	-0,53	Errors	-0,46	Cached	-0,65	Active	-0,64
	VmHWM	-0,06	Dirty	0,21	CommAS	-0,52	Inactive	0,43	Active	-0,65	L_C	0,63
	Dirty	0,05	VmSize	0,10	Buffers	-0,52	S_T	-0,34	L_C	0,64	M_F	0,61
	AnonPg	-0,05	VmRSS	0,10	Cached	-0,52	AnonPg	-0,28	Proc-fd	0,63	Inactive	0,29
	VmRSS	-0,04	AnonPg	0,10	Active	-0,52	Threads	-0,26	Threads	0,62	Mapped	0,14
	Cached	-0,04	CommAS	0,06	M_F	0,52	CommAS	-0,25	S_T	-0,61	Cached	0,12
	Buffers	-0,04	Threads	0,05	VmRSS	-0,51	M_F	0,23	M_F	0,60	AllocFH	-0,12

Table VIII: Significative trends (p-value $<< 0.05$) in the M_F before and after $Threads$ increment

Id	1^{th} Interval				2^{th} Interval			
	Length (h)	$slope - 5\%$	$slope\ (KB/h)$	$slope + 5\%$	Length (h)	$slope - 5\%$	$slope\ (KB/h)$	$slope + 5\%$
4	15,5	-6.34E+03	-6.26E+03	-6.18E+03	9	-3.02E+03	-2.90E+03	-2.79E+03
9	29	-1.58E+03	-1.55E+03	-1.53E+03	6	-6.68E+02	-5.79E+02	-4.92E+02
10	7	–9.78E+03	-9.68E+03	-9.58E+03	1	-4.46E+04	-4.06E+04	-3.16E+04

Summarizing, the analysis shows that, as a consequence of a complex bug activation: *i)* once the average number of threads increases, MySQL is no longer capable to sustain the load; *ii)* however, even by reducing the number of requests, no gain is perceived because the server enters in a degraded state; *iii)* in response to this, it creates more threads to satisfy incoming requests, but the server performance, in terms of S_R, L_C, S_T, does not increase with time; on the contrary it decreases, with no relation with the applied workload, and eventually no more requests are served. This behavior can be observed most probably under intensive I/O bound workloads (i.e., serving more than 50 concurrent inserts) that causes MySQL threads to content shared resources in the MySQL kernel. This situation was also experienced by MySQL developers, which called it *Thread Thrashing*, and it was recognized as a server performance issue particularly difficult to reproduce (bug #22868). As countermeasure, the restart of the network interface proved to be a good mean to restore a clean state for the database, and, since generally it has very small overhead compared to system reboot and/or database restarting, it could been preferred for preventing DBMS aging-related failures caused by concurrency bugs.

VI. Discussion

The presented study has investigated the aging phenomenon in the MySQL DBMS. We have conducted workload-dependent experiments with the goal of highlighting aging trends and evaluating aging effects when complex bugs are activated. Our findings are as follows:

1) MySQL DBMS is affected by software aging. The eight experiments presented in Section IV highlighted a memory depletion trend in each experiment, with an estimated TTF that, depending on the workload applied, may amount to a minimum of 8 days (worst case) up to about 500 days (best case). This highlights that the phenomenon might be severe, and may affect of long-running software applications running on top of MySQL.

2) The phenomenon is confirmed to be related to the workload. In preliminary experiments MySQL, aging trends and the estimated TTFs vary when different workload patterns are applied. So far, many other systems affected by aging experienced a similar behavior. Such type of trends allows for accurate estimate of the expected TTFs. However, by looking at performance degradation of some treatments, we get inaccurate TTF estimates and observed atypical behaviors that we investigated in the successive phase of the analysis.

3) The considered workload parameters can be tuned to increase the likelihood of activating concurrency bugs. The analysis of the aging trends of first experiments and the inspection of MySQL bug report suggest that the performance degradation, and hence the database failure, were related to the activation of concurrency bug. Thus, we planned further specific experiments, in which the workload parameters were configured to activate concurrency bugs. The activation of concurrency bugs is reproduced in three experiments: we observed that the DBMS enters a degraded performance state, i.e., the *thread thrashing* state, which eventually leads to the failure of the DBMS.

4) TTF estimation by means of memory depletion analysis is very inaccurate when MySQL is experiencing performance degradation (Table V). This implies that, in such a case, the rate of memory depletion is not the best aging indicator. As consequence, rejuvenation techniques only based on memory depletion analysis (e.g., [14]) could not be effective to avoid the occurrence of failures. Instead, finer-grained indicators, such as the number of threads in our case, should be taken into account to prevent the failure.

5) Workload-based models, capturing the failure rate as dependent on the instantaneous and/or mean accumulated work, could be not accurate in the TTF estimation. In other words, when more complex ARBs are activated, we observed that, no matter the workload actually applied or served, the failure rate increases (e.g., cf. with Figure 3). This may suggest to design different aging treatments: workload-based models could be used to estimate TTF during "normal" memory depletion trends; specific detectors may be used to reveal atypical aging behaviors, and hence to trigger the most proper countermeasures. In our case, when an anomaly in the number of threads was detected, the restarting of the networking interface was performed for avoiding the failures.

In future studies we will analyze aging effects when different types of ARB are activated. Our aim is to enforce the knowledge on potential relationships between the root cause of aging, e.g., a concurrency bug, and the observed effects. Indeed, the conducted experiments showed that different types of ARBs may require different detection strategies, e.g., by monitoring additional indicators and/or by using different models and statistical techniques for TTF estimation. Thus, we will focus our analyses on the activation of complex aging bugs on different systems, such as middleware and application servers, in order to verify to what extent the results of this study can be generalized.

Acknowledgment

This work has been partially supported by the Italian industrial research project "Iniziativa Software CINI-Finmeccanica" and by the Italian Ministry for Education, University, and Research (MIUR) in the framework of the Project of National Research Interest (PRIN) DOTS-LCCI: Dependable Off-The-Shelf based middleware systems for Large-scale Complex Critical Infrastructures 2008 (dots-lcci.prin.dis.unina.it).

References

[1] L. Li, K. Vaidyanathan, and K. S. Trivedi, "An approach for estimation of software aging in a web server," in *Proc. the Int. Symp. Empirical Soft. Engineering*, 2002, pp. 91–100.

[2] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, "Software aging analysis of the linux operating system," in *IEEE 21st Int. Symp. Soft. Reliability Engineering*, 2010, pp. 71–80.

[3] M. Grottke, A. Nikora, and K. Trivedi, "An empirical investigation of fault types in space mission system software," in *IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN)*, 2010, pp. 447 –456.

[4] M. Grottke and K. S. Trivedi, "Fighting bugs: Remove, retry, replicate, and rejuvenate," *Comp.*, vol. 40, pp. 107–109, 2007.

[5] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, "Software aging and rejuvenation: Where we are and where we are going," in *IEEE Third Int. Workshop on Software Aging and Rejuvenation (WoSAR)*, 2011, pp. 1 –6.

[6] R. Matias and P. Filho, "An experimental study on software aging and rejuvenation in web servers," in *30th Annual Int. Computer Software and Applications Conference*, vol. 1, 2006, pp. 189–196.

[7] J. Alonso, J. Torres, J. L. Berral, and R. Gavalda, "Adaptive on-line software aging prediction based on machine learning," *Int. Conf. on Dependable Systems and Networks*, vol. 0, pp. 507–516, 2010.

[8] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing heisenbugs in concurrent programs," in *Proc. the 8th USENIX conference on Operating systems design and implementation*, 2008, pp. 267–280.

[9] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *Proc. the 13th Int. Conf. Architectural support for programming languages and operating systems*, 2008, pp. 329–339.

[10] [Online]. Available: http://www.mysql.com/why-mysql/white-papers/

[11] M. Grottke, L. Li, K. Vaidyanathan, and K. Trivedi, "Analysis of software aging in a web server," *Reliability, IEEE Trans. on*, vol. 55, no. 3, pp. 411 –420, 2006.

[12] G. Hoffmann, K. Trivedi, and M. Malek, "A best practice guide to resource forecasting for computing systems," *Reliab., IEEE Trans. on*, vol. 56, no. 4, pp. 615–628, 2007.

[13] S. Garg, A. Puliafito, M. Telek, and K. Trivedi, "Analysis of preventive maintenance in trans. based software systems," *Comp., IEEE Trans. on*, vol. 47, no. 1, pp. 96 –107, 1998.

[14] K. Vaidyanathan and K. Trivedi, "A comprehensive model for software rejuvenation," *Dependable and Secure Computing, IEEE Trans. on*, vol. 2, no. 2, pp. 124 – 137, 2005.

[15] D. Wang, W. Xie, and K. S. Trivedi, "Performability analysis of clustered systems with rejuvenation under varying workload," *Perform. Eval.*, vol. 64, pp. 247–265, 2007.

[16] M. Zhivich and R. K. Cunningham, "Secure systems: The real cost of software errors," *IEEE Security and Privacy*, vol. 7, no. 2, pp. 87–90, 2009.

[17] I. Cisco Systems, "Cisco security advisory: Cisco catalyst memory leak vulnerability. document id: 13618," 2001.

[18] S. Garg, C. Kintala, Y. Huang, and K. Trivedi, "Minimizing completion time of a program by checkpointing and rejuvenation," in *Proc. the Int. Conf. on Measurement and modeling of computer systems*, vol. 24, no. 1, 1996, pp. 252–261.

[19] L. Silva, H. Madeira, and J. Silva, "Software aging and rejuvenation in a soap-based server," in *Int. Symp. Network Computing and Applications*, 2006, pp. 56 –65.

[20] K. Vaidyanathan and K. S. Trivedi, "A measurement-based model for estimation of resource exhaustion in operational software systems," in *Proc. the 10th Int. Symp. Software Reliability Engineering (ISSRE)*, 1999, pp. 84–93.

[21] H. Eto, T. Dohi, and J. Ma, "Simulation-based optimization approach for software cost model with rejuvenation," *Lect. Not. in Comp. Science*, vol. 5060 LNCS, pp. 206–218, 2008.

[22] A. Bovenzi, D. Cotroneo, R. Pietrantuono, and S. Russo, "Workload characterization for software aging analysis," in *22nd Int. Symp. Software Reliability Engineering*, 2011, pp. 240–249.

[23] R. Matias, K. Trivedi, and P. Maciel, "Using accelerated life tests to estimate time to software aging failure," in *IEEE 21st Int. Symp. Soft. Reliability Engineering*, 2010, pp. 211–219.

[24] J. Zhao, Y. Jin, K. Trivedi, and R. Matias, "Injecting memory leaks to accelerate software failures," in *22nd Int. Symp. Soft. Reliability Engineering*, 2011, pp. 260 –269.

[25] A. Avritzer, A. Bondi, M. Grottke, K. S. Trivedi, and E. J. Weyuker, "Performance assurance via software rejuvenation: Monitoring, statistics and algorithms," in *Proc. the Int. Conf. on Dependable Sys. and Networks*, 2006, pp. 435–444.

[26] D. Montgomery, *Design and Analysis of Experiments*, ser. Student solutions manual. Wiley, 2008.

[27] M. Steinbach, G. Karypis, and V. Kumar, "A comparison of document clustering techniques," in *In KDD Workshop on Text Mining*, 2000.

[28] P. K. Sen, "Estimates of the regression coefficient based on kendalls tau," *Journal of the American Statistical Association*, vol. 63, no. 324, pp. 1379–1389, 1968.

[29] N. E. Fenton, *Software Metrics: A Rigorous and Practical Approach*, 2nd ed. Boston, MA, USA: Int. Thomson Computer Press, 1996.