A

A Survey of Software Aging and Rejuvenation Studies

DOMENICO COTRONEO, ROBERTO NATELLA, ROBERTO PIETRANTUONO,
STEFANO RUSSO, Università degli Studi di Napoli Federico II

Software aging is a phenomenon plaguing many long-running complex software systems, which exhibit performance degradation or an increasing failure rate. Several strategies based on the proactive *rejuvenation* of the software state have been proposed to counteract software aging and prevent failures. This survey paper provides an overview of studies on Software Aging and Rejuvenation (SAR) that appeared in major journals and conference proceedings, with respect to the statistical approaches that have been used to forecast software aging phenomena and to plan rejuvenation, the kind of systems and aging effects that have been studied, and the techniques that have been proposed to rejuvenate complex software systems. The analysis is useful to identify key results from SAR research, and it is leveraged in this paper to highlight trends and open issues.

Categories and Subject Descriptors: C.4 [**Performance of Systems**]: Reliability, availability, and serviceability

General Terms: Reliability, Performance

Additional Key Words and Phrases: Software Aging; Software Rejuvenation; Performance Degradation; Aging-Related Bugs; Software Aging Literature

1. INTRODUCTION

The *software aging* phenomenon consists in the increase of the failure rate or performance degradation of a system as it executes, which can be due to the accumulation of errors in the system state and to the consumption of resources such as physical memory [Huang et al. 1995; Grottke et al. 2008]. This phenomenon has been known by practitioners since a long time. Early evidences of software aging were found already in the 1960s: the Safeguard military system was affected by hangs occurring once error reporting buffers were full [Bernstein and Kintala 2004]. As software has grown in size and complexity, software aging has been observed in an increasing number of long-running systems, including telecommunication switching and billing software [Huang et al. 1995; Avritzer and Weyuker 1997], and it led to the well-known accident of the Patriot anti-missile system that caused the loss of human lives [Marshall 1992]. Software aging can be attributed to elusive *software bugs*: studies in the early 1990s by Lawrence Bernestein on telecommunication systems [Bernstein 1993] pointed out the high incidence of bugs that, when triggered, do not immediately cause a software failure, but manifest themselves as memory leakage, unreleased file locks, data cor-

This work was partially supported by the FP7 project *CRITICAL-STEP* (http://www.critical-step.eu), Marie Curie Industry-Academia Partnerships and Pathways (IAPP) n. 230672, funded by the European Commission, and by the *TENACE* PRIN project (n. 20103P34XC) funded by the Italian Ministry of Education, University and Research.
Authors' email addresses: {cotroneo, roberto.natella, roberto.pietrantuono, sterusso}@unina.it
Authors' postal address: Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione (DIETI), Università degli Studi di Napoli Federico II, Via Claudio 21, 80125, Naples, Italy.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© YYYY ACM 1550-4832/YYYY/01-ARTA $10.00
DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

ruption and numerical error accumulation, making the system to slowly degrade its performance and to eventually fail. Often, such bugs are too subtle or too costly to be removed during development. Research at the AT&T Bell laboratories on fault-tolerant software then identified *Software Rejuvenation* as a cost-effective solution to counteract Software Aging [Huang and Kintala 1993; Huang et al. 1995; Wang et al. 1995; Bernstein and Kintala 2004]. Software Rejuvenation is a *proactive* approach for preventing performance degradation and failures due to Software Aging: it consists in the occasional or periodical clean-up of aging effects (which can be achieved by a simple software restart, or by more complex techniques), in order to postpone failures and to restore performance. Software Rejuvenation represented a novel form of preventive software maintenance compared to other forms of preventive software maintenance existing at that time [Kajko-Mattsson 2001], which were focused on the installation of updates in order to prevent field failures [Adams 1984] or re-engineering a software program in order to cope with its obsolescence [Parnas 1994].

Since early studies on Software Aging and Software Rejuvenation by the AT&T Bell laboratories in the 1990s, much efforts have been devoted to characterize the software aging phenomenon and to devise cost-effective rejuvenation strategies. After two decades, a significant body of knowledge has been established and an international community of researchers in the area of Software Aging and Rejuvenation (SAR) has grown. It thus becomes important to look at what has been done, in order to identify what has still to be accomplished to make industry practitioners aware of the Software Aging problem and to foster the adoption of Software Rejuvenation approaches, and which are the future challenges for researchers of the SAR community.

Starting from our preliminary analysis in [Cotroneo et al. 2011a], this paper provides a comprehensive analysis of SAR literature, by reviewing 495 papers that were published in the fields of software engineering and software dependability. The aim of this paper is to provide an overall picture of the state-of-the-art in Software Aging and Rejuvenation. We survey relevant studies with respect to the approaches that have been used to forecast software aging phenomena and to plan rejuvenation, the kind of systems and aging symptoms that have been studied, and the techniques that have been proposed to rejuvenate complex software systems. The analysis is useful to identify key results from SAR research, and it is leveraged in this paper to highlight trends and open issues deserving attention in the near future by the SAR community.

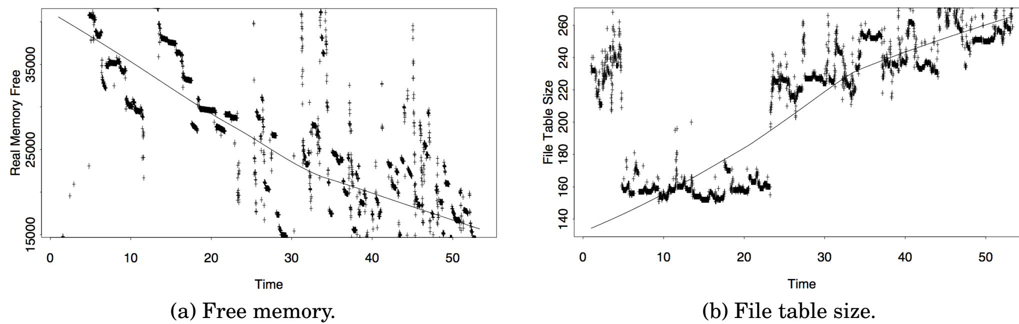
The paper is organized as follows. Section 2 summarizes basic definitions and concepts about Software Aging and Rejuvenation. In Section 3, we describe the procedure that we followed to collect SAR research papers. Section 4 reviews SAR literature with respect to different dimensions. Section 5 concludes the paper with a discussion about open issues and research opportunities.

2. BASIC CONCEPTS ON SOFTWARE AGING AND REJUVENATION

Before analyzing Software Aging and Rejuvenation studies, we briefly provide in this section some definitions and concepts that will be recalled in this survey. An in-depth discussion about fundamental concepts can be found in [Grottke et al. 2008]. In general terms, the Software Aging phenomenon consists in the *increase of failure rate* and/or *decrease of performance* of a long-running software system. In turn, it is due to the activation and propagation of the so-called **Aging-Related Bugs** (ARBs), a particular class of software faults that manifest their effect only after a long period of execution. The activation of these bugs does not immediately cause a failure of the system: considering the conceptual *fault-error-failure* chain proposed in [Avizienis et al. 2004], the peculiarity of ARBs is that their activation/propagation depends on *the* ***total time*** *the system has been running*, and/or that *they lead to* ***error accumulation****, which causes the system to gradually shift from a correct state to a failure-prone one*. After a long

enough execution time has elapsed, or after a significant amount of errors has been accumulated, ARBs lead to **aging-related failures** (e.g., an operation that allocates memory fails, causing the crash of a process). Error accumulation usually takes the form of bad resource management that leads to resource exhaustion, such as memory leaks, unterminated threads, and unreleased file locks: in such cases, the expected time to aging-related failure is referred to as **time to (resource) exhaustion** (TTE). Moreover, error accumulation is influenced by the amount and type of work performed by the system, which is referred to as **workload**.

Aging effects are the results of error accumulation, in terms of leaked resources or corrupted state; these effects can be detected by means of **aging indicators**, that is, system variables that can be directly measured and that can be related to Software Aging phenomena. Examples of aging indicators are data about resource usage of the operating system, such as free physical memory, used swap space, file and process tables size. In [Garg et al. 1998b], an experiment was performed by monitoring OS resource usage in a LAN of UNIX workstations using a distributed SMNP monitoring tool, in order to identify software aging trends. Fig. 1 shows the consumption of real memory and file tables across 53 days. Since aging phenomena may not be evident simply by visual inspection of data, trend detection techniques are usually adopted to detect the onset of software aging. In Fig. 1, a trend is detected by *smoothing* of observed data by *robust locally weighted regression*, which provide visual cues that there is a gradual decrease of free memory and and increase of file table size. To make conclusive statements regarding the presence or absence of software aging trends with statistical confidence, the use of statistical techniques is required: examples of techniques that were usually adopted are the *seasonal Kendall* and the *Mann-Kendall tests for trend*, for testing the hypothesis that there is an upward or a downward trend in the observed data, and the *Sen procedure* to estimates for the slope of a trend. Using the estimated slope, it is possible to compute the expected TTE of a given resource.



(a) Free memory. (b) File table size.

Fig. 1: Trend analysis of resource consumption in a UNIX system [Garg et al. 1998b] © IEEE.

Resource leaking and other aging effects can be due to aging-related bugs in application software, in the libraries that the application is using or in the application execution environment (e.g., the operating system). However, fixing these bugs is not always feasible, since bugs could be in third-party or reused code for which developers lack source code and/or expertise on their internals; moreover, it can be very difficult to identify these bugs in complex software applications. Software rejuvenation is a cost-effective approach to remove aging effects and avoid aging-related failures, which does not require to know the location of aging-related bugs, or even the very fact of their

existence. Software Rejuvenation was defined in [Huang et al. 1995] as the *preemptive rollback of continuously running applications to prevent failures in the future*. Since an application may be unavailable during rejuvenation, rejuvenation can increase the downtime and incur in some costs (e.g., costs due to the loss of business). However, these costs can be minimized by scheduling rejuvenation during the most idle times of an application. Instead, it is likely that the costs of downtime will be high if the downtime is unscheduled, as it happens during a failure. Rejuvenation can avoid, or at least postpone, aging-related failures, therefore it can reduce the overall downtime and related costs.

For this reason, the most important problem in the SAR field is to plan *when* to perform rejuvenation during execution (**rejuvenation schedule**) in order to improve availability and reduce costs. An example is represented by the simple model based on continuous-time Markov chains that was introduced in [Huang et al. 1995] to analyze software rejuvenation (Fig. 2). In this model, after starting the application stays in the "highly robust state" S_0, in which the probability of aging-related failures is negligible since the application has not aged. After some time (at a rate r_2 much lower than the others), the application will go into a failure-probable state S_P (e.g., some resources have been leaked and are close to be exhausted); in this state, the application can fail with rate λ and go to the failed state S_F, and will be repaired at a rate r_1. If the system performs rejuvenation, it will go from state S_P to S_R at rate r_4, and then to the robust state S_0 at rate r_3. This model allows to compute the expected downtime (both scheduled and unscheduled) and its costs; in turn, it allows to analyze whether, and under which conditions, software rejuvenation is beneficial to availability. For instance, if the cost of rejuvenation is small and the failure rate is large, rejuvenation should be performed as soon as the application goes into a failure probable state. However, several other models, which will be discussed in the following sections, have been proposed in the SAR literature.

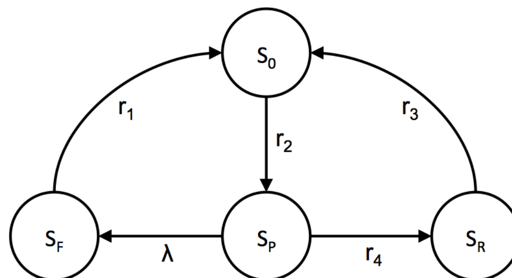


Fig. 2: Probabilistic state transition model for a system with rejuvenation.

A simple way to perform software rejuvenation is to restart the application. Restarting can involve queuing the incoming messages temporarily, cleaning up the in-memory data structures, respawning the processes at the initial state or at a previously checkpointed state. Moreover, rejuvenation can be performed at different levels, ranging from restarting an individual component to rebooting the whole node in which the application is running. The definition of **rejuvenation techniques** (i.e., *how* to perform rejuvenation), able to reduce the likelihood of aging failures and to keep low rejuvenation costs, is another important problem of SAR, that will be analyzed in the following of this paper.

3. ANALYSIS OF LITERATURE

To have a picture of the current status of SAR literature, we analyzed conference proceedings and journals in the area of software dependability and software engineering. We collected papers on SAR by following a two-step procedure:

(1) *Bootstrap*: We obtained an initial set of papers through a keyword-based search of papers appeared in a selected set of conference proceedings and journals;
(2) *Recursive closure*: We extended the set of papers by inspecting the reference list of each paper in the initial set and adding all referenced papers relevant to Software Aging and Rejuvenation to the set. Moreover, with the aid of search engines, we identified all papers referencing any paper in the initial set, and relevant to Software Aging and Rejuvenation, and we also added these papers to the set. These operations have been repeated for each newly added paper, until no more papers could be added to the set, thus obtaining a "recursive closure" of the initial set of papers.

To perform our analysis, we relied on the SciVerse Scopus (`http://www.scopus.com`) and the IEEE Xplore (`http://ieeexplore.ieee.org/`) search engines. These digital libraries index and provide access to scientific journals and proceedings from major computer science publishers. We considered publications from ACM, IEEE, Elsevier, Wiley, and Springer. During the bootstrap phase, we adopted the following criteria:

— **Search keywords**. To obtain the initial set of papers, a search has been carried out by querying for the words "aging", "rejuvenation", "restart", or "leak" in the metadata (title, abstract, and keywords) of publications. It is important to note that these words may not refer to the Software Aging phenomenon as intended in this paper, and that the results of the search were manually inspected in order to remove irrelevant results. For instance, in the case of software engineering studies, the word "aging" has also been adopted for indicating software obsolescence (e.g., because of changed requirements or maintenance actions) [Parnas 1994]. Other contexts in which the word "aging" has been adopted include the wear-out of hardware components, and computer systems for aiding elderly people. Similarly, the word "leak" has also been adopted in the context of computer security to refer to unauthorized information disclosure.
— **Selection of conferences and journals**. The search described above was focused on a set of relevant publication venues in the fields of software dependability and software engineering, since the number of papers returned by the search criteria mentioned above is very large. First, we identified a set of journals and transactions from the aforementioned publishers that are related to these fields, such as *IEEE Transactions on Software Engineering* and *Journal of Systems and Software*. We subsequently identified flagship conferences and symposia in these fields (and workshops held jointly with them), including the *IEEE/IFIP International Conference on Dependable Systems and Networks*, the *ACM/IEEE International Conference on Software Engineering*, and *ACM SIGMETRICS/Performance*. Other journals and proceedings were added by querying search engines for "software engineering" in the journal or proceedings title, and by analyzing publications from well-known SAR researchers (i.e., authors appearing most frequently in our preliminary analysis [Cotroneo et al. 2011a], such as K.S. Trivedi).

By querying search engines, we obtained a list of 1,021 publications. These publications were analyzed in order to discard irrelevant results: in most cases, the analysis of the abstract was enough to identify with confidence whether the publication was

actually related to Software Aging and Rejuvenation. This process produced a set of 204 publications.

Although the *bootstrap* phase allowed us to find a large number of SAR papers, it cannot assure that the sample of papers is comprehensive and unbiased (e.g., results could be biased due to the choice of keywords and of publication venues). Therefore, we extended the set of papers by recursively including papers referencing or referenced by the papers in the set (i.e., we checked references and citations of papers in the initial set of papers, as well as of each newly added paper), in order to obtain a *recursive closure*. Again, papers were manually inspected in order to make sure that they were actually related to the SAR field. This procedure allowed us to include 291 additional relevant studies, thus obtaining a set of 495 papers. These papers constitute a large bibliography about Software Aging and Rejuvenation, and it can provide a comprehensive picture of the state of the art in this field. These papers were further analyzed as discussed in the following sections.

It is important to note that there are some redundancies among the analyzed papers due to papers that are based on previous papers from the same authors (e.g., a journal paper based on a previous conference paper). However, in peer-review research venues, the degree of originality is one of the most important evaluation criteria, and extended versions of previously published papers should provide new material in order to be accepted for publication. Therefore, we assume that every paper has some original contents, even in the presence of redundancies. Although the extent of new material can be substantial in some cases and marginal in others, this is a subjective judgement and we do not have available a way to measure originality in a quantitative way (in fact, reviewers typically evaluate originality in a qualitative way, on the basis of their own knowledge and perception of the value of paper contributions). Therefore, we included in our analysis both extended papers and their preliminary versions. However, counting both types of papers in the analysis of the literature reflects the fact that more work than an individual paper has been devoted to a given problem, therefore this approach does not severely affect the analysis of the topics on which SAR research has been focused.

In Figure 3, the number of papers per year is reported. An increasing trend can be noticed. A sharp increase in the number of SAR-related papers was fostered by the 2008 WoSAR workshop, which took place for the first time in that year. A sharp decrease occurred in 2009 due to the absence of WoSAR in that year. The workshop took again place in 2010 and 2011, along with the publication of the Special Issue on Software Aging and Rejuvenation (based on WoSAR 2008 papers) on the *Journal of Systems and Software* in 2010. This is confirmed by the high number of papers discussed at WoSAR and JSS, as shown in Figure 4, which provides the number of publications for venues with at least 2 SAR papers. In the area of software dependability, the venues preferred for SAR studies were ISSRE, DSN (and satellite workshops), and PRDC. However, several studies also appeared in software engineering conferences, such as ICSE, PLDI, and several other conferences published in the LNCS series. The journals with the highest number of SAR studies are JSS, TR, TDSC, SPE, and PEVA.

4. DETAILED ANALYSIS OF SOFTWARE AGING AND REJUVENATION STUDIES

In order to provide a framework for analyzing the state-of-the-art in SAR research, we introduce four orthogonal *dimensions*, that we believe can provide a comprehensive overview of the literature and insights on future research directions that could be pursued by the Software Aging and Rejuvenation community. Each dimension consists of a set of *classes* that are used to classify the surveyed studies. The dimensions are:

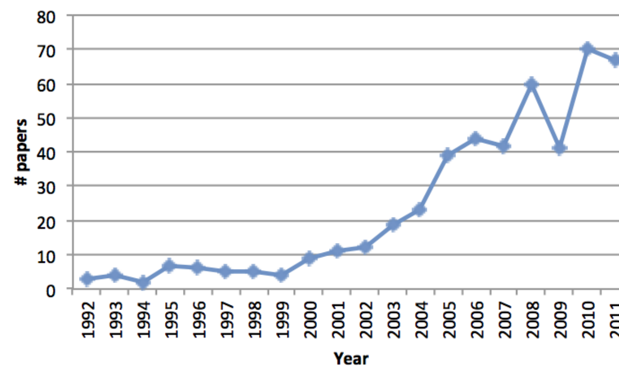


Fig. 3: Number of SAR publications per year.

— **Type of analysis**: This dimension reflects the type of analysis that is conducted in the studies. It includes *model-based*, *measurement-based*, and *hybrid* studies about strategies for planning software rejuvenation, studies that deal with the *detection* and the *avoidance* of aging-related bugs respectively during software verification and development, studies that propose novel *rejuvenation techniques*, and *field failure data* studies.
— **Type of system**: This dimension classifies the system in which the Software Aging phenomenon has been studied, in terms of *domain* and *criticality* of the system.
— **Aging effects and aging indicators**: This dimension classifies the studies with respect to the *system resources or performance indicators* in which effects of Software Aging were experienced, and which can be adopted for planning Software Rejuvenation.
— **Rejuvenation techniques**: This dimension describes the technique which is adopted or proposed in the study in order to clean-up the system state and reduce the likelihood of aging failures.

4.1. Type of analysis

This section deals with the type of analysis that researchers undertook in these years to study Software Aging and Software Rejuvenation.

Model-based studies: Software Aging has been analyzed starting from the empirical observation that several long-running systems are affected by transient failures and need, from time to time, to be rebooted. As early studies on telecommunication and transactional systems demonstrated [Gray 1985; Bernstein 1993], there is a high incidence of bugs that manifest themselves as memory leakage, broken pointers, unreleased file locks, and numerical error accumulation, making the system to slowly degrade its performance and to eventually fail. Therefore, practitioners adopted some form of software rejuvenation since a long time, in order to counteract aging effects [Bernstein and Kintala 2004]. This approach raised the problem of optimal scheduling of rejuvenation actions, in order to be cost-effective. [Huang et al. 1995], followed by others in the subsequent years [Garg et al. 1998a; 1995; Garg et al. 1996], attempted to model the phenomenon in order to provide an abstract view of it and a mathematical treatment. This allowed researchers to cope with software aging by formulating analytically the phenomenon, focusing the attention on how to optimally schedule rejuvenation actions.

Analytical models were first employed in order to prove that software rejuvenation can reduce the costs of system downtime [Huang et al. 1995] and minimize program

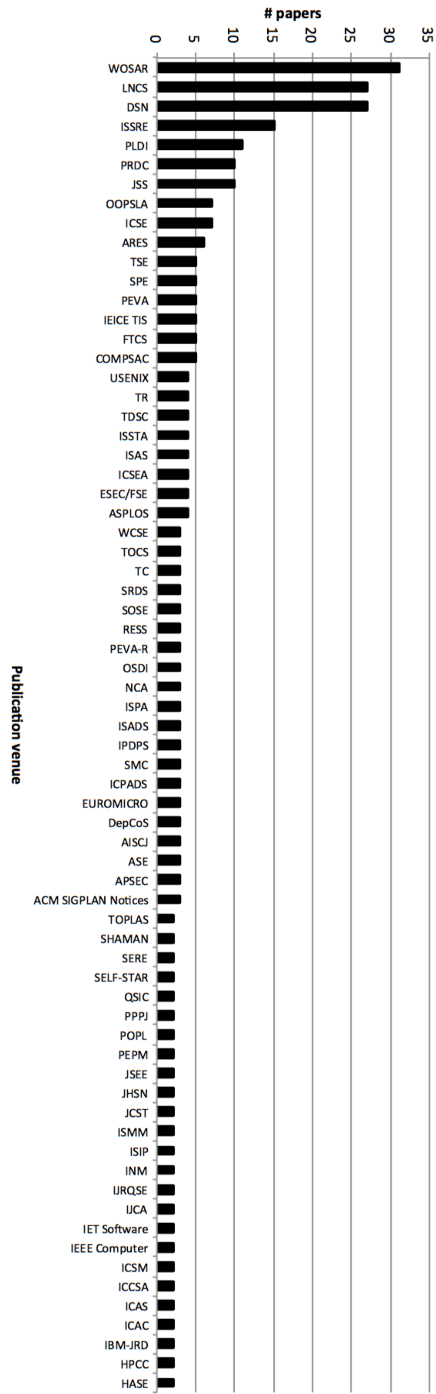


Fig. 4: Number of SAR publications in major journals and conference proceedings.

completion time [Garg et al. 1996] in the presence of software aging. Model-based studies can be distinguished by the type of stochastic process used to model the phenomenon. There are many type of models adopted, typically Markov-based. These include:

— *Markov processes and semi-Markov processes*: this type of processes with their variants are today the basis for model-based SAR analysis. The first work on software aging by Huang *et al.* modeled the phenomenon using a four-state Continuous-Time Markov Chain (CTMC), which still represents the basic model of the phenomenon (as discussed in Section 2).
This basic model has been extended in many ways, using Markov and Semi-Markov processes. Examples are [Garg et al. 1998a], in which a *non-homogeneous* CTMC is used (i.e., in which the sojourn time in each state is not exponentially distributed), and [Bao et al. 2005], where Semi-Markov Processes are used (i.e., in which the transition rates from one state to another depend, besides the current state, on the time spent in it) to model proactive fault management.
Markov chains are still widely used in their basic form, basically to: *i)* analyze more complex systems, possibly with multiple rejuvenation strategies (e.g., in [Xie et al. 2004] cluster systems behavior is described by CTMC), to *ii)* analyze aging in new contexts, such as systems using virtualization technologies (examples are given by the works in [Myint and Thein 2010] and [Kourai and Chiba 2007], where aging and rejuvenation is studied for virtualized servers), or to *iii)* describe more complex failure manifestation (instead of modeling one failure state causing the total service unavailability, various degrees of “failures” are considered for modeling a gradually decreasing service rate, i.e., a performance degradation; in [Du et al. 2009], [Pfening et al. 1996], [Koutras and Platis 2011], [Okamura and Dohi 2011], multiple degradation systems performance is described by a CTMC).
— *Other Markov-based models*: although most of studies adopt the classical Markov and semi-Markov processes, some papers explored different types of modeling. For instance, authors in [Garg et al. 1995; 1998a; Wang et al. 2007] used Markov Regenerative Processes (MRGP), in conjunction with Stochastic Petri Nets (SPN), in order to build a simple but general model to estimate the optimal rejuvenation schedule in a software system. Markov regenerative processes are a generalization of Markov and semi-Markov processes, which can capture the behavior of real systems with both deterministic and exponentially distributed event times. Thus, while the first models consider exponentially distributed rejuvenation rate from the *failure probable* state, authors using MRGP coped with deterministic rejuvenation interval, starting from the *robust state*. Authors show the effect of the such a deterministic rejuvenation interval on the steady state expected down time and cost.
The problem of finding the optimal rejuvenation schedule has been formulated also as a *Markov Decision Process* (MDP) where: the time is discretized and represents one dimension of the state description, the decisions at each state determines if the system should be rejuvenated or not, and the solution consists in finding the optimal policy so that a cost function is minimized. An example is represented by one of the first SAR works, in [Pfening et al. 1996], where authors adopted a Markov decision process to build a software rejuvenation model in a telecommunication system including the occurrence of buffer overflows. A more recent example is represented by the work [Okamura and Dohi 2011], in which Partially Observable Markov Decision Processes (POMDP) are also used to model the phenomenon. In their future work, authors plan to include in the formulation information coming from the system (i.e., “signs of aging”) in order to build online adaptive algorithms to control the software

rejuvenation, in which case the strategy would be an example of *hybrid* approach (i.e., both model- and measurements-based), described later in this section.

— *Petri Nets*: Petri Nets and their numerous variants are formalisms strictly related to Markov-based models, but that are more compact and in some cases easier to define. This formalism allows more easily to express performability metrics with multiple levels of performance, and is particularly useful to express metrics in more complex systems, such as systems with multiple nodes (e.g., clusters).
  For instance, Deterministic and Stochastic Petri Nets (DSPN) were employed in [Wang et al. 2007] in order to build a model to analyze performability of cluster systems under varying workload. Similarly, Stochastic Reward Nets (SRN) were used by [Vaidyanathan et al. 2001] to model clustered systems. A recent approach based on Petri nets has been presented by [Salfner and Wolter 2010], in which authors focused on the evaluation of time-triggered system rejuvenation policies using a queuing model, formulated as an extended stochastic Petri net. More recently, the work in [Andrade et al. 2011] combined Stochastic Reward Nets (SRN) with SysML to make it easier for system administrators to analyze rejuvenation in their server systems. Their goal is to adopt a semi-formal language, SysML, to describe the system configurations and maintenance operations, allowing people not having expertise in availability modeling to design and study the effects of different rejuvenation policies deployed in server systems.

Model-based studies analyze abstract models, making some simplifying assumptions about a system, such as assumptions about the underlying stochastic distributions that characterize the system. **These models can apply to a wide range of systems, therefore model-based approaches can provide more general findings and can be more portable across systems than measurement-based approaches**. However, **model-based software rejuvenation can be less effective than measurement-based approaches**, since it may not exploit some peculiarities of a specific system, and may not be able to adapt to conditions different than expected. In any case, model-based approaches rely on real data in order to populate model parameters, which could be obtained from measurement-based analysis.

Measurements-based studies: a considerable attention has also been devoted to the empirical analysis of Software Aging based on measurements from real systems. Since the phenomenon manifests itself as performance degradation and/or resources consumption [Huang and Kintala 1993; Huang et al. 1995], researchers focused on approaches for exploiting empirical measurements from the system, in order to identify whether the system is in a failure-prone state due to Software Aging, to forecast the time-to-aging-failure, and to plan Software Rejuvenation. Measurements-based studies also provide detailed information about aging phenomena in real systems, which is useful to better understand the nature and the extent of Software Aging and to raise the awareness on this issue.

The basic idea of measurement-based rejuvenation approaches is to directly monitor system variables, namely *aging indicators*, that can denote the onset of software aging, and predict the occurrence of aging failures by analyzing the collected runtime data statistically. This will provide hints about the best time to perform rejuvenation. Several approaches have been proposed to perform prediction, that can be grouped in:

— *Time series analysis*: a widely used approach is adopting time-series analysis on monitored resources. Time series are typically analyzed by first using trend tests to accept/reject the hypothesis of no trend in data (e.g., Mann-Kendall, t-student, Seasonal Kendall tests), and using techniques to estimate such trend and possible seasonality in data (e.g, multiple linear regression, regression smoothing, Sen's slope estimate procedure, autoregressive models).

One of the first measurement-based analyses is reported by [Garg et al. 1998b]: in that paper, a set of 9 Unix Workstations was monitored for 53 days using an SNMP-based monitoring tool. During the observation period, 33% of reported outages were due to resource exhaustion, highlighting that Software Aging is a non-negligible source of failures in computer systems. In [Li et al. 2002; Grottke et al. 2006], the authors analyzed performance degradation also in the Apache Web Server by sampling web server's response time to predefined HTTP requests at fixed intervals. Collected data were analyzed using similar techniques adopted in [Garg et al. 1998b]. Seasonal patterns analysis have been also considered in [Grottke et al. 2006], in which trends are analyzed also in presence of seasonal variation in data. Time-series ARMA/ARX models have been also used by [Li et al. 2002] on the Web Server Apache, in order to estimate the resource exhaustion. Compared with the linear regression and extended linear regression models, ARX model incurs higher initial overhead, but once it is established, it can be used for prediction for a long period without reestimating the parameters in the model.
Time-series analysis has been also adopted to study the relationship of the Software Aging phenomenon with workload in complex systems, including the Linux Kernel code [Cotroneo et al. 2010], and the Java Virtual Machine [Cotroneo et al. 2011b]. In both cases, the analysis of workload parameters is used to provide indications on potential sources of Software Aging, by highlighting the subsystems whose parameters are correlated to the experienced aging trends. Principal Component Analysis (PCA) followed by multiple linear regression are adopted in such works, in order to remove first-order correlation among predictors and then to provide linear estimates of aging trends, by regression, reducing the problem of multicollinearity.
ARIMA (Autoregressive Integrated Moving Average) and Holt-Winters (Triple Exponential Smoothing) models have been used in [Magalhaes and Silva 2010], where authors developed a framework for detection of performance anomalies caused by aging, which is targeted to web and component-based applications. In particular, the framework monitors application/system parameters, used to determine the correlation between the application response time and the input workload, in turn used to train machine learning algorithms. At run-time, parameters collected by monitoring are estimated ARMA and Holt-Winters algorithms, and the estimations classified by the trained ML algorithms to determine if the application may incur in some performance anomaly.
Four different time-series models have been used in [Araujo et al. 2011b] in order to schedule software rejuvenation properly. Used models are: the linear model, the quadratic model, the exponential growth model and the model of the Pearl-Reed logistic. They have been adopted for predicting memory consumption trends on the Eucalyptus cloud computing framework.
One more paper considering non-linear models is in [Jia et al. 2008], in which aging is studied in Apache by constructing a dynamic model to describe the software aging process following the method of nonlinear dynamic inversion. Software aging process is shown to be nonlinear and chaotic.
In the best practice guide by Hoffmann et *al.* [Hoffmann et al. 2007], multivariate non-linear models (support vector machines, radial, and universal basis functions) have been compared with multivariate linear models. The former ones have shown better performance than linear models in the benchmarking case studies.

— *Machine learning*: Machine learning approaches are a more sophisticated form of data analysis, which adopt algorithms from the field of Artificial Intelligence (e.g., classifiers and regressors) to identify trends and classify a system state as robust or failure-prone. A work in this regard appeared in [Cassidy et al. 2002], in which authors adopt pattern recognition methods to predict Software Aging phenomena in

shared memory pool latch contention in large OLTP servers. The approach applies non-linear, non-parametric regression to a large set of system variables, and analyze the residual error between the predicted and the actual system values using a sequential probability ratio test, in order to predict the onset of Software Aging effects. Results showed that these methods allowed to detect significant deviations from "standard" behavior with a 2 hours early warning. Another example of the application of machine learning approach to predict software aging failures is explored in [Alonso et al. 2011a] in the context of a three-tier J2EE system; in that work, Alonso *et al.* propose a machine learning approach to build automatically *regression trees* models that relate several system variables (e.g., number of connections and throughput) to aging trends, based on the observation that Software Aging trends can be approximated using a piecewise linear model. The models were trained using data samples collected in preliminary experiments, and were used to predict the TTE of system resources under conditions different than the ones observed during the training phase. Three different machine learning algorithms (namely, naive Bayes classifier, decision trees and a neural network model) have been also used, in combination with time-series models, in the previously mentioned work [Magalhaes and Silva 2010], in order to predict aging in web applications.

— *Threshold-based approaches:* differently from the previous ones, this kind of approaches define thresholds for some aging indicators, and the rejuvenation is triggered when the monitored indicators exceed such thresholds. For instance, indicators may refer to resource consumption. Difficulties arise in identifying the best indicators and the right thresholds for them, that are able to prevent actual failures and useless rejuvenation actions at the same time. An examples of this approach is in the work [Silva et al. 2009], which adopts thresholds on mean response time and on quality of service indicators. Authors propose a rejuvenation approach based on self-healing techniques, that exploits virtualization to optimize recovery. They implemented a rejuvenation framework, called *VM-Rejuv*, in which an *Aging Detector* module detecting aging conditions based on the mentioned thresholds.
The work in [Silva et al. 2006] is a further example of threshold-based approach; the paper presents an analysis of software aging in a SOAP-based server, in which a dependability benchmarking study is conducted to evaluate some SOAP-RPC implementations, focusing in particular on Apache Axis, where they revealed the presence of aging by parameters monitoring.
In [Matias Jr. 2006], authors presents an evaluation of aging effects in Apace Web server, based on a controlled experiment. In that work, the memory consumed by Apache was observed together with three controllable workload parameters: page size, page type (dynamic or static) and request rate; authors adopted thresholds on the usage of virtual memory as aging indication. The work in [Araujo et al. 2011b] combines threshold-based approach (wit ha threshold on memory utilization) with time-series analysis. It implements a rejuvenation policy in the Eucalyptus cloud computing infrastructure, by using multiple thresholds and forecasting by time series analysis models. Time-series models adopted are: linear model, quadratic model, growth curve model, and S-curve trend model. Thresholds on resources are also adopted in [Avritzer and Weyuker 2004], where threads and memory are monitored and actions a re taken when they exceed some thresholds (e.g., garbage collection).

Measurement-based studies forecast software aging based on direct measurements (e.g., based on time series), and provide empirical data about software aging phenomena. The advantage of this kind of approach is that **software aging forecasting can adapt to the current condition of the system** (e.g., the current operational

profile, which may not have been foreseen before operation), and can **accurately predict the occurrence of aging phenomena**. However, this kind of approach **may not be easily generalizable**, since they exploit some peculiar aspect related to the nature of the considered system (e.g., the fact that some particular resource exhibits seasonal or fractal patterns [Garg et al. 1998b; Shereshevsky et al. 2003]). Moreover, measurement-based approaches are not meant to estimate long-term dependability measures such as availability.

Hybrid studies: remarkable attempts have been made to **combine the benefits of both model-based and measurement-based approaches**, by describing the phenomenon analytically, most often by Markov-based models, and determining the model's parameters through measurement, i.e., via observed data. These works are here referred to as *hybrid*, in that they combine aspects of the previous two approaches. **Despite the practical importance of hybrid approaches, only a minority of studies has been made to exploit measures for feeding models**.

Interesting examples are represented by the papers [Vaidyanathan and Trivedi 1999] and [Vaidyanathan and Trivedi 2005]. They presented results of an analysis conducted on the same set of Unix workstation used in [Garg et al. 1998b]. While the latter considered only time-based trend detection and estimation of resource exhaustion without considering the workload, these papers took the system workload into account and built a model to estimate resource exhaustion times. They considered some parameters to include the workload in the analysis, such as the number of CPU context switches and the number of system call invocations. Different workload states were first identified through statistical cluster analysis and a state-space model was built, determining sojourn time distributions; then, a reward function, based on the resource exhaustion rate for each workload state, was defined for the model. By solving the model, authors obtained resource depletion trends and TTE for each considered resource in each workload state. The methodology allows carrying out a workload-driven characterization of aging phenomena. The second one of these two works [Vaidyanathan and Trivedi 2005] is a clearer example of a hybrid approach, in which: *i)* a measurement-based semi-markovian model for system workload is built, *ii)* TTE for each considered resource and state (using reward functions) is estimated, *iii)* and finally a semi-Markov availability model is provided, based on field data rather than on assumptions about system behavior.

A further noticeable attempt is the work carried out by Eto, Dohi and Ma [Eto et al. 2008], who used reinforcement learning to estimate the optimal rejuvenation schedule adaptively, i.e., by considering runtime data to update the estimation. Thus, their estimation technique does not require the complete knowledge on system failure (degradation) time distribution in the operational phase, even if the underlying state transition of software is governed by models, i.e., by CTMCs or SMPs.

In all the three classes of studies, one more factor to be considered is the inclusion of workload dependency in the analysis. Since aging has been shown to be clearly correlated with workload variation, several authors accounted for its impact. Thus, many authors, such as [Andrzejak and Silva 2007; Vaidyanathan and Trivedi 2005; Bao et al. 2005; Garg et al. 1998a; Matias Jr. 2006; Bovenzi et al. 2011] considered the workload in their analyses, too, and vary the estimate of the time-to-aging-failure at a given time as a function of the workload actually experienced by the system. Several different approaches have been taken for this purpose: for instance, in the discussed study [Vaidyanathan and Trivedi 2005] authors modeled the workload by a semi-Markov process, whereas in [Matias Jr. 2006; Carrozza et al. 2010; Bovenzi et al. 2011] the design of experiment technique is used to plan experiments with varying workload in a pure measurement-based approach.

Studies on rejuvenation techniques: one important area that is being investigated by researchers is about the different techniques of rejuvenation; these works do not focus on rejuvenation scheduling only, but face the problem of the efficiency of the rejuvenation action in terms of cost or downtime, by proposing or comparing several rejuvenation techniques. While model-based and measurement-based studies aim to determine the best time in which to execute rejuvenation by assuming that some software rejuvenation technique will be adopted, the aim of studies on rejuvenation techniques is to define and analyze rejuvenation approaches from a technical point of view (e.g., how to adopt virtualization not to interfere with the running service [Silva et al. 2009], or the definition of rejuvenation scheduling models for HPC systems via OS restart based on TTF or on Reliability data [Naksinehaboon et al. 2010]). We refer to this category as *Rejuvenation Techniques*, which cannot fall in none of the previous category. Section 4.4 reports more in detail on the different types of rejuvenation methods proposed.

Studies on avoidance, verification, and debugging of software affected by aging-related bugs: a parallel track should be taken into account, regarding many works that faced problems connected to Software Aging, but from a different perspective than the SAR research community. This track is considerably growing in the area of software engineering, and addresses problems like verification, testing, debugging, and fault avoidance, applied specifically to aging-related bugs. Examples are works on static and dynamic code analysis for aging-related fault detection such as memory leaks [Xu et al. 2011; Xu and Rountev 2008a; Weimer 2006; Heine and Lam 2006]. These works are here referred to as *Verification/Testing/Debug/Avoidance* studies.

Finally, some few works provide insights into aging-related bugs and failures from real field data (i.e., not obtained from controlled experiments); these are reported as **Field Failure Data** papers. An example of these works is the one by Grottke et *al.* [Grottke et al. 2010] that analyzes the reports from NASA/JPL space missions, and classifies bugs as *Bohrbugs*, *Mandelbugs* and *Aging-Related Bugs*. In that case, there is no aging analysis as conducted typically (i.e., analysis on the *effect* of aging), but the analysis is on the source of aging, i.e., aging-related bugs.

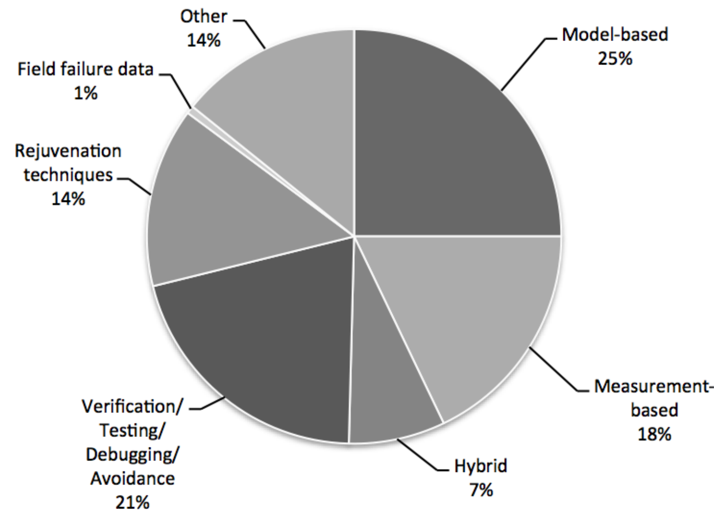


Fig. 5: Type of analysis.

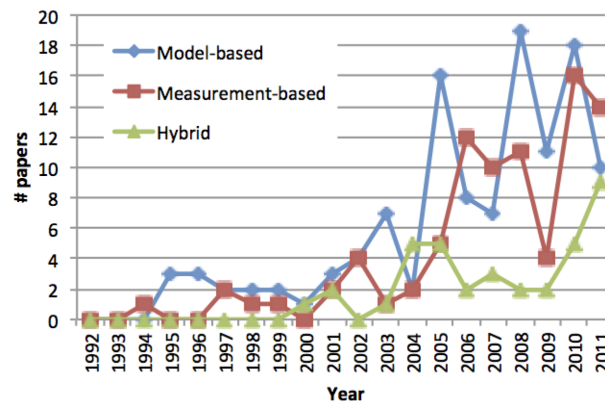


Fig. 6: Type of analysis per year.

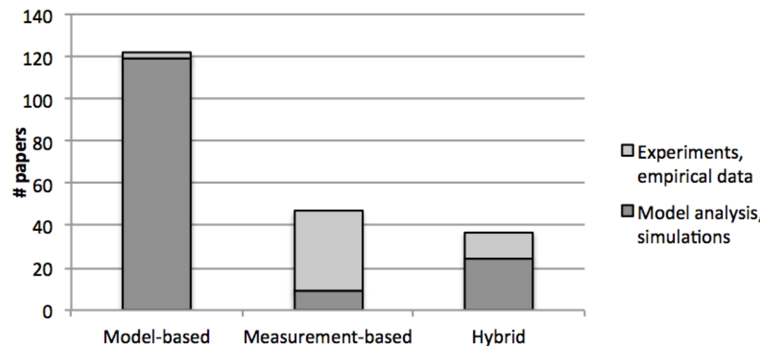


Fig. 7: Type of validation in mode-based, measurement-based and hybrid studies.

Figures 5, 6, and 7 synthesize the share of each class of works. Figure 5 shows the distribution of the type of analysis. Model-based, measurement-based, and hybrid classes cover most of the papers (50.10%, 248 papers). A significant slice of the SAR literature is represented by the “Verification/Testing/Debugging/Avoidance” class (21%), meaning that several other communities deal with topics related to Software Aging. This also highlights **the need for increasing the connection and cooperation among research groups coming from the dependability and the software engineering fields, in order to get to a more uniform treatment of the problem**.

Figure 6 focuses the attention on papers coming from the main track of Software Aging and Rejuvenation community, i.e., model-based, measurement-based and hybrid papers. The figure depicts the evolution over time of the three classes. For all the classes, the number of papers is increasing in the last years. In several cases, model-based papers result the largest share. The reason for this predominant trend may be that models can be used to **analyze many different hypothetical situations** (e.g., different types of systems, different rejuvenation strategies) **without having to obtain experimental data**, which allows to analyze several facets of the phenomenon by tuning or varying the analyzed model. It can also be noted that the high number of

model-based and measurement-based studies in 2008 and 2010 reflect the appearance of WoSAR and its Special Issue, as mentioned in Section 3.

Measurement-based approaches, which could be more expensive to apply, were also performed in many cases, in order to **provide evidences of software aging on a specific system**. For sure both kind of papers are needed to study the phenomenon in a wide perspective. However, hybrid approaches represent a small part of SAR studies (7%), even though it seems to increase. **We believe that hybrid studies can play an important role and deserve more attention**, since they explore how to mix the best of both model- and measurement-based approaches in order to achieve efficient and adaptive rejuvenation schedules, and to be generically applicable at the same time. Indeed, hybrid approaches can provide strategies and examples on how measurements, which are typically collected by modern systems for monitoring and debugging purposes, can be exploited to mitigate software aging and improve availability and performability. Furthermore, the application of software rejuvenation schedules to real systems could serve to provide evidence that model-based approaches are effective at improving availability and performability. A remarkable example of cross-check between models and real failures can be found in [Matias et al. 2010b], where the actual time-to-failure of the Apache Web Server is compared with the prediction obtained from Accelerated Life Tests.

Finally, Figure 7 compares the mentioned approaches with respect to the type of validation that authors proposed in their work. The plot only includes publications that are model-based, measurement-based, or hybrid studies since we are considering how software rejuvenation scheduling is validated in these three types of studies. Most of model-based works are validated through simulation and/or numerical analyses, as well as measurement-based approach are mostly validated by experiments on real data. It is interesting to note, that some (very few) works among the model-based ones are validated through empirical data. For instance, the work in [Zhao et al. 2010] proposes a BP (Error Back Propagation Network) model validated through actual data from the Apache web server. Similarly, few works of the measurement-based class are validated by model-based analyses. An example is in [Kim et al. 2007], in which authors evaluate the survivability of sensor nodes in a sensor network under Denial of Service (DoS) attacks, and validate their approach through simulation.

4.2. Type of system

Software aging has been shown to affect many kinds of long-running software systems, ranging from business-oriented to highly critical systems. It is important to figure out on what class of systems researchers mainly focused their attention when studying the aging phenomenon. Different domains also mean different development practices, cycles, techniques, and methodologies, which may affect the probability of the final system of being affected by aging.

In this Section, we first roughly distinguish papers in three classes, according to the scenario in which the system is adopted: **safety-critical systems**, **non-safety-critical systems**, and **unspecified**. Then, a more detailed analysis on subclasses of systems is presented.

The first class indicates systems employed in scenarios that are critical from the safety point of view, i.e., systems whose malfunctioning may cause serious damages or loss of human lives, such as military systems or space systems. The non-safety-critical systems class includes business and mission-critical applications, but not safety-critical ones, such as Web Servers or DBMSs typically employed in business applications. The class *unspecified* refers to papers that do not present an experimentation on real systems, but that use simulations or numerical examples to demonstrate the validity of their results.

Figure 8 shows the proportion of the three classes. Papers on non-safety-critical systems are the greatest part of the literature, while safety-critical systems are discussed in a minority of cases. This can be explained in part by the fact that the latter are fewer and are much better designed and tested. Figure 9, which shows the time evolution of the two considered classes, confirms that non-safety-critical systems have always been the greatest slice, and that the distance between the number of papers on non-critical vs. critical systems seems to increase with time.

However, from the figure it is important to note that, although safety-critical systems are designed to respect stringent dependability requirements and are tested extensively, **a non-negligible percentage of papers have considered aging phenomena in safety-critical systems**. On one hand, this confirms that aging problems are very difficult to detect during the development phase: the percentage suggests that even though safety-critical systems are usually developed through well-defined development practices and undergo to extensive testing activities, aging can manifest itself during operation. In fact, being a subclass of Mandelbugs, aging-related bugs are hard to reproduce—even when activated, their manifestation takes long time to become evident, and this makes the testing time insufficient to reveal the problem in most of cases. On the other hand, this suggests that **software aging should be systematically taken into account by developers of critical systems**, both in early phases of the lifecycle and at operational time.

It should be also noted that many papers do not perform experiments on real software systems (the *unspecified* class). The greatest part of these papers present model-based approaches for time-based rejuvenation, and validate their approach by numerical examples, as pointed out in the previous section.

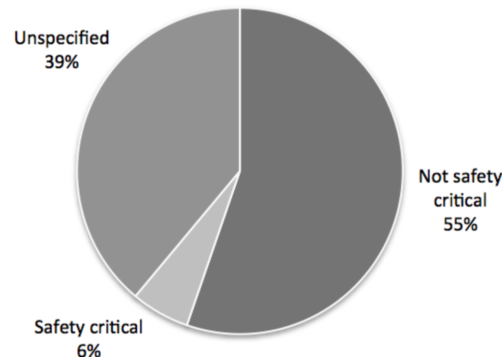


Fig. 8: Type of system.

Figure 10 reports a more detailed analysis of system categories analyzed in the SAR literature. It should be noted that the sum can be different than the number of surveyed papers, since in some cases a paper may belong to more than one category (e.g., papers that analyze more than one system). The first evidence is that **a quite large variety of systems has been shown to suffer from aging**. It includes, among others, web applications, web servers, OSs, DMBSs, cloud computing and virtualization environments, middleware, and server applications. Among these, web applications and web servers represent a relevant slice. Indeed, this type of applications is widely spread in the market and are used by large communities in the

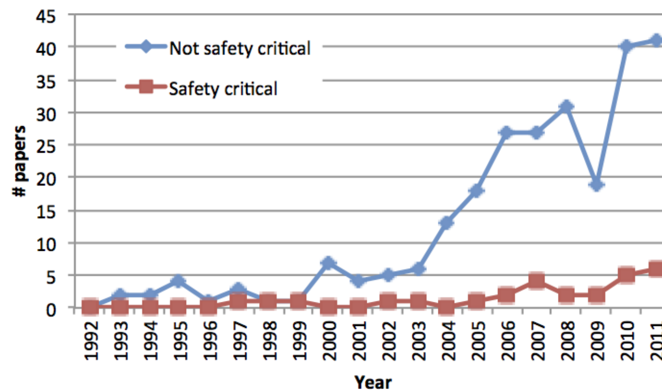


Fig. 9: Type of system in past studies per year.

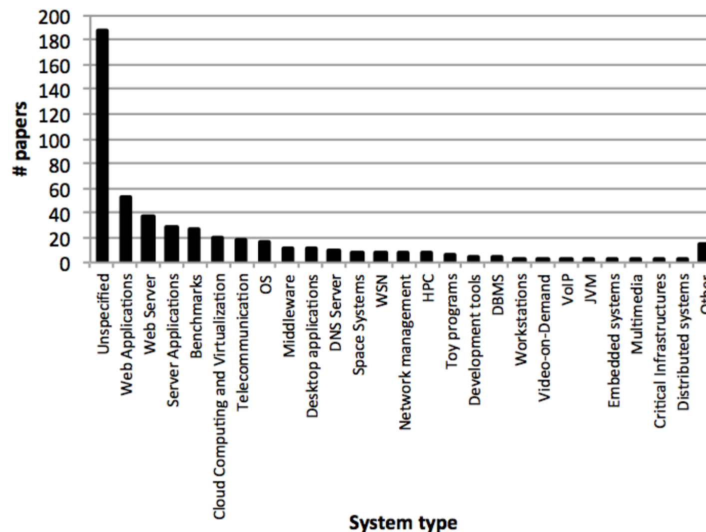


Fig. 10: System categories considered in past work.

Internet: this partially justifies their attractiveness as case-study for researchers. Examples of web applications used for these studies are web services and service-oriented applications (e.g., [Silva et al. 2009],[Andrzejak 2008], [Wolter and Reinecke 2010]), application servers (e.g.,[Ning et al. 2007; Silva et al. 2006]), e-commerce applications (e.g., [Alonso et al. 2011b; Avritzer and Weyuker 2004; Alonso et al. 2010; Magalhaes and Silva 2010]). Some commonly mentioned technologies in these studies are J2EE as application server, Tomcat/AXIS for web services support, and the TPC-W benchmark especially for simulating e-commerce applications. There is also a study on aging effects in client-side web applications (e.g., [Kiciman and Livshits 2007]).

The *web server* class, which is tightly related to the previous class, is also a relevant one. One of the most noticeable examples and the most studied application is the Apache Web Server. Several measurements-based analyses have been carried out on this system, especially by researchers closer to the dependability and fault toler-

ance rather than to software engineering community. Such analyses better characterized the phenomenon using this server as case study. Some of the most well-known measurements-based studies adopted Apache as case study: for instance, the work in [Li et al. 2002; Matias Jr. 2006; Grottke et al. 2006; Bovenzi et al. 2011] collect measures from Apache execution regarding resources usage, such as memory consumption, swap space, cache usage, and response times, and apply statistical techniques to predict aging. Apache data analysis also allowed to further develop solutions for coping with seasonality patterns in aging data [Grottke et al. 2006].

Server applications also account for a significant share; some relevant examples of such works are in [Castelli et al. 2001], where the impact of different (time- and prediction-based) rejuvenation policies on availability has been evaluated by means of analytical models based on stochastic reward nets (SRNs), in the *IBM Director for xSeries*® servers; in [Okamura et al. 2003], in which authors derive analytically the optimal software rejuvenation policies, under some system dependability measures, in a Transaction-Based Multi-Server System, also comparing single-server with multi-server configuration; in [Oppenheimer et al. 2003] failure causes using data from three large-scale Internet services are first analyzed, such as operator errors, configuration errors, failures in custom-written front-end software, and then various preventive maintenance techniques are compared to each other, finding that improvement in the maintenance tools and systems used by service operations staff would decrease time to diagnose and repair problems. Benchmark category is also relevant; the most of papers in this category is however related to static/dynamic analysis for memory leak detection, which adopt benchmark suites to test their result (e.g., [Novark et al. 2009; Jung and Yi 2008]).

It is interesting to note that the third class of systems is represented by *cloud computing*, which a relatively young research area. They have appeared mainly after 2007, after the widespread adoption of this kind of systems. In such studies, researchers often analyze several rejuvenation policies based on virtual machine and/or virtual machine monitor reboot/rejuvenation. Such strategies are then evaluated both by model-based approaches (e.g., [Machida et al. 2011]), and by measurements (e.g., [Kourai and Chiba 2007; Araujo et al. 2011b]). The availability of cloud computing software to experiment such strategies without excessive costs is favoring aging analysis on cloud. A relevant example is represented by the studies on Eucalyptus cloud-computing framework ([Araujo et al. 2011a; Matos et al. 2011; Araujo et al. 2011b]). Aging papers in this area are in a growing trend, since most of them have been published in 2011.

Telecommunication systems are among the first class of systems considered for aging analysis. Early examples of software aging and rejuvenation on communication systems come from AT&T labs, such as the ones provided in [Huang et al. 1995]. In those years, other studies on software aging considered telecommunications systems in their experiments: in [Huang et al. 1996] the fault tolerance capabilities of several communication products and services enriched with rejuvenation components is studied; in [Balakrishnan et al. 1997] software aging is analyzed in telecommunications billing applications, as well as in the related switching software. In subsequent years, some other papers dealt with telecommunication applications, such as the papers in [Liu et al. 2002; Liu et al. 2005] where software rejuvenation is proposed as a proactive system maintenance technique deployed in a CMTS (Cable Modem Termination System) cluster system, and the work in [Okamura et al. 2005], in which authors evaluate dependability performance of a communication network system with the software rejuvenation. Approximately half of the found work on telecommunication systems are published before 2000.

Some interesting aspects in the remaining classes are worth to mention: in the *OS* class, the greatest slice is represented by works on Unix and Linux systems (e.g., [Vaidyanathan and Trivedi 1999; Yoshimura et al. 2011; Cotroneo et al. 2010]); but it is worth to note also some papers on other OSs, such as Solaris [Ni et al. 2008], Windows NT [Robb 2000], and Android [Park and Choi 2012]. Works in the *DNS server* category are mainly aimed at preventing security-related failures by rejuvenation, such as [Huang et al. 2006] (see also 4.3), and at detecting security vulnerabilities causing aging effects [Antunes 2008]. It is finally worth to point out the *Space* system category, which is one of the most relevant and studied type of critical systems: examples are in [Tai et al. 1999], which describes the X2000 computing system for NASA's 15-year long Pluto-Kuiper Express mission, and the empirical study on bugs in space missions [Grottke et al. 2010].

4.3. Aging effects and aging indicators

The analysis of *aging effects* (i.e., the kind of erroneous states caused by aging) and *aging indicators* in this section shows how Software Aging has been manifesting itself in complex software systems. Aging indicators are an important area of study, since they are instrumental for detecting when the system state is prone to aging-failures, by monitoring them during the system execution. Aging indicators can be *indicators of resource usage* and *performance indicators*. The following classes of aging indicators were identified among the surveyed studies:

Memory consumption: Empirical evidence showed that free memory exhibits the shortest Time to Exhaustion (TTE) among system resources [Garg et al. 1998b], and that memory management defects are a significant cause of failures [Sullivan and Chillarege 1991]. Therefore, many studies on Software Aging and Rejuvenation analyze Software Aging phenomena affecting free memory, by measuring the amount of *free physical memory* and *swap space* [Grottke et al. 2006; Vaidyanathan and Trivedi 1999], and several measurement-based approaches apply time series and statistical models to these variables.

Performance degradation: SAR studies have often reported performance degradation in software systems affected by Software Aging. A cause of performance degradation is the depletion of system resources: for instance, the consumption of physical memory increases the time required by memory allocation procedures and garbage collection mechanisms, since their computational complexity is a function of the amount of memory areas that have been allocated [Carrozza et al. 2010; Ferreira et al. 2011; Cotroneo et al. 2011b]. An increasing request response time and a decreasing throughput have been reported for web applications [Silva et al. 2006], web servers [Grottke et al. 2006; Matias Jr. 2006], and CORBA-based applications [Carrozza et al. 2010]. In the presence of this kind of phenomena, software rejuvenation can be triggered when the quality of service (e.g., in terms of response time or throughput) is below a given threshold.

Other resource consumption: Software Aging can impact on several kind of resources. Besides memory-related resources (e.g., physical memory, virtual memory, swap space, cache memory), analyzed papers deal with the these type of resources:

— filesystem-related resources, such as stream descriptors and file handles [Weimer 2006; Zhang et al. 2011; Garg et al. 1998b];
— storage, whose space may be consumed by bad management [Bobbio et al. 2001];
— network-related resources, such as socket descriptors [Weimer 2006];
— concurrency-related resources, such as locks, threads and processes [Zhang et al. 2011; Garg et al. 1998b];

— application-specific resources, such as DBMS shared pool latches [Cassidy et al. 2002] and OSGi references [Gama and Donsez 2008].

It should be noted that in several papers, the approach being proposed is not constrained to a specific resource, but is focused on detecting incorrect API usage and incorrect exception handlers that may cause a resource leakage. For instance, the work in [Zhang et al. 2011] presents an approach that dynamically mines resource usage patterns by monitoring API calls, and provides an experimental evaluation on open source programs based on the Java I/O and concurrency APIs. A frequent kind of resource leak in Java programs is represented by sockets and file handles, due to faulty exception handlers that do not release these resources [Weimer 2006; Zhang et al. 2011]. Other resources can also be affected by Software Aging depending on the kind of system, such as free disk space in DBMS systems [Bobbio et al. 2001]. Some papers analyze a wider set of resources. In [Garg et al. 1998b], a network of UNIX workstations was monitored in order to identify aging trends in the consumption of several resources (related to virtual memory, the OS kernel, the filesystem, the disk, and the network), and a statistically significant aging trend was noticed in the process table size and in the file table size (although their TTE is lower than the TTE of free memory).

In addition to the aging effects mentioned above, **there exists other kind of aging effects that have been studied only in recent works**. A field in which software rejuvenation has been recently studied is related to **security attacks**, that is, attempts of malicious users to access unauthorized resources or to make the system unavailable. In fact, security attacks may take place and gradually compromise a system over a long period of time (e.g., password theft through bruteforce guessing, or flood attacks that trigger software aging phenomena), which can be mitigated by periodically rejuvenating a system, such as by changing cryptographic keys, by restarting compromised processes, and by randomizing the location of data and instructions in memory [Sousa et al. 2010; Tai et al. 2005; Valdes et al. 2003; Cox et al. 2006; Huang et al. 2006; Roeder and Schneider 2010]. A challenge in deploying software rejuvenation for security purposes is to define precise aging indicators that can be related to security attacks. Currently, the aging rate has to be assumed at design time [Sousa et al. 2010; Nguyen and Sood 2009] or should be based on imperfect attack/intrusion detectors that could raise false alarms and miss attacks [Aung et al. 2005; Nagarajan and Sood 2010].

Another kind of aging effects that have been discussed in a few recent works, which we refer to as **other aging effects**, are related to the *accumulation of numerical errors* [Grottke et al. 2008] and *memory fragmentation* [Grottke et al. 2008; Macedo et al. 2010]. This kind of aging effects are not necessarily caused by bugs in the software, but are related to the nature of floating-point arithmetic and memory allocation algorithms, respectively. In the case of numerical errors, **we did not found in the literature aging indicators able to estimate the extent of such errors in the system state**.

Finally, it should be noted that many studies propose models and approaches for dealing with Software Aging regardless of which specific kind of resource depletion or aging effect is experienced, which is usually the case of model-based studies. We denote these papers as **unspecified aging effects**.

Fig. 11 shows the number of papers related to each class of aging indicators. Most of past studies focused on software aging effects are related to *memory consumption* [Garg et al. 1998b; Matias et al. 2010a; Shereshevsky et al. 2003], *performance degradation* [Magalhaes and Silva 2010; Zhao and Trivedi 2011] or both [Grottke et al. 2006; Silva et al. 2006; Cotroneo et al. 2007; Carrozza et al. 2010; Silva et al. 2009; Matias Jr.

2006]. These two aspects are the most frequent issues occurring in non-safety-critical systems (see section 4.2), and they are considered by an increasing number of SAR studies (Fig. 12). These issues have less relevance for safety-critical systems. For instance, in the case of software that undergoes a safety certification process, dynamic memory management is typically avoided in order to accomplish the most stringent safety integrity levels. By contrast, **none of analyzed papers tackled arithmetic issues, such as the accumulation of round-off errors**. These errors are much more relevant in safety-critical contexts, given the fact that software is responsible for controlling physical actuators and an erroneous output may have severe consequences. A well-known example of aging failure related to numerical errors occurred in the Patriot missile system, which was caused by a round-off error in the conversion of the total execution time from an integer to a floating-point number [Grottke et al. 2008].

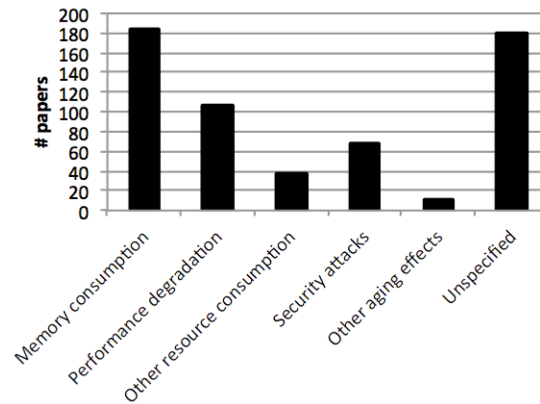


Fig. 11: Aging effects considered in past work.

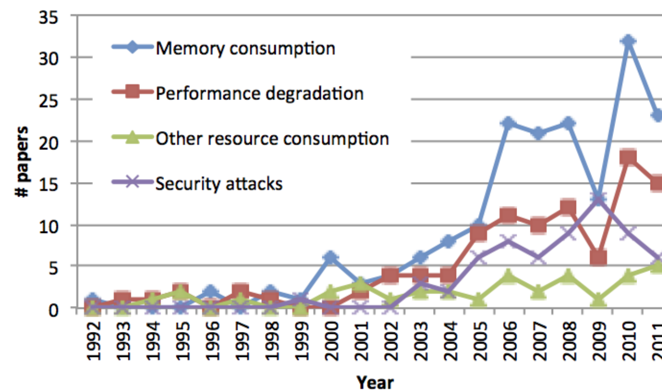


Fig. 12: Aging effects in past studies per year.

4.4. Software rejuvenation techniques

The fourth dimension against which we evaluated SAR studies is with respect to the software rejuvenation techniques that were proposed or adopted to counteract software aging. As discussed in Section 4.1, most of the SAR papers are focused on determining the optimal schedule to perform rejuvenation, by either analytical models (i.e., time-based rejuvenation), or by measurements (i.e., inspection-based rejuvenation). In this section, the attention is focused on techniques adopted to rejuvenate a system. Rejuvenation aims to bring the software from a *failure-prone* state (e.g., errors have been accumulated due to resource leakage) to an *aging-free* state. Therefore, rejuvenation techniques can be compared with respect to how the state is processed and to the resulting *aging-free* state that is achieved after rejuvenation.

When reviewing SAR studies, we recognized two broad classes of software rejuvenation techniques, respectively *application-specific* actions, i.e., techniques that take advantage of special feature of the application domain or architecture, and *application-generic* actions, i.e., techniques that restart the system or its parts and that are not specific to a particular class of systems. Whenever a study did neither propose nor suggest rejuvenation techniques, it was classified as *unspecified*. We identified the following categories among application-generic techniques, discussed in the following paragraphs: *Application Restart*, *OS Reboot*, *Virtual Machine Monitor (VMM) and Virtual Machine (VM) Restart*, and *Cluster Failover*.

Application Restart: The whole application is restarted (e.g., all processes of the application are shut down and restarted). This is the simplest form of software rejuvenation [Huang et al. 1995; Huang et al. 1998]. It relies on state initialization mechanisms available at the OS level, which de-allocates all resources that were part of the application's state when a process is terminated, such as dynamic memory areas, open file handles and sockets. Moreover, this rejuvenation takes advantage of initialization mechanisms that programmers introduce in their application to bring it in its initial state. This type of rejuvenation does not affect the part of the state that belongs to the software environment (i.e., the OS or other applications). For instance, OS resources that are not released when a process is terminated (e.g., a temporary file stored into a disk, or leaked memory allocated by other applications) are not reclaimed by rejuvenation.

OS Reboot: This action restarts the OS, and typically all the applications running in the same OS. This form of rejuvenation is also referred to as *node reboot* in distributed systems. Although it is possible to preserve the state of applications and restore it after the reboot, this possibility is never considered in SAR studies. In its simplest form, the OS reboot involves a hardware reset (memory and hardware devices are re-initialized and tested before software is started), the boot of the OS kernel, and the restart of all user-space applications. More sophisticated schemas reduce the time required to OS reboot, by bringing the OS in its initial state without involving a hardware reset [Nellitheertha 2004; Oracle 2012; Alonso et al. 2011b], as in the case of the Linux and the Solaris OSs. This is achieved by invalidating the contents of main memory and restarting the execution of the OS from its entry point. Another problem is represented by the loss of performance after the OS reboot: since the contents of the file cache (i.e., a copy of file contents stored in main memory to speed up file accesses) are lost after the reboot, the performance of the system is degraded until the file cache is re-populated. To solve this problem, the *warm-cache reboot* mechanism [Kourai 2010] preserves the file cache on main memory during the reboot and enables an operating system to restore the file cache after the reboot. Moreover, this mechanism must handle the problem of file cache inconsistency with disks (i.e., files contents that have been modified in main memory, but have not been written back to the disk before the reboot—since pending

modifications may be incomplete or affected by data corruption, they should be discarded after the reboot). *CacheMind* [Kourai 2010] handles this problem by running the OS in a virtual machine, and by using the Virtual Machine Monitor (i.e., a software layer that creates a virtual machine environment for an OS and its applications) to keep track of the status of file cache pages, in order to guarantee consistency after the reboot. It should be noted that, when an OS runs in a virtual machine and it is rejuvenated by an OS reboot, the virtual machine infrastructure (i.e., the VMM and the virtual machine instance in which the OS runs) is not affected by rejuvenation.

Virtual Machine Monitor (VMM) and Virtual Machine (VM) Restart: Software rejuvenation can act on a virtual machine infrastructure, by restarting the VMM and/or its VMs. In [Machida et al. 2010], several alternative strategies are devised, depending on whether rejuvenation only affects the VMM, or also involves VMs running on top of the VMM. In *Cold-VM* rejuvenation, the VMs are also restarted when the VMM is rejuvenated. In *Warm-VM* rejuvenation, the execution state of each VM (including the OS and applications running in the VM) are stored to persistent memory, and resumed after the restart of the VMM, in order to reduce the downtime of restarting VMs and their services (although the software running in the VMs is not rejuvenated). This operation can be quickly performed by adopting an *on-memory suspend/resume* mechanism, in which the memory images of VMs is preserved in main memory during the VMM restart, in order to avoid slow read/write operations to persistent storage (as in the case of *RootHammer* [Kourai and Chiba 2007; 2011]). In *Migrate-VM* rejuvenation, the downtime is further reduced by migrating a VM to another host while the VMM is being rejuvenated, in order to make them available during rejuvenation. This latter schema also does not rejuvenate VMs, and is limited by the capacity of other hosts to accept migrated VMs. The best technique (or the best combination of them) depends on the speed of storing/migrating the state of VMs and on the capacity of hosts, as well as on the aging rate of VMs and VMMs, therefore the rejuvenation policy should be determined according to these factors [Machida et al. 2010].

Cluster Failover: A cluster system is a system composed by a set of replicated servers that provide the same service, with the aim to provide high performance and reliability. In a cluster system, an individual server can be rejuvenated (e.g., by means of application restart or OS reboot) while the other replica are active and the workload is redirected to them (at the cost of reducing cluster performance during rejuvenation) [Avritzer et al. 2007; Wang et al. 2007; Xie et al. 2004]. Another approach is to activate a standby (i.e., idle) replica of the system when rejuvenation is triggered [Park and Kim 2002]. In [Silva et al. 2007; Silva et al. 2009], a cluster failover framework for web applications based on virtualization is proposed, namely *VM-Rejuv*. The framework consists of a *Load Balancer*, an *Active Server*, and a *Standby Server*, each running in a dedicated VM. The Load Balancer redirects requests to the Active Server while it is correctly working, and monitors the Active Server for aging symptoms (e.g., performance falls below a threshold). When rejuvenation is triggered, new requests are redirected to the Standby Server; the Active Server is rejuvenated only after that all pending requests have been processed and session data have been migrated to the Standby Server, in order to assure a clean restart (i.e., rejuvenation does not cause the loss of session data and the failure of user requests). This framework can be implemented in a cost-effective way by using off-the-shelf application servers, monitoring, and load balancing software, at the cost of a moderate overhead.

Application-generic actions do not make use of application-specific features, but rely on *restarting* the system or its components to perform software rejuvenation, or they activate a replica of the system. By following this approach, the system or the component being rejuvenated is brought to its *initial state*, which is assured to be aging-free.

This kind of rejuvenation is simple to implement since it makes use of initialization mechanisms of the system.

By contrast, application-specific rejuvenation is tailored for a specific system: it aims at reducing the cost required to perform rejuvenation, in terms of time to rejuvenate and system downtime, by cleaning a specific aging-affected resource. This kind of rejuvenation exploits peculiar features of the system, such as the peculiarities of the domain, of the software architecture, and of the kind of resources being managed. Some examples of application-specific rejuvenation in the context of OSs are represented by flushing of kernel tables and filesystem de-fragmentation. Other application-specific approaches have been discussed by SAR papers in the following contexts:

Component-based systems: An individual component or part of an application can be restarted to perform software rejuvenation. Compared to an application restart, this kind of rejuvenation focuses on a subset of the application state, such as the set of resources allocated by an individual process in a multi-process application. This rejuvenation approach aims to reduce the time required for rejuvenation and the application downtime, by avoiding the restart of parts of the application that are not affected by software aging. A well-known example is represented by the Apache web server [Matias et al. 2010b; Grottke et al. 2006], in which a master process spawns a set of concurrent child processes that handle client requests. This system is rejuvenated by restarting a child process after it has handled a given number of requests, or when the process is killed by the user or by other applications (e.g., in [Grottke et al. 2006], child processes are periodically restarted by the *cron* daemon to perform log rotation). A more general form of component restart is represented by *microrebootable software* [Candea et al. 2002; Candea et al. 2004], in which a system is decomposed in components that are loosely-coupled (e.g., they do not share the same memory address space) and stateless (e.g., important state is located in dedicated storage outside the application), in order to quickly restart a component without affecting the other ones. This approach has been successfully implemented in a J2EE application [Candea et al. 2004] (in which individual Enterprise Java Beans can be restarted) and in a mission-critical system [Candea et al. 2002] (consisting of a set of components running in different Java Virtual Machines). A limitation of component rejuvenation is that it can be applied only in systems made up of individually-restartable components; if this is not the case, the system has to be modified based on the microreboot schema.

Embedded systems: Modern embedded systems are characterized by a high degree of complexity. These systems are resource-constrained, therefore the dynamic memory requirements of software tasks have to be carefully estimated at design time. These aspects expose them to subtle aging-related bugs. In [Sundaram et al. 2007], the Opportunistic Micro-Rejuvenation approach was proposed for resource-constrained multitasking embedded systems. This approach is based on a Shared Supplementary Memory (SSM), which is a shared memory area used by tasks that exceed their stack or heap limits: when a stack or heap overflows, it is reallocated in order to grow in the SSM. When the SSM usage is greater than a threshold, the task that consumes most of the SSM is rejuvenated. This approach allows to mitigate inaccurate estimations of memory requirements and to increase the reliability of embedded systems.

Long-running desktop applications: The *Libckp* checkpointing library has been proposed in [Wang et al. 1995] for increasing the reliability of long-running UNIX applications. This library provides an API that allows programmers to save the program state at a given point in the program, and to restore that state in the case of failures. [Wang et al. 1995] proposed several usage scenarios of this library related to software rejuvenation. A first scenario is to bypass long program initialization (e.g., a large amount of data is read from a remote database), by storing the program state just after initialization, and using that state to restart the program. Another scenario

is to periodically revert the program to a previously checkpointed clean state in order to avoid aging symptoms and to restart the program. A requirement for this solution is that the state in which rejuvenation is performed does not contain any useful information, which makes this solution application-specific. This schema has been proposed for long-running programs that consist of a large number of *independent* iterations, and has been experimented in the context of CAD (Computer-Aided Design) applications, simulation programs and signal processing applications.

Stateful distributed systems: In distributed systems with replicas (e.g., cluster systems), software rejuvenation can be performed by deactivating a replica while the load is redirected to the remaining ones. However, this schema assumes that the system is stateless; if this is not the case, the state of the replica being rejuvenated may become obsolete and therefore be inconsistent with the other replicas that continue to execute. In order to apply rejuvenation to stateful distributed systems, a framework has been proposed in [Tai et al. 2005] based on the notion of *eventual consistency*, that is, a concurrency control protocol is adopted that guarantees eventual rather than immediate consistency among replicas. When a replica undergoes rejuvenation, update requests are saved in a buffer and are replayed by a sequencer, i.e., a node in a distributed system that assures that all replica receive requests in the same global order. In this way, the rejuvenated replica will appear as a "slow" replica and its state will eventually converge to the state of the other ones.

Database management systems: Software rejuvenation approaches have been devised specifically tailored for software aging phenomena in DBMSs. An instance of software aging phenomena is represented by the gradual increase in *shared pool latches contention*, that is, the increase of the waiting time for accessing shared memory areas due to synchronization mechanisms [Tsai et al. 2006; Cassidy et al. 2002]: this phenomenon is caused by the exhaustion and/or fragmentation of shared memory areas, and can be mitigated by flushing data in shared memory areas to the disk. Another instance is the exhaustion of disk space in DBMS systems due to the growth of the redo log file (i.e., a file that keeps track of committed transactions, and that is used to restore the database state from the last database backup in the case of a failure). The analysis in [Bobbio et al. 2001] copes with this phenomenon by regularly archiving the redo log file to a secondary disk. In [Baker and Sullivan 1992] an OS extension, namely the *Recovery Box*, is proposed to provide a quick recovery mechanism for UNIX server applications, which has been adopted to improve recovery of a DBMS system. The Recovery Box provides an API (to be used by both the applications and the OS itself) to save and restore relevant system state across reboots (e.g., session data in transaction-based systems), which is stored in non-volatile memory. This recovery speed of the Postgres DBMS is improved by storing in the Recovery Box initialization data (e.g., a cache with system catalogs), internal data structures such as hash tables and linked lists, and client connections (which have to be detected through a timeout and restarted from the client side, and requires the client to check whether its last transaction committed). Using the Recovery Box (or similar supports), a DBMS can be quickly rejuvenated by reducing its restart time.

Runtime supports in programming languages and frameworks: a well-known approach that mitigates software aging phenomena is represented by *garbage collection*, which inspects the software state in order to reclaim resources that are not reachable (i.e., there are no references to the resource) and therefore represent a waste. It is adopted in several programming languages (e.g., *Java* and *C#*) to relieve programmers of memory management duties and therefore to prevent memory and resource leak bugs, and it is provided by runtime supports in programming languages and frameworks. Garbage collection cannot assure the absence of aging-related bugs, since the leaked resource (i.e., a resource that will not be used anymore by the

program but that it is still allocated) may have references and therefore is not reclaimable [Carrozza et al. 2010]. The *Melt* approach [Bond and McKinley 2008] has been proposed for mitigating this problem through *leak tolerance*: it identifies reachable objects that are not accessed, and moves them to the disk in order to free memory. This approach does not completely rejuvenate the software, but increases its time-to-aging-failure in order to improve the user experience and to give developers more time and information to fix leaks. Other approaches have been proposed for detecting and diagnose leaks in garbage-collected environments. In [Gama and Donsez 2008], stale references in the OSGi web application framework (i.e., references to services that have been unloaded and therefore are invalid) are identified by tracking service reference objects through Aspect Oriented Programming. In [Xu and Rountev 2008b], Java containers (e.g., hashmaps and lists) are profiled in order to identify stale objects in the containers, by analyzing the time since last retrieval and memory consumption of objects. In order to make garbage collection application-independent, the *Kernel-Assisted Leak toleration* (KAL) schema, proposed in [Jeong et al. 2010], introduces a OS kernel extension to reclaim leaked memory in C/C++ software with low intrusiveness and overhead. KAL takes a snapshot of memory while the application is running (using a *precopy* schema that avoids to suspend the application), analyzes the snapshot to identify leaks (this analysis can be performed on a remote host in order to reduce the overhead), and reclaims memory by invoking standard memory management routines.

Several SAR studies are **not based on any particular rejuvenation technique**, and assume some form of application-generic rejuvenation (e.g., application restart or OS reboot). This is the case of most of model-based studies [Huang et al. 1995; Pfening et al. 1996; Suzuki et al. 2003; Koutras and Platis 2008]. Therefore, we introduce the "unspecified rejuvenation" class. This class includes studies that focus on determining the optimal rejuvenation time, no matter what specific rejuvenation policy is adopted, as in the case of model-based studies that are not tailored to a specific system.

Figure 13 shows the number of times each rejuvenation approach has been proposed among the surveyed papers. We found that about two out of three papers belong to the "unspecified" class, that is, no particular rejuvenation technique is mentioned. **The high number of "unspecified rejuvenation" papers denotes that the focus is often on optimal time scheduling rather than on the design of effective rejuvenation actions**. Among the remaining studies, application-generic actions are evenly distributed, where *Application Restart* is the most commonly adopted [Huang et al. 1995; Silva et al. 2006; Machida et al. 2011; Koutras and Platis 2011].

Application-specific rejuvenation techniques represent a minority of approaches, where *Component Restart* is the most studied form of application-specific approach, due to the remarkable number of studies on the Apache Web Server that exploit the multi-process architecture of this software. Other techniques that rejuvenate the system state at a more detailed level, such as ad-hoc state checkpointing mechanisms [Wang et al. 1995], are seldom considered.

Application-specific approaches have the **greatest potential to improve the speed of rejuvenation and reduce the probability of failures**. These techniques are effective at reducing the cost of rejuvenation since they do not bring the system to its *initial state*, and avoid to redo work for reconstructing the relevant system state (e.g., to restart transactions that were taking place at the time of rejuvenaton). This issue is negligible in the case of *stateless* applications (e.g., a web server) [Matias Jr. 2006; Candea et al. 2004], although it has great importance in the case of *stateful* applications [Tai et al. 2005; Baker and Sullivan 1992; Wang et al. 1995]. However, this kind of approach is perceived as not cost-effective, since it needs to be tailored to the specific application in order to save only the relevant part of the system state and avoid to include aging-related errors in the checkpoint [Wang et al. 1995], which requires de-

velopment efforts and can be error-prone. **This area has received little attention among SAR studies at the time of writing**, and may deserve attention in future in order to further advance the field of software rejuvenation.

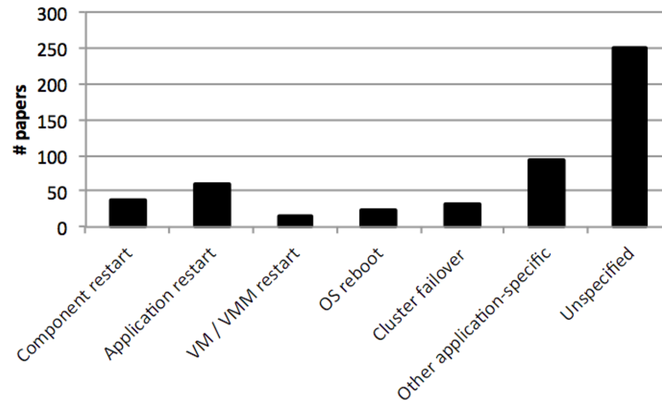


Fig. 13: Software rejuvanation approaches.

5. DISCUSSION

This paper surveyed and analyzed the literature of Software Aging and Rejuvenation with respect to four relevant aspects: the type of studies that have been conducted, the type of systems that exhibited aging phenomena, the type of aging effects that have been observed in real systems and their related aging indicators, and the techniques that have been proposed so far to rejuvenate software systems.

The first dimension (Section 4.1) highlighted that many studies have been devoted on designing analytical models for scheduling the rejuvenation time. These models are becoming more and more refined and comprehensive. However, the works addressing model-based rejuvenation often lack experimentation on real systems, and in most cases models are validated by numerical examples, or by simulations. This is an important concern towards making these studies useful for practical scenarios, because (i) models make some kind of assumptions about the system being modeled, which can be validated only by comparing the actual behavior of the system with the prediction of models, and (ii) the deployment of model-based rejuvenation on real systems can reveal practical issues that would be neglected otherwise, such as the problem of relating measurements of aging indicators with the parameters of the models and with the rejuvenation schedule. Moreover, experimentation on real systems would also provide examples of how software rejuvenation strategies can be applied, and encourage their adoption by practitioners.

A remarkable trend, which represents an interesting research area, is the development of hybrid approaches in the context of real-world systems, which leverage measurement-based approaches to exploit model-based approaches. A future direction is the implementation of frameworks for online monitoring and aging estimation to enable the adoption of Software Rejuvenation in existing systems, in order to increase in the industrial world the perception and the awareness of the Software Aging problem. A noteworthy case that pioneered this direction is the Software Rejuvenation Agent integrated in the IBM Director® server management tool, which allows system

administrators to schedule Software Rejuvenation of IBM xSeries® cluster servers [Castelli et al. 2001].

The need for additional studies on real systems is also highlighted in the analysis of the second dimension (Section 4.2). The non-negligible percentage of studies on Software Aging in safety-critical systems suggests that it is worth to further investigate this issue in safety-critical systems. In fact, safety-critical systems and their software are increasingly complex, and consequently it is more and more difficult to assure that the software is free from aging-related bugs. Therefore, it becomes important to research means to take into account aging-related bugs in the *design and validation process* of safety-critical software, since these bugs can affect safety requirements that are imposed on long-running systems by safety certification standards [RTCA 1992]. The analysis of the type of systems also highlighted that Software Aging has been experienced in novel application scenarios such as cloud computing systems and embedded and ubiquitous systems. It is therefore likely that this phenomenon will have to be studied and mitigated in future generations of software systems.

The analysis of aging effects and aging indicators in Section 4.3 reports that memory and performance issues were the most studied in the literature. However, further research is needed for the investigation of other kinds of Software Aging phenomena. There is evidence of several other types of aging bugs, such as numerical errors, storage-related bugs, and bugs related to the management of system-dependent data structures [Huang et al. 1995; Bobbio and Sereno 1998; Grottke et al. 2008]. In particular, there is no suitable approach among the analyzed studies that copes with the accumulation of numerical errors, for which there is not an aging indicator that could be used by traditional measurement-based approaches. We also believe that it is important to analyze more in-detail the kind of aging-related bugs that affect real systems, in order to extend SAR research towards aging effects that have been neglected in the past. Past field failure data studies [Grottke et al. 2010; Chillarege 2011] provided an estimate of the extent of aging-related bugs, although SAR researchers would benefit from studies that look at the kind of aging effects caused by aging-related bugs, as well as from data about more software systems. This analysis pointed out that Software Rejuvenation is also being considered for improving the security of software systems, by cleaning-up the system state and regenerating compromised code and data (e.g., by replacing passwords and cryptographic keys, or by adopting a new system configuration). The problem of scheduling software rejuvenation is still open, since it is unclear what is the best way to detect the onset of security-related aging effects and to predict the time to security-related failures.

Finally, in Section 4.4 we analyzed software rejuvenation techniques. This topic received less attention than the determination of the optimal rejuvenation schedule (Fig. 5). However, rejuvenation techniques are useful to keep low the cost of software rejuvenation, to achieve high availability and to reduce performance loss. Most studies adopt or assume an application-independent approach, that involves a software restart. Other approaches are being developed that are application-specific, that is, they exploit special features of the system in order to improve the efficiency of Software Rejuvenation. Research on this topic has provided interesting results in several contexts, such as embedded systems, distributed systems, and DBMSs. In particular, the problem of accounting for the state in stateful systems has to be addressed in some domains in order to avoid service disruption and data loss, and to make Software Rejuvenation affordable in these contexts. Another remarkable result is the selective rejuvenation of parts of the system by using component restarts. Since these approaches are application-dependent, additional research could be devoted to provide methodolo-

gies and frameworks for integrating Software Rejuvenation in their specific system in a cost-effective way.

REFERENCES

ADAMS, E. 1984. Optimizing preventive service of software products. *IBM Journal of Research and Development 28,* 1, 2–14.

ALONSO, J., BELANCHE, L., AND AVRESKY, D. 2011a. Predicting software anomalies using machine learning techniques. *Proceedings - 2011 IEEE International Symposium on Network Computing and Applications, NCA 2011*, 163–170.

ALONSO, J., MATIAS, R., VICENTE, E., CARVALHO, A., AND TRIVEDI, K. 2011b. A comparative evaluation of software rejuvenation strategies. In *Software Aging and Rejuvenation (WoSAR), 2011 IEEE Third International Workshop on*. 26 –31.

ALONSO, J., TORRES, J., BERRAL, J., AND GAVALDÀ, R. 2010. J2ee instrumentation for software aging root cause application component determination with aspectj. *Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum, IPDPSW 2010*.

ANDRADE, E., MACHIDA, F., KIM, D., AND TRIVEDI, K. 2011. Modeling and analyzing server system with rejuvenation through sysml and stochastic reward nets. *Proceedings of the 2011 6th International Conference on Availability, Reliability and Security, ARES 2011*, 161–168.

ANDRZEJAK, A. AND SILVA, L. 2007. Deterministic models of software aging and optimal rejuvenation schedules. *10th IFIP/IEEE International Symposium on Integrated Network Management 2007, IM '07*, 159–168.

ANDRZEJAK, L. SILVA, A. 2008. Using machine learning for non-intrusive modeling and prediction of software aging. In *IEEE Network Operations and Management Symposium*.

ANTUNES, J. 2008. Detection and prediction of resource-exhaustion vulnerabilities. In *International Symposium on Software Reliability Engineering*.

ARAUJO, J., MATOS, R., MACIEL, P., MATIAS, R., AND BEICKER, I. 2011a. Experimental evaluation of software aging effects on the eucalyptus cloud computing infrastructure. In *Middleware*.

ARAUJO, J., MATOS, R., MACIEL, P., VIEIRA, F., MATIAS, R., AND TRIVEDI, K. 2011b. Software rejuvenation in eucalyptus cloud computing infrastructure: A method based on time series forecasting and multiple thresholds. In *Software Aging and Rejuvenation (WoSAR), 2011 IEEE Third International Workshop on*. 38 –43.

AUNG, K., PARK, K., AND PARK, J. 2005. A model of its using cold standby cluster. *Lecture Notes in Computer Science 3815 LNCS*, 1–10.

AVIZIENIS, A., LAPRIE, J., RANDELL, B., AND LANDWEHR, C. 2004. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on 1,* 1, 11–33.

AVRITZER, A., BONDI, A., AND WEYUKER, E. 2007. Ensuring system performance for cluster and single server systems. *Journal of Systems and Software 80,* 4, 441–454.

AVRITZER, A. AND WEYUKER, E. 1997. Monitoring smoothly degrading systems for increased dependability. *Empirical Software Engineering 2,* 1, 59–77.

AVRITZER, A. AND WEYUKER, E. J. 2004. The role of modeling in the performance testing of e-commerce applications. *IEEE Transactions on Software Engineering 30,* 12, 1072–1083.

BAKER, M. AND SULLIVAN, M. 1992. The Recovery Box: Using Fast Recovery to provide High Availability in the UNIX Environment. In *Proc. Summer 1992 USENIX Conference*. 31–43.

BALAKRISHNAN, M., PULIAFITO, A., TRIVEDI, K., AND VINIOTIS, Y. 1997. Buffer losses vs. deadline violations for abr traffic in an atm switch: a computational approach. *Telecommunication Systems 7,* 1-3, 105–123.

BAO, Y., SUN, X., AND TRIVEDI, K. 2005. A workload-based analysis of software aging, and rejuvenation. *Reliability, IEEE Transactions on 54,* 3.

BERNSTEIN, L. 1993. Innovative technologies for preventing network outages. *AT & T TECH J. 72,* 4, 4–10.

BERNSTEIN, L. AND KINTALA, C. 2004. Software rejuvenation. *CrossTalk 17,* 8, 23–26.

BOBBIO, A. AND SERENO, M. 1998. Fine grained software rejuvenation models. In *Computer Performance and Dependability Symposium, 1998. IPDS'98. Proceedings. IEEE International*. IEEE, 4–12.

BOBBIO, A., SERENO, M., AND ANGLANO, C. 2001. Fine grained software degradation models for optimal rejuvenation policies. *Performance Evaluation 46,* 1, 45–62.

BOND, M. AND MCKINLEY, K. 2008. Tolerating memory leaks. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*, 109–125.

BOVENZI, A., COTRONEO, D., PIETRANTUONO, R., AND RUSSO, S. 2011. Workload characterization for software aging analysis. In *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on*. 240 –249.

CANDEA, G., CUTLER, J., FOX, A., DOSHI, R., GARG, P., AND GOWDA, R. 2002. Reducing recovery time in a small recursively restartable system. In *Dependable Systems and Networks, 2002. Proc. Int'l. Conf.*

CANDEA, G., KAWAMOTO, S., FUJIKI, Y., FRIEDMAN, G., AND FOX, A. 2004. Microreboot—a technique for cheap recovery. In *Proc. Symp. on Operating Systems Design & Implementation*. USENIX Association, 31–44.

CARROZZA, G., COTRONEO, D., NATELLA, R., PECCHIA, A., AND RUSSO, S. 2010. Memory leak analysis of mission-critical middleware. *Journal of Systems and Software 83,* 9, 1556–1567.

CASSIDY, K., GROSS, K., AND MALEKPOUR, A. 2002. Advanced pattern recognition for detection of complex software aging phenomena in online transaction processing servers. In *Dependable Systems and Networks, 2002. Proc. Int'l. Conf.*

CASTELLI, V., HARPER, R., HEIDELBERGER, P., HUNTER, S., TRIVEDI, K., VAIDYANATHAN, K., AND ZEGGERT, W. 2001. Proactive management of software aging. *IBM Journal of Research and Development 45,* 2, 311–332.

CHILLAREGE, R. 2011. Understanding bohr-mandel bugs through odc triggers and a case study with empirical estimations of their field proportion. In *Software Aging and Rejuvenation (WoSAR), 2011 IEEE Third International Workshop on*. 7 –13.

COTRONEO, D., NATELLA, R., PIETRANTUONO, R., AND RUSSO, S. 2010. Software aging analysis of the linux operating system. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st Int'l. Symp.*

COTRONEO, D., NATELLA, R., PIETRANTUONO, R., AND RUSSO, S. 2011a. Software aging and rejuvenation: Where we are and where we are going. In *Software Aging and Rejuvenation (WoSAR), 2011 IEEE Third International Workshop on*. 1 –6.

COTRONEO, D., ORLANDO, S., PIETRANTUONO, R., AND RUSSO, S. 2011b. A measurement-based ageing analysis of the jvm. *Software Testing Verification and Reliability*.

COTRONEO, D., ORLANDO, S., AND RUSSO, S. 2007. Characterizing aging phenomena of the java virtual machine. In *Reliable Distributed Systems, 2007. 26th IEEE Int'l. Symp.*

COX, B., EVANS, D., FILIPI, A., ROWANHILL, J., HU, W., DAVIDSON, J., KNIGHT, J., NGUYEN-TUONG, A., AND HISER, J. 2006. N-variant systems: a secretless framework for security through diversity. In *Proceedings of the 15th conference on USENIX Security Symposium-Volume 15*. USENIX Association, 9.

DU, X., QI, Y., HOU, D., CHEN, Y., AND ZHONG, X. 2009. Modeling and performance analysis of software rejuvenation policies for multiple degradation systems. *Proceedings - International Computer Software and Applications Conference 1*, 240–245.

ETO, H., DOHI, T., AND MA, J. 2008. Simulation-based optimization approach for software cost model with rejuvenation. *Lecture Notes in Computer Science 5060 LNCS*, 206–218.

FERREIRA, T., MATIAS, R., MACEDO, A., AND ARAUJO, L. 2011. An experimental study on memory allocators in multicore and multithreaded applications. In *Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2011 12th International Conference on*. 92 –98.

GAMA, K. AND DONSEZ, D. 2008. Service coroner: A diagnostic tool for locating osgi stale references. *EUROMICRO 2008 - Proceedings of the 34th EUROMICRO Conference on Software Engineering and Advanced Applications, SEAA 2008*, 108–115.

GARG, S., KINTALA, C., HUANG, Y., AND TRIVEDI, K. 1996. Minimizing completion time of a program by checkpointing and rejuvenation. *Performance Evaluation Review 24,* 1, 252–261.

GARG, S., PULIAFITO, A., TELEK, M., AND TRIVEDI, K. 1995. Analysis of software rejuvenation using markov regenerative stochastic petri net. In *Software Reliability Engineering, 1995. Proc., Sixth Int'l. Symp.*

GARG, S., PULIAFITO, A., TELEK, M., AND TRIVEDI, K. 1998a. Analysis of preventive maintenance in transactions based software systems. *Computers, IEEE Transactions on 47,* 1.

GARG, S., VAN MOORSEL, A., VAIDYANATHAN, K., AND TRIVEDI, K. 1998b. A methodology for detection and estimation of software aging. In *Software Reliability Engineering, 1998. Proc. Ninth Int'l. Symp.*

GRAY, J. 1985. Why Do Computers Stop and What Can Be Done About It? In *Proc. Symp. on Reliability in Distributed Software and Database Systems*. 3–11.

GROTTKE, M., LI, L., VAIDYANATHAN, K., AND TRIVEDI, K. 2006. Analysis of software aging in a web server. *Reliability, IEEE Transactions on 55,* 3.

GROTTKE, M., MATIAS, R., AND TRIVEDI, K. 2008. The fundamentals of software aging. In *Software Reliability Engineering Workshops, 2008. IEEE Int'l. Conf.*

GROTTKE, M., NIKORA, A., AND TRIVEDI, K. 2010. An empirical investigation of fault types in space mission system software. In *Dependable Systems and Networks (DSN), 2010 Int'l. Conf.*

HEINE, D. AND LAM, M. 2006. Static detection of leaks in polymorphic containers. *Proceedings - International Conference on Software Engineering 2006*, 252–261.

HOFFMANN, G., TRIVEDI, K., AND MALEK, M. 2007. A best practice guide to resource forecasting for computing systems. *Reliability, IEEE Transactions on 56,* 4.

HUANG, Y., ARSENAULT, D., AND SOOD, A. 2006. Scit-dns: Critical infrastructure protection through secure dns server dynamic updates. *Journal of High Speed Networks 15,* 1, 5–19.

HUANG, Y., CHUNG, P., KINTALA, C., LIANG, D., AND WANG, C. 1998. Nt-swift: Software implemented fault tolerance on windows nt. In *Proceedings of the 1998 USENIX WindowsNT Symposium*.

HUANG, Y. AND KINTALA, C. 1993. Software implemented fault tolerance: Technologies and experience. In *1993 IEEE International Symposium on Fault-Tolerant Computing*.

HUANG, Y., KINTALA, C., KOLETTIS, N., AND FULTON, N. 1995. Software rejuvenation: analysis, module and applications. In *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers, Twenty-Fifth Int'l. Symp.*

HUANG, Y., KINTALA, C. M. R., BERNSTEIN, L., AND WANG, Y.-M. 1996. Components for software fault tolerance and rejuvenation. *AT&T technical journal 75,* 2, 29–37.

JEONG, J., SEO, E., CHOI, J., KIM, H., JO, H., AND LEE, J. 2010. Kal: Kernel-assisted non-invasive memory leak tolerance with a general-purpose memory allocator. *Software - Practice and Experience 40,* 8, 605–625.

JIA, Y.-F., ZHAO, L., AND CAI, K.-Y. 2008. A nonlinear approach to modeling of software aging in a web server. In *Software Engineering Conf., 2008.15th Asia-Pacific*.

JUNG, Y. AND YI, K. 2008. Practical memory leak detector based on parameterized procedural summaries. In *Proceedings of the 7th international symposium on Memory management*. ACM, 131–140.

KAJKO-MATTSSON, M. 2001. Can we learn anything from hardware preventive maintenance? In *Engineering of Complex Computer Systems, 2001. Proceedings. Seventh IEEE International Conference on*. IEEE, 106–111.

KICIMAN, E. AND LIVSHITS, B. 2007. Ajaxscope: A platform for remotely monitoring the client-side behavior of web 2.0 applications. *SOSP'07 - Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, 17–30.

KIM, D., YANG, C., AND PARK, J. 2007. Adaptation mechanisms for survivable sensor networks against denial of service attack. *Proceedings - Second International Conference on Availability, Reliability and Security, ARES 2007*, 575–579.

KOURAI, K. 2010. Cachemind: Fast performance recovery using a virtual machine monitor. In *Dependable Systems and Networks Workshops (DSN-W), 2010 International Conference on*. 86 –92.

KOURAI, K. AND CHIBA, S. 2007. A fast rejuvenation technique for server consolidation with virtual machines. In *Dependable Systems and Networks, 2007. 37th Int'l. Conf.*

KOURAI, K. AND CHIBA, S. 2011. Fast software rejuvenation of virtual machine monitors. *Dependable and Secure Computing, IEEE Transactions on 8,* 6.

KOUTRAS, V. AND PLATIS, A. 2008. Modeling perfect and minimal rejuvenation for client server systems with heterogeneous load. In *Dependable Computing, 2008. 14th IEEE Pacific Rim Int'l. Symp.*

KOUTRAS, V. AND PLATIS, A. 2011. Applying partial and full rejuvenation in different degradation levels. In *Software Aging and Rejuvenation (WoSAR), 2011 IEEE Third International Workshop on*. 20 –25.

LI, L., VAIDYANATHAN, K., AND TRIVEDI, K. 2002. An approach for estimation of software aging in a web server. In *Empirical Software Engineering, 2002. Proc. 2002 Int'l. Symp.*

LIU, Y., MA, Y., HAN, J. J., LEVENDEL, H., AND TRIVEDI, K. S. 2005. A proactive approach towards always-on availability in broadband cable networks. *Computer Communications 28,* 1, 51–64.

LIU, Y., TRIVEDI, K., MA, Y., HAN, J., AND LEVENDEL, H. 2002. Modeling and analysis of software rejuvenation in cable modem termination systems. In *Software Reliability Engineering, 2002. Proc. 13th Int'l. Symp.*

MACEDO, A., FERREIRA, T., AND MATIAS, R. 2010. The mechanics of memory-related software aging. In *Software Aging and Rejuvenation (WoSAR), 2010 IEEE Second Int'l. Workshop on*.

MACHIDA, F., KIM, D. S., AND TRIVEDI, K. 2010. Modeling and analysis of software rejuvenation in a server virtualized system. In *Software Aging and Rejuvenation (WoSAR), 2010 IEEE Second Int'l. Workshop on*.

MACHIDA, F., NICOLA, V., AND TRIVEDI, K. 2011. Job completion time on a virtualized server subject to software aging and rejuvenation. In *Software Aging and Rejuvenation (WoSAR), 2011 IEEE Third International Workshop on*. 44 –49.

MAGALHAES, J. AND SILVA, L. 2010. Prediction of performance anomalies in web-applications based-on software aging scenarios. In *Software Aging and Rejuvenation (WoSAR), 2010 IEEE Second Int'l. Workshop on*.

MARSHALL, E. 1992. Fatal error: how patriot overlooked a scud. *Science 255,* 5050, 1347–1347.

MATIAS, R., BARBETTA, P., TRIVEDI, K., AND FILHO, P. 2010a. Accelerated degradation tests applied to software aging experiments. *Reliability, IEEE Transactions on 59,* 1.

MATIAS, R., TRIVEDI, K., AND MACIEL, P. 2010b. Using accelerated life tests to estimate time to software aging failure. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st Int'l. Symp.*

MATIAS JR., R., F. P. 2006. An experimental study on software aging and rejuvenation in web servers. *Proceedings - International Computer Software and Applications Conference 1*, 189–196.

MATOS, R., MACIEL, P., AND MATIAS, R. 2011. Software aging issues on the eucalyptus cloud computing infrastructure. In *IEEE International Conference on Systems, Man, and Cybernetics*.

MYINT, M. AND THEIN, T. 2010. Availability improvement in virtualized multiple servers with software rejuvenation and virtualization. *SSIRI 2010 - 4th IEEE International Conference on Secure Software Integration and Reliability Improvement*, 156–162.

NAGARAJAN, A. AND SOOD, A. 2010. Scit and ids architectures for reduced data ex-filtration. In *Dependable Systems and Networks Workshops (DSN-W), 2010 International Conference on*. IEEE, 164–169.

NAKSINEHABOON, N., TAERAT, N., LEANGSUKSUN, C., CHANDLER, C., AND SCOTT, S. 2010. Benefits of software rejuvenation on hpc systems. *Proceedings - International Symposium on Parallel and Distributed Processing with Applications, ISPA 2010*, 499–506.

NELLITHEERTHA, H. 2004. Reboot Linux faster using kexec. In *developerWorks technical library*.

NGUYEN, Q. AND SOOD, A. 2009. Quantitative approach to tuning of a time-based intrusion-tolerant system architecture. In *Proc. 3rd Workshop Recent Advances on Intrusion-Tolerant Systems*. 132–139.

NI, Q., SUN, W., AND MA, S. 2008. Memory leak detection in sun solaris os. In *International Symposium on Computer Science and Computational Technology*.

NING, M., YONG, Q., DI, H., XIA, P., AND YING, C. 2007. Application server aging prediction model based on wavelet network with adaptive particle swarm optimization algorithm. *Lecture Notes in Computer Science 4682 LNAI*, 14–25.

NOVARK, G., BERGER, E., AND ZORN, B. 2009. Efficiently and precisely locating memory leaks and bloat. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 397–407.

OKAMURA, H. AND DOHI, T. 2011. A pomdp formulation of multistep failure model with software rejuvenation. In *Software Aging and Rejuvenation (WoSAR), 2011 IEEE Third International Workshop on*. 14 –19.

OKAMURA, H., MIYAHARA, S., AND DOHI, T. 2003. Dependability analysis of a transaction-based multi-server system with rejuvenation. *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences E86-A,* 8, 2081–2090.

OKAMURA, H., MIYAHARA, S., AND DOHI, T. 2005. Rejuvenating communication network system under burst arrival circumstances. *IEICE TRANSACTIONS on Communications E88-B,* 12, 4498–4506.

OPPENHEIMER, D., GANAPATHI, A., AND PATTERSON, D. A. 2003. Why do internet services fail, and what can be done about it? In *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*. USITS'03. USENIX Association, Berkeley, CA, USA, 1–1.

ORACLE. 2012. *Booting and Shutting Down Oracle Solaris on x86 Platforms*. Oracle Solaris 11 Information Library.

PARK, J. AND CHOI, B. 2012. Automated memory leakage detection in android based systems. *International Journal of Control and Automation 5,* 2, 35–42.

PARK, K. AND KIM, S. 2002. Availability analysis and improvement of active/standby cluster systems using software rejuvenation. *Journal of Systems and Software 61,* 2, 121–128.

PARNAS, D. 1994. Software aging. In *Proceedings of the 16th international conference on Software engineering*. IEEE Computer Society Press, 279–287.

PFENING, A., GARG, S., PULIAFITO, A., TELEK, M., AND TRIVEDI, K. 1996. Optimal software rejuvenation for tolerating soft failures. *Performance Evaluation 27-28*, 491–506.

ROBB, D. 2000. Defragmenting really speeds up windows nt machines. *Spectrum, IEEE 37,* 9, 74–77.

ROEDER, T. AND SCHNEIDER, F. 2010. Proactive obfuscation. *ACM Transactions on Computer Systems (TOCS) 28,* 2, 4.

RTCA. 1992. DO-178B Software Considerations in Airborne Systems and Equipment Certification. *Requirements and Technical Concepts for Aviation*.

SALFNER, F. AND WOLTER, K. 2010. Analysis of service availability for time-triggered rejuvenation policies. *Journal of Systems and Software 83,* 9, 1579–1590.

SHERESHEVSKY, M., CROWELL, J., CUKIC, B., GANDIKOTA, V., AND LIU, Y. 2003. Software aging and multifractality of memory resources. In *Dependable Systems and Networks, 2003. Proc. 2003 Int'l. Conf.*

SILVA, L., ALONSO, J., SILVA, P., TORRES, J., AND ANDRZEJAK, A. 2007. Using virtualization to improve software rejuvenation. *Proceedings - 6th IEEE International Symposium on Network Computing and Applications, NCA 2007*, 33–42.

SILVA, L., ALONSO, J., AND TORRES, J. 2009. Using virtualization to improve software rejuvenation. *Computers, IEEE Transactions on 58,* 11.

SILVA, L., MADEIRA, H., AND SILVA, J. 2006. Software aging and rejuvenation in a soap-based server. *Proceedings - Fifth IEEE International Symposium on Network Computing and Applications, NCA 2006 2006*, 56–65.

SOUSA, P., BESSANI, A., CORREIA, M., NEVES, N., AND VERISSIMO, P. 2010. Highly available intrusion-tolerant services with proactive-reactive recovery. *Parallel and Distributed Systems, IEEE Transactions on 21,* 4, 452 –465.

SULLIVAN, M. AND CHILLAREGE, R. 1991. Software Defects and Their Impact on System Availability—A Study of Field Failures in Operating Systems. In *Fault-Tolerant Computing, 1991. FTCS-21. Digest of Papers., Twenty-First International Symposium*. IEEE, 2–9.

SUNDARAM, V., HOMCHAUDHURI, S., GARG, S., KINTALA, C., AND BAGCHI, S. 2007. Improving dependability using shared supplementary memory and opportunistic micro rejuvenation in multi-tasking embedded systems. In *Dependable Computing, 2007. 13th Pacific Rim Int'l. Symp.*

SUZUKI, H., DOHI, T., KAIO, N., AND TRIVEDI, K. 2003. Maximizing interval reliability in operational software system with rejuvenation. In *Software Reliability Engineering, 2003. 14th Int'l. Symp.*

TAI, A., ALKALAI, L., AND CHAU, S. 1999. On-board preventive maintenance: A design-oriented analytic study for long-life applications. *Performance Evaluation 35,* 3, 215–232.

TAI, A., TSO, K., SANDERS, W., AND CHAU, S. 2005. A performability-oriented software rejuvenation framework for distributed applications. In *Dependable Systems and Networks, 2005. Proc. Int'l. Conf.*

TSAI, T., VAIDYANATHAN, K., AND GROSS, K. 2006. Low-overhead run-time memory leak detection and recovery. In *Dependable Computing, 2006. 12th Pacific Rim Int'l. Symp.*

VAIDYANATHAN, K., HARPER, R., HUNTER, S., AND TRIVEDI, K. 2001. Analysis and implementation of software rejuvenation in cluster systems. *Performance Evaluation Review 29,* 1, 62–71.

VAIDYANATHAN, K. AND TRIVEDI, K. 1999. A measurement-based model for estimation of resource exhaustion in operational software systems. In *Software Reliability Engineering, 1999. Proc. 10th Int'l. Symp.*

VAIDYANATHAN, K. AND TRIVEDI, K. 2005. A comprehensive model for software rejuvenation. *Dependable and Secure Computing, IEEE Transactions on 2,* 2.

VALDES, A., ALMGREN, M., CHEUNG, S., DESWARTE, Y., DUTERTRE, B., LEVY, J., SAIDI, H., STAVRIDOU, V., AND URIBE, T. 2003. An architecture for an adaptive intrusion-tolerant server. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 2845*, 158–178.

WANG, D., XIE, W., AND TRIVEDI, K. 2007. Performability analysis of clustered systems with rejuvenation under varying workload. *Performance Evaluation 64,* 3, 247–265.

WANG, Y.-M., HUANG, Y., VO, K.-P., CHUNG, P.-Y., AND KINTALA, C. 1995. Checkpointing and its applications. In *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth Int'l. Symp.*

WEIMER, W. 2006. Exception-handling bugs in java and a language extension to avoid them. *Lecture Notes in Computer Science 4119 LNCS*, 22–41.

WOLTER, K. AND REINECKE, P. 2010. Stochastic models for dependable services. *Electronic Notes in Theoretical Computer Science 261*, 5–21.

XIE, W., HONG, Y., AND TRIVEDI, K. 2004. Software rejuvenation policies for cluster systems under varying workload. In *Dependable Computing, 2004. Proc. 10th IEEE Pacific Rim Int'l. Symp.*

XU, G., BOND, M., QIN, F., AND ROUNTEV, A. 2011. Leakchaser: Helping programmers narrow down causes of memory leaks. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 270–282.

XU, G. AND ROUNTEV, A. 2008a. Precise memory leak detection for java software using container profiling. *Proceedings - International Conference on Software Engineering*, 151–160.

XU, G. AND ROUNTEV, A. 2008b. Precise memory leak detection for java software using container profiling. In *Software Engineering, 2008. ACM/IEEE 30th Int'l. Conf.*

YOSHIMURA, T., YAMADA, H., AND KONO, K. 2011. Can linux be rejuvenated without reboots? In *Software Aging and Rejuvenation (WoSAR), 2011 IEEE Third International Workshop on*. 50 –55.

ZHANG, H., WU, G., CHOW, K., YU, Z., AND XING, X. 2011. Detecting resource leaks through dynamical mining of resource usage patterns. In *Dependable Systems and Networks Workshops (DSN-W), 2011 41st Int'l. Conf.*

ZHAO, J. AND TRIVEDI, K. 2011. Performance modeling of apache web server affected by aging. In *Software Aging and Rejuvenation (WoSAR), 2011 IEEE Third International Workshop on*. 56 –61.

ZHAO, J., TRIVEDI, K., WANG, Y., AND CHEN, X. 2010. Evaluation of software performance affected by aging. In *Software Aging and Rejuvenation (WoSAR), 2010 IEEE Second Int'l. Workshop on*.