Accelerated Life Testing (ALT) Applied to Software Systems

LTB 2021
Load Testing & Benchmarking

ICPE 2021, Rennes, France

April 19, 2021

Kishor Trivedi

Department of Electrical and Computer Engineering
Duke University, Durham, NC, USA
ktrivedi@duke.edu

Duke
UNIVERSITY

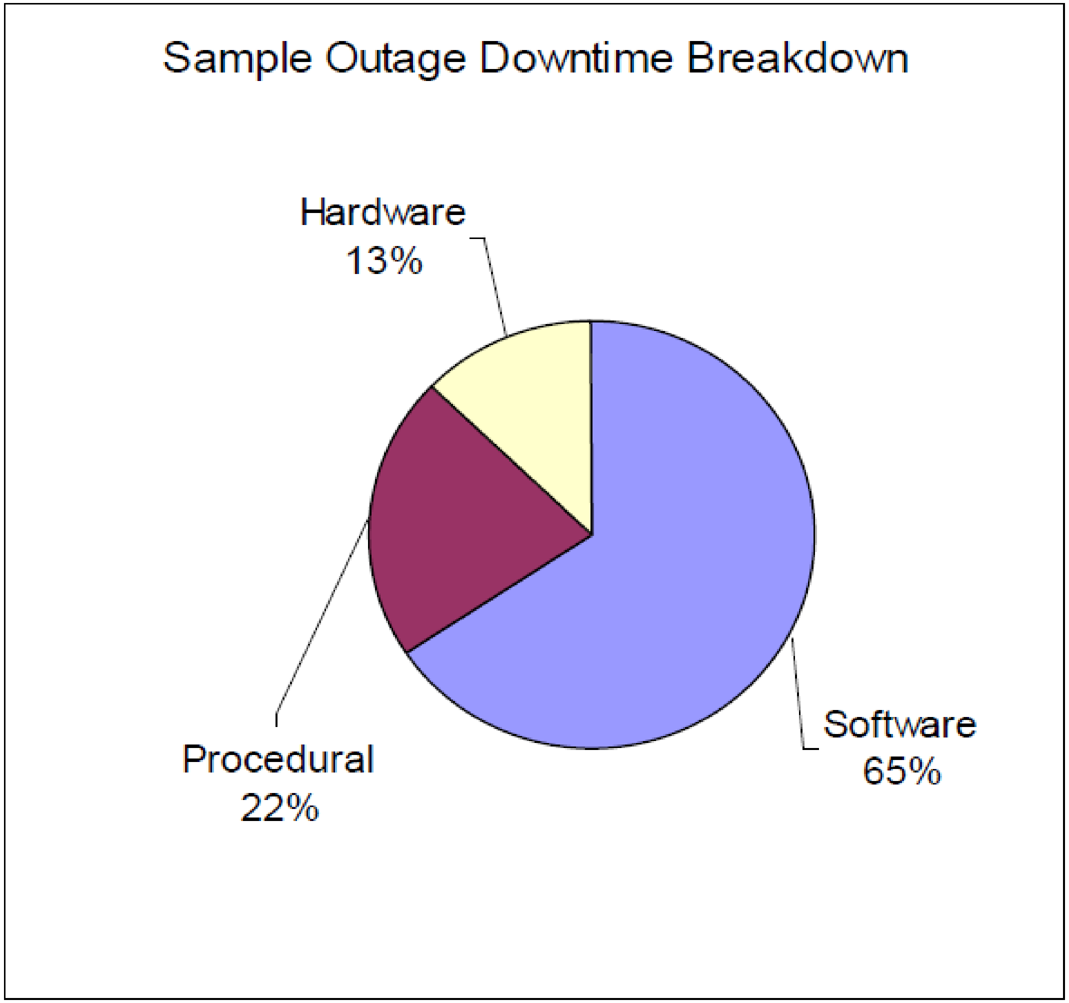
Outline

- Motivation
- Software Aging phenomenon
- Environmental Diversity
- Software Rejuvenation
- Accelerated Life Testing (ALT)
- Applying ALT to Software Systems

Too many failures and downtime in practice

- Hardware Fault Tolerance & Fault management relatively well developed
- System outages more due to software failures

Impact of Software Field Problems



Failures/downtime due to software bugs

amazon.com	Oct. 2012	Amazon Webservices – 6 hours (Memory leak) Amazon EC2 – 2 hours
Google	Sept. 2011	Google Docs service outage – 1 hour (Memory leak)

Failures/downtime due to software bugs

Google	Jul. 2017	Google Cloud Storage service outage (3 hours and 14 min.) - API low-level software bug
Microsoft	Jul. 2017	Jul. 2017 - Microsoft Azure service outage (4 hours) – Load Balancer Software bug

These examples indicate that even the most advanced tech companies are not offering highly reliable software

Fault, Error & Failure

- **Failure** occurs when the delivered service no longer complies with the desired output.
- **Error** is that part of the system state which is liable to lead to subsequent failure.
- **Fault** (or **bug**) is adjudged or hypothesized cause of an error.

Faults (bugs) may cause **errors** that may lead to **failures**

.... **Fault** ⟶ **Error** ⟶ **Failure** ....

Software is a big problem

- Fault avoidance through good software engineering practices does not achieve the goal of low enough fault density for large/complex software systems
- Impossible to fully test and remove faults to ensure software is fault-free
- Deployed software contain many bugs leading to failures during operation
- Yet there are stringent requirements for failure-free operation

Key Challenge:

Reliable software operation given buggy software

➡ **A possible solution: Software Fault Tolerance**

Traditional Software fault tolerance

- In the 1970's researchers pondered over the nature of fault tolerance in software systems
- They thought: it does not make sense to use Identical Copies of software for fault tolerance
- This is different from hardware fault tolerance – where redundant components with identical part numbers are used
- If software fails under some workload, its (identical) copy will also fail on this workload
- So, Software fault tolerance traditionally uses **design diversity**

What is Design diversity?

- Design multiple software versions to the same specifications
  - Use different algorithms,
  - Use different programmers,
  - Use different design/programming languages,
  - Use different development/testing methods
- This is done to minimize the probability of the same bugs in these multiple diverse versions, so as to try to achieve mutual independence from the viewpoint of bugs

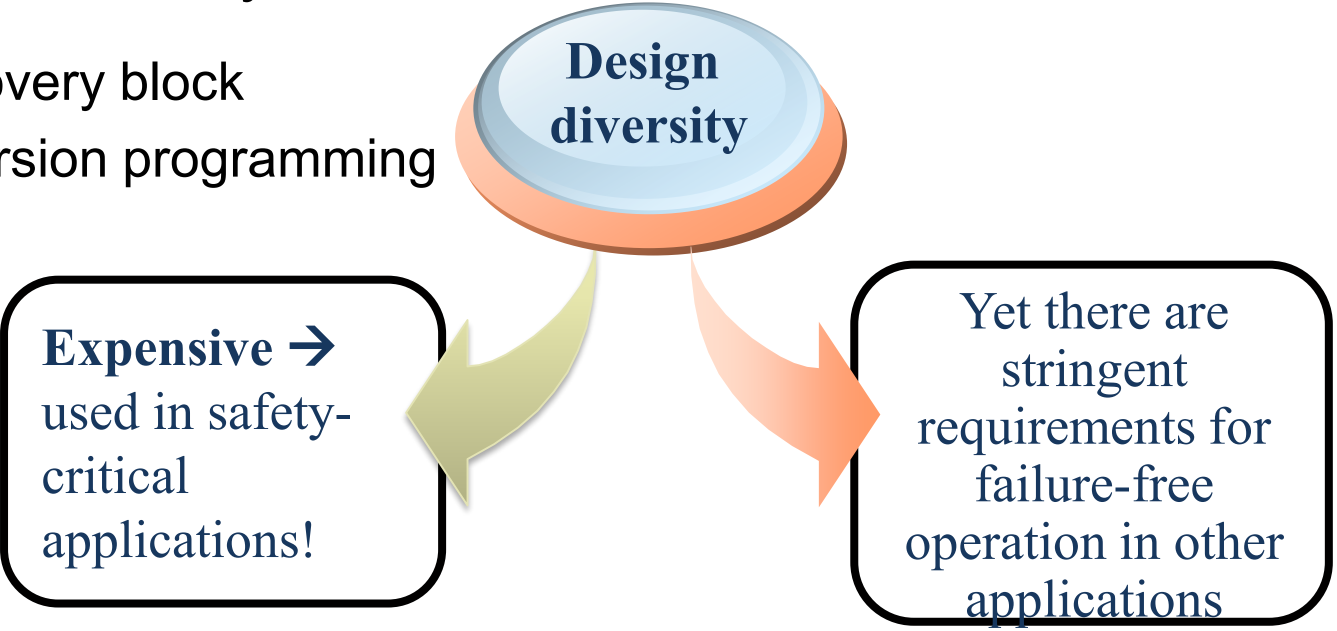
Classical Software Fault Tolerance

- **Employing Design diversity**
  - **Recovery block**
  - **N-version programming**

Software Fault Tolerance

Classical Techniques

- Design diversity
  - Recovery block
  - N-version programming



Design diversity

Expensive → used in safety-critical applications!

Yet there are stringent requirements for failure-free operation in other applications

Need: *Affordable* Software Fault Tolerance

A possible answer: Environmental Diversity

Outline

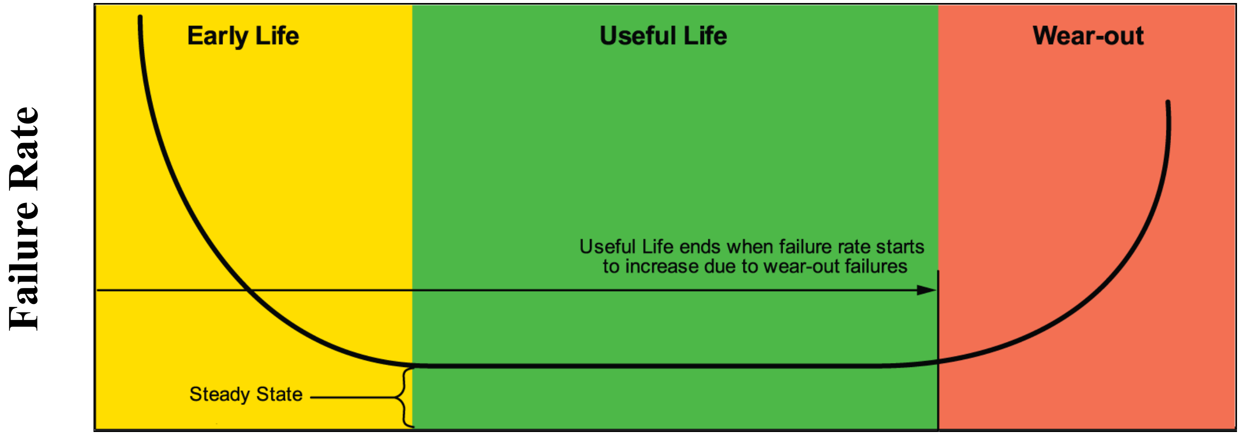
- Motivation
- Software Aging phenomena
- Environmental Diversity
- Software Rejuvenation
- Accelerated Life Testing (ALT)
- Applying ALT to Software Systems

Contradicting Common notion

- It is commonly believed that unlike hardware, software does not age
- But since 1995, we know that software does age

Software aging

- **Software aging** is the name given to a phenomenon empirically observed in many software systems.
  - *as the runtime period of the system or process increases, system slows down and its failure rate tends to increase.*
- In physical (hardware) systems this behavior is well known as the *wear-out phase* illustrated by the bathtub curve.



Software aging

- Unlike hardware, there is no physical/chemical deterioration in software
- Thus, software “appears” to age or it “behaves as if it is aging”
- **What constitutes software aging**?
  - the deterioration of the application process’s internal state
  - it is caused by the cumulative effects of successive error occurrences
  - the notion of **error accumulation** is essential for characterizing the software aging phenomenon

Software aging

- Software applications running for a long period of time exhibit degradation with respect to usage of system resources
  - Increasing failure rate
  - Decreasing performance
  - Eventually leading to system failure
- Due to bugs in applications, OS or software libraries

OS: Windows, Solaris, Linux, Andriod
Applications: Netscape, Internet Explorer
Databases: Oracle, MySQL
Middleware: JES

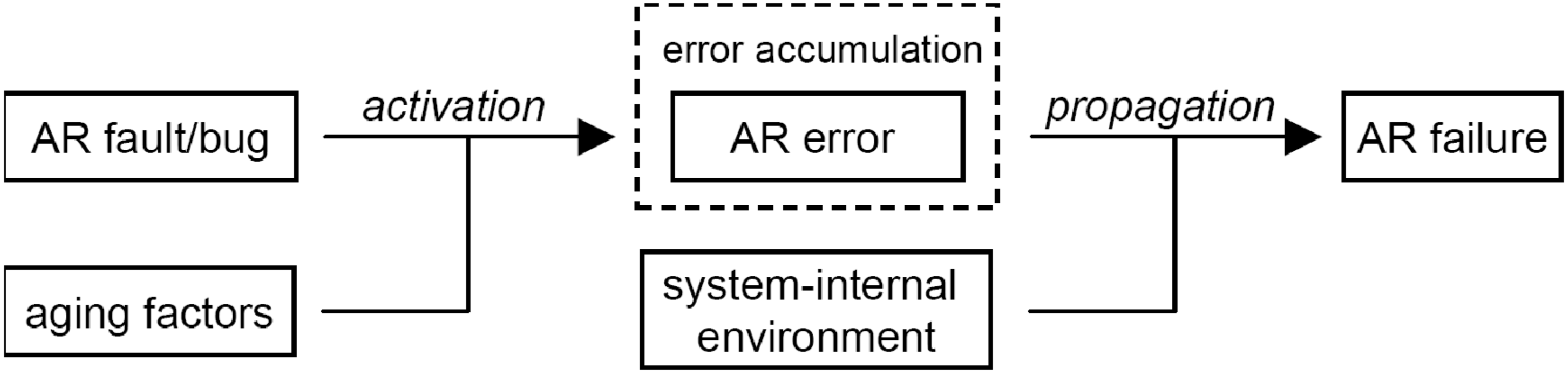
Aging-related Bug (ARB): Definition

- **Aging-related bug :=** A fault that leads to the accumulation of errors either inside the running application or in its system-context environment, resulting in an increased failure rate and/or degraded performance.

Example:

- A bug causing memory leaks in the application
- Note that the aging phenomenon requires a delay between (first) fault activation and failure occurrence – large error latency
- Note also that the software *appears to age* due to such a bug; there is no physical deterioration

Causality Chain for Aging-related (AR) Failures



Aging-related Failure is a failure caused by the accrual of *aging effects* in a system.

---

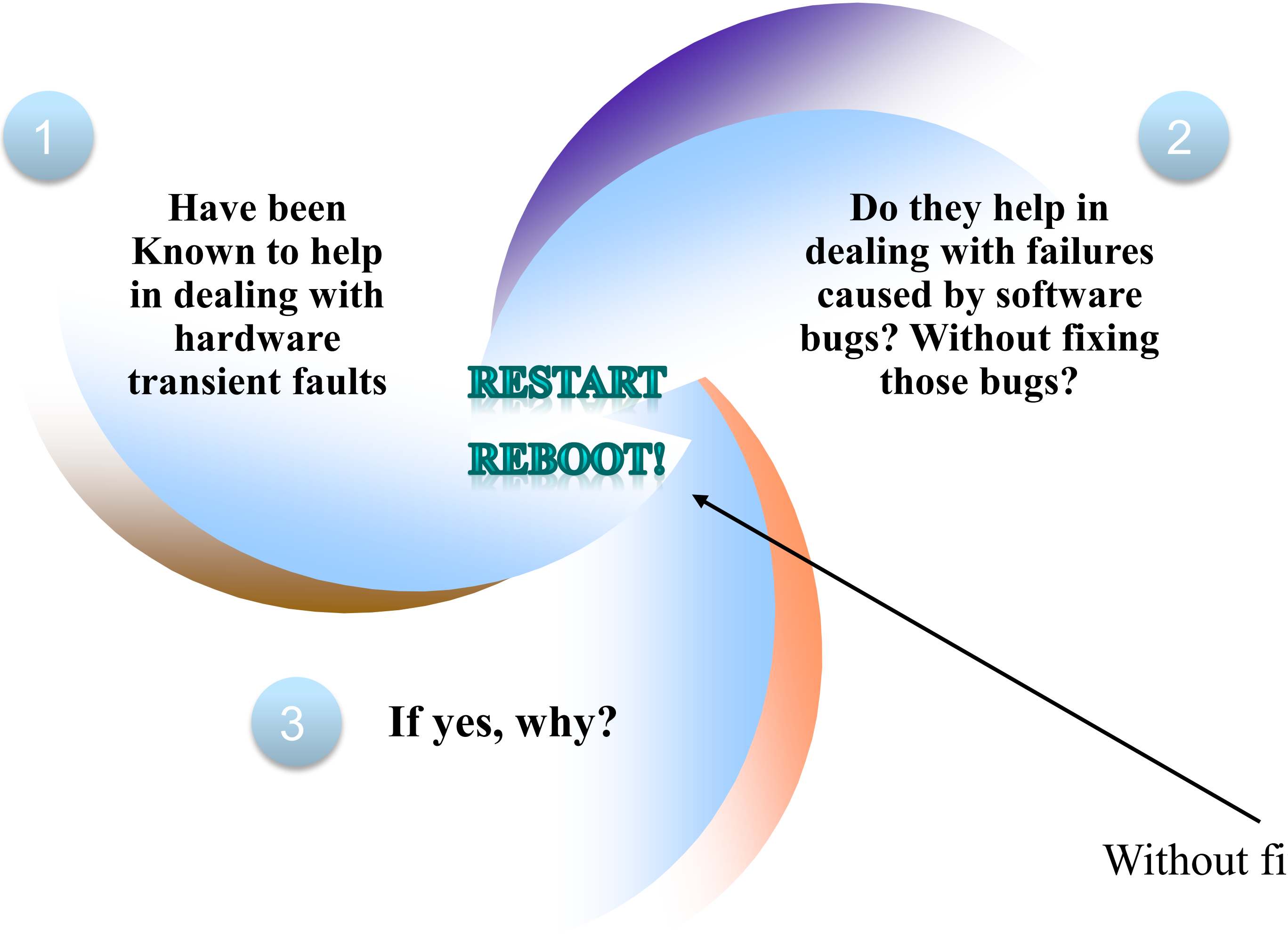
* *M. Grottke, R. Matias, and K. Trivedi, "The fundamentals of software aging," Workshop on Software Aging and Rejuvenation, WoSAR 2008.*

Software aging

- Software aging often leads to the exhaustion of system resources.
  - Memory leaks
  - Unreleased locks
  - Nonterminated threads
  - Shared-memory pool latching
  - Storage fragmentation

- Common recovery technique is to restart process, reboot VM/node, or fail-over to an identical replica (with the same bugs)

Software Fault Tolerance: New Thinking



1

Have been Known to help in dealing with hardware transient faults

RESTART

REBOOT!

2

Do they help in dealing with failures caused by software bugs? Without fixing those bugs?

3

If yes, why?

Without fixing bugs

Outline

- Motivation
- Software Aging phenomena
- Environmental Diversity
- Software Rejuvenation
- Accelerated Life Testing (ALT)
- Applying ALT to Software Systems

Reactive maintenance

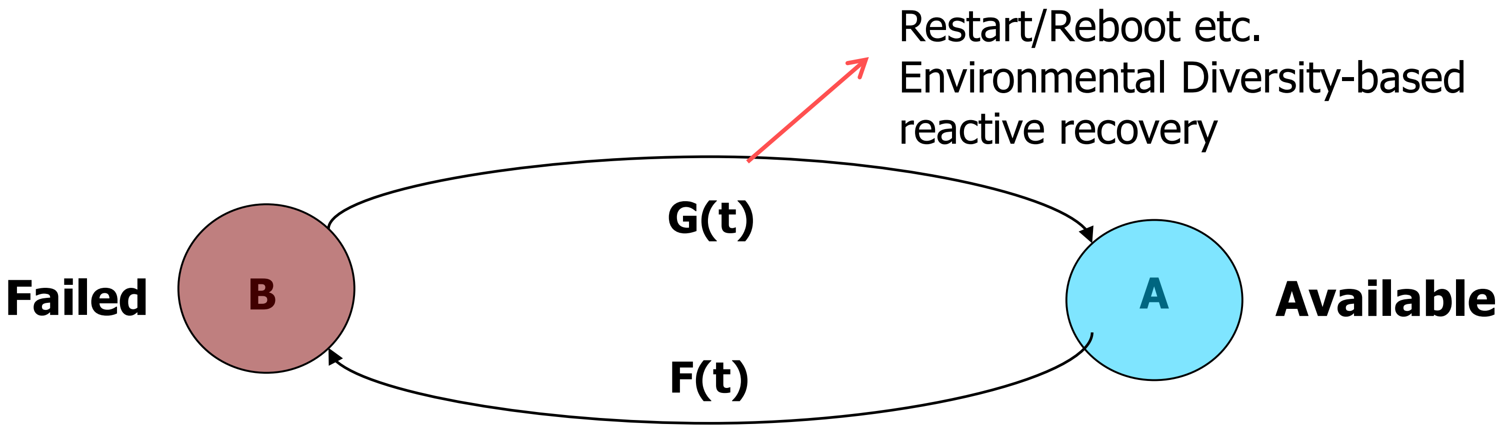
- Applied after a failure occurs
    - Process restart, VM reboot, OS reboot, fail-over to an identical software system, etc.
- The bug that caused the failure, most often not removed
- System is expected to work after restart, reboot, etc.

What is Environmental diversity?

- The environment of the application is understood as
  - OS resources and other applications running concurrently
- The underlying idea of Environmental diversity
  - Restart an application or reboot the node (without fixing the bug) and it most likely works -- Why?
  - These actions counteract the software aging effects
    - Frees up OS resources, Removes error accumulation
  - Thus, the environment where the application is executed has been changed and cleaned enough leading to increased availability of OS resources
- This is fault tolerance via environmental diversity

Semi-Markov Availability Model

State A: the system is up and available
State B: the system is down and undergoing reactive recovery: restart/reboot



Restart/Reboot etc.
Environmental Diversity-based
reactive recovery

$$\text{Steady state availability} = A_{ss} = \frac{MTTF}{MTTF + MTTR}$$

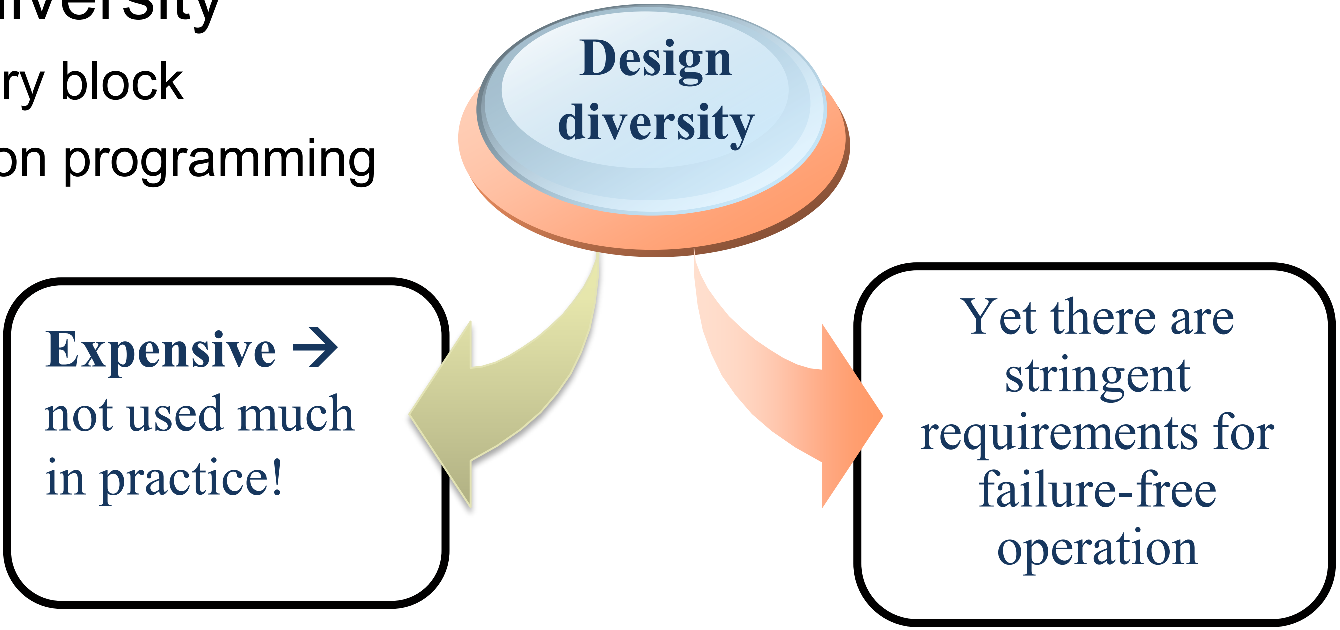
MTTF: mean time to aging-related failure – need to obtain from TTF data
MTTR: mean time to recover after an aging-related failure - experimentally

Implications of aging

- For hardware components or systems that are subject to aging, it is common to carry out preventive maintenance to improve its reliability/availability
- Since software is now known to age as well, preventive maintenance of a software system can be used to improve its reliability/availability
- Preventive maintenance in software systems has an exciting name: software rejuvenation

Software Fault Tolerance: Classical Techniques

- Design diversity
  - Recovery block
  - N-version programming
  - ……



Design diversity

Expensive → not used much in practice!

Yet there are stringent requirements for failure-free operation

Challenge: *Affordable* Software Fault Tolerance

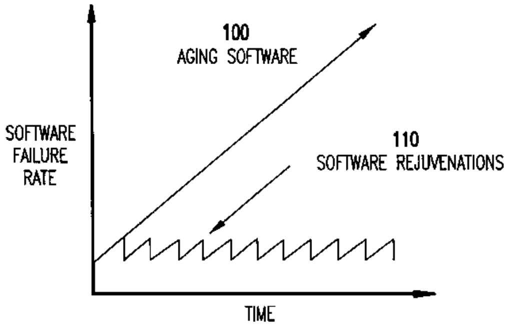
A possible answer: preventive maintenance based on environmental diversity known as software rejuvenation

Outline

- Motivation
- Software Aging phenomena
- Environmental Diversity
- Software Rejuvenation
- Accelerated Life Testing (ALT)
- Applying ALT to Software Systems

Software Rejuvenation

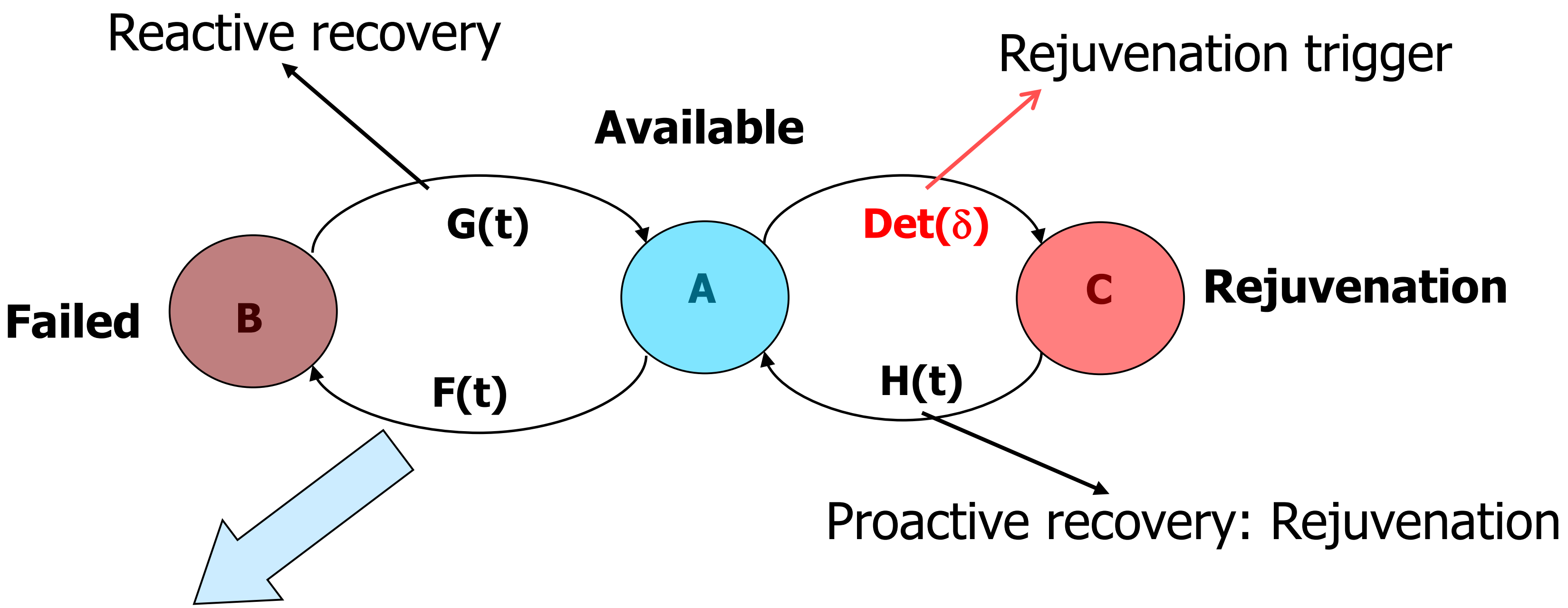
- Proactive technique to counteract software aging
  - To prevent or postpone failures and slow down performance degradation
  - Periodic, proactive rollback to a clean state
- Potential actions
  - Garbage collection
  - Defragmentation
  - Flushing kernel/file tables
  - Application or service restarts
  - VM or VMM or OS reboot
- Rejuvenation of the environment, not of software
- Due to overhead of performing rejuvenation determining optimal schedule is important



Source: IBM

Telco, space systems, defense systems, web services, cloud services, …

Semi-Markov availability model with Software Rejuvenation



Reactive recovery

Rejuvenation trigger

Available

G(t)

Det(δ)

Failed

B

A

C

Rejuvenation

F(t)

H(t)

Proactive recovery: Rejuvenation

Distribution of TTF Needed to determine optimal value of rejuvenation trigger interval

state A: the system is up and available
state C: the state in which software rejuvenation is being carried out
state B: the system is down and under reactive recovery

Optimal Rejuvenation Schedule

- Collect (past) times to failure (TTF) data
- Fit the data to a known (IFR) distribution such as Weibull or Hypo-exponential
- Find optimal times to trigger rejuvenation
  - Need to solve a fixed-point equation
  - These equations are known – several papers and books provide the equations [See Trivedi & Bobbio, 2017 Cambridge University Press -- greenbook]

A Difficulty

- Software aging failures are very difficult to observe experimentally
  - because the accumulation of aging effects usually are random and require long runtimes
- Thus, collecting data for statistically significant predictions of software aging is typically a long-running task and may be unaffordable in many circumstances (ex. highly reliable systems)
- This is an important problem that prevents many experimental and analytical studies of Software Aging & Rejuvenation from using representative parameter values
- Potential solution: ALT (Accelerated Life-Testing)

Outline

- Motivation
- Software Aging phenomena
- Environmental Diversity
- Software Rejuvenation
- Accelerated Life Testing (ALT)
- Applying ALT to Software Systems

Our Approach

- Accelerate the occurrences of Aging-related failures through
  - controlled experiments using the quantitative accelerated life test method (QALT)

- Since QALT was developed for physical/chemical systems
  - we need to adapt it to software systems suffering from aging

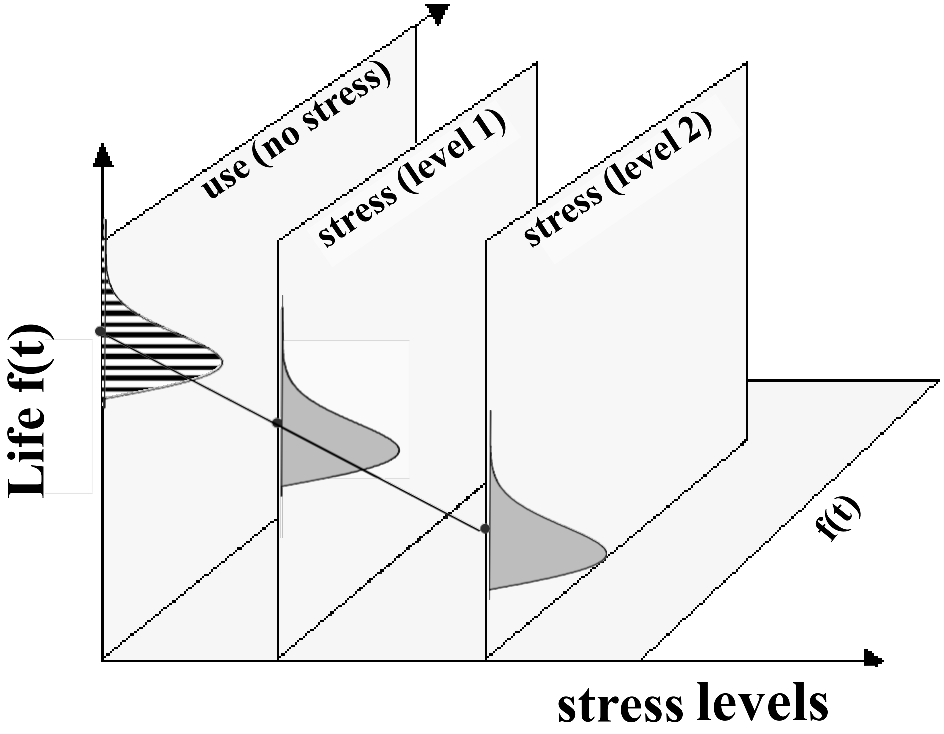
The QALT Method

- Quantitative accelerated life tests are used in several engineering fields to significantly reduce the time needed for testing
- QALT is designed to quantify the life characteristics (e.g., MTTF and Distr. of TTF) of a SUT by applying controlled stresses
- Since the SUT is tested in accelerated mode, the obtained results must be properly adjusted
  - the lifetime data obtained under stress is used to estimate the lifetime distribution of the SUT for its normal use condition

    We use Inverse Power Law (IPL) for this purpose

The QALT Method

Lifetime densities & Stress levels



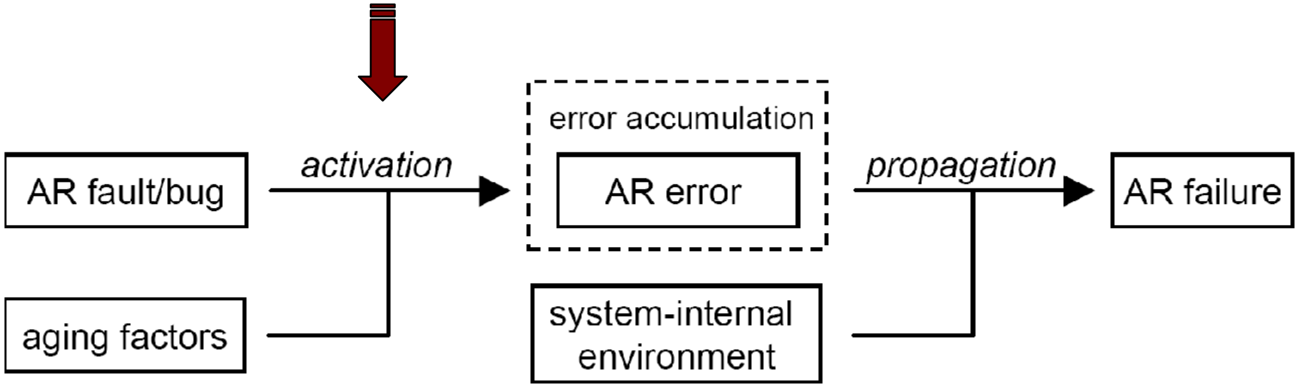
The QALT Method

- Main components of Accelerated Life Test Plan are:

1) Accelerating stress variable and their levels of utilization (load)
2) Stress loading scheme
3) Life-stress relationship model
4) Sample size (n)
5) Allocation proportion ($\propto$)

Acceleration of Aging Effects

The key idea is to accelerate the activation of Aging-related bugs/faults (ARBs or AR faults) by controlling the workload.

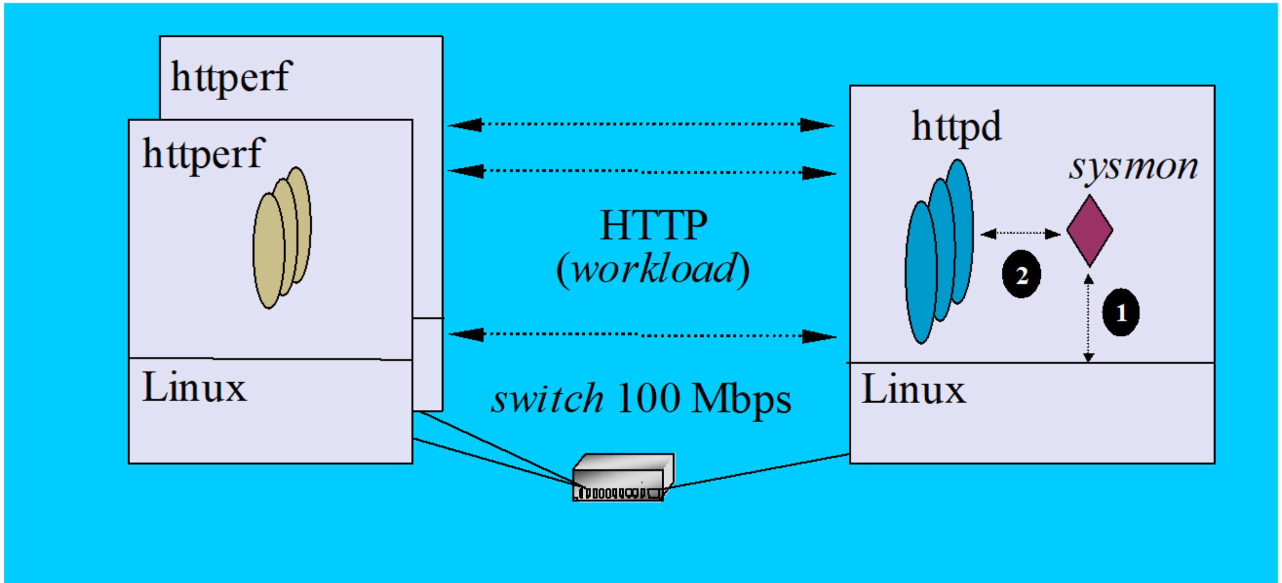


Example 1

System Under Test (SUT)

- The Apache web server (httpd), version 2.0.48, is known to suffer from software aging
  - its main cause of aging is memory leak
- We verified that HTTP requests addressing dynamic pages cause memory leak in this httpd version
  - Experiments we conducted indicated that the “page size” of dynamic requests intensifies the aging effects.

Test Bed



❶ OS *health* monitoring (e.g. swap, buffer-cache, …).

❷ Measurement of *httpds'* size.

QALT Planning

- Main parameters
  - Stress variable (*Aging Factor*)
    - HTTP requests addressing dynamic pages
  - Stress Levels
    - three page sizes
  - Sample size (n)
    - *calculated based on a pilot sample of failure times*
  - Tests allocation proportion ($\propto$)
    - traditional plan (*the same sample size for each stress level*)
    - $\propto = n \div 3$

Example 1

QALT Planning

QALT Experimental Plan

Stress loading		Allocation	
Level	*Page size (kB)*	*Proportion* $\propto$	*Replications* n_{ALT}
Use	200		
S1	400	1/3	7
S2	600	1/3	7
S3	800	1/3	7

QALT Execution

- To better control the exposure of httpd to aging effects, we test only one httpd process.
- The SUT is considered failed when the size (RSS) of the httpd process crosses 100 megabytes.
- The rationale is that in a web server system with 3-GB of RAM running 200 httpd processes
  - if at least 30 processes (15%) grow to around 100 megabytes we have saturation of the server memory.

QALT Execution (cont'd)

- We measure the size (RSS) of the httpd processes after every 100 requests.
- We consider the time to failure not in the units of wall- clock time but
  - the number of bunches of 100 requests processed before the httpd size crosses the specified threshold.
- The wall-clock time may be estimated from the total number of requests until failure and the average request rate.

Example 1

QALT Execution (cont'd)

Time to Failures (TTF)

(in bunches of 100 requests)

TTF (*S1*)	TTF (*S2*)	TTF (*S3*)
84	34	20
86	36	21
88	37	22
93	38	23
95	38	23
95	39	23
97	40	24

QALT Execution (cont'd)

- samples of times to failure (TTF), were fitted to several probability distributions.
- The criteria used to build the best-fit ranking were the log-likelihood function (Lk), and the Pearson's linear correlation coefficient (ρ),
  - whose parameter estimation methods were MLE, and LSE, respectively.
- The Weibull probability distribution showed the best fit for the three accelerated lifetime data sets

QALT Execution (cont'd)

Selection of Lifetime distribution for each stress level

Stress level	Model	GOF		Best-fit Ranking
		Lk	ρ (%)	
S1	Weibull	-20.5086	96.75	1st
	Lognormal	-20.8627	95.98	2nd
	Exponential	-38.5870	-74.38	3rd
S2	Weibull	-13.9211	99.31	1st
	Lognormal	-14.3368	97.60	2nd
	Exponential	-32.3570	-74.53	3rd
S3	Weibull	-11.2420	97.65	1st
	Lognormal	-11.8037	95.33	2nd
	Exponential	-28.7276	-73.96	3rd

Lk = log-likelihood function

ρ = Pearson's linear correlation coefficient

Example 1

QALT Execution (cont’d)

Estimated Weibull Parameters

Stress	Parameter	ML Estimate	CI (90%)	
			Lower	*Upper*
S1	β_1	24.0889	14.3941	40.3134
	η_1	93.3175	90.8149	95.8891
S2	β_2	25.0000	15.2671	40.9377
	η_2	38.2697	37.2791	39.2865
S3	β_3	22.0825	13.3316	36.5775
	η_3	22.8595	22.1925	23.5465

QALT Execution (cont'd)

- The Weibull distribution showed the best fit for the sample of failure times
- We use it in conjunction with the IPL model to create our life-stress relationship model.

2-parameter Weibull (*pdf*)

$$f(t) = \frac{\beta}{\eta}\left(\frac{t}{\eta}\right)^{\beta-1} e^{-\left(\frac{t}{\eta}\right)^{\beta}}$$

IPL

$$\eta(s) = \frac{1}{v \cdot s^{w}}$$

$$f(t,s) = \beta v s^{w} (v s^{w} t)^{\beta-1} e^{-(v s^{w} t)^{\beta}}$$

IPL-Lognormal (*pdf*)

η, β are respectively the scale and shape parameters.
s is the stress level value.
v is one of the IPL model parameters to be determined, ($v > 0$).
w is another model parameter to be determined.

Example 1

QALT Execution (cont'd)

- The MTTF equation can be directly derived from the IPL-Weibull model.
- We use this equation to estimate the mean time to failure of the SUT at a specific use rate.

$$MTTF(s) = \frac{1}{vs^{w}} \cdot \Gamma\left(\frac{1}{\beta} + 1\right)$$

Estimated IPL-Weibull Parameters

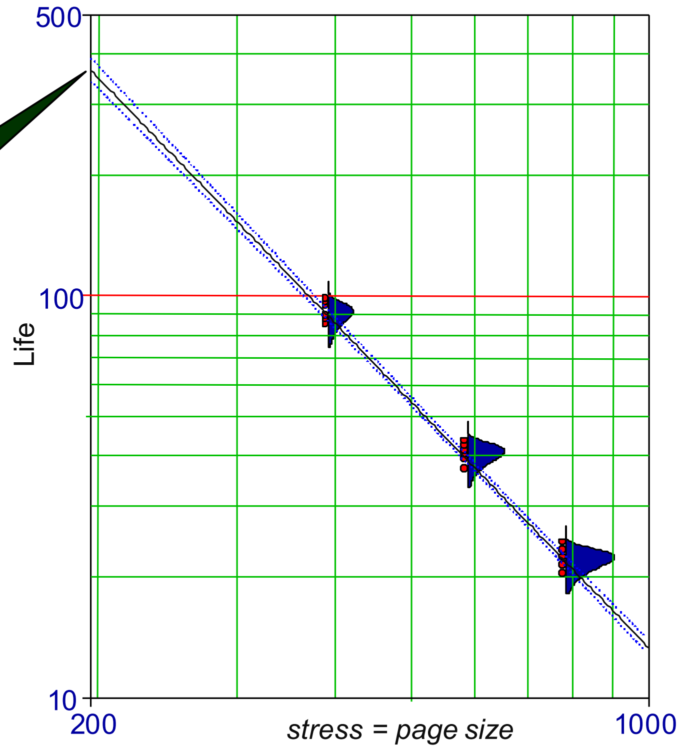
Estimated IPL-Weibull Parameters

Parameter	ML Estimate	CI (90%)	
		Lower	*Upper*
v	5.7869E-8	3.7257E-8	8.9885E-8
w	2.0340	1.9646	2.1034
β	18.9434	13.5270	26.5286

Example 1

QALT Execution (cont'd)

- Using the estimated parameters for the IPL-Weibull model, we calculate the mean life of the SUT for its use condition.



Estimated MTTF= 365.48
(at use condition)

IPL-Weibull model fitted to the accelerated lifetime dataset

QALT Execution (cont’d)

- In order to evaluate the accuracy of the estimates, we ran an experiment using the page size equals to 200 kB
- This setup refers to the SUT operating in its use condition (w/o accelerating stress)

Observed and Estimated MTTF

	MTTF	**CI (90%)**	
		Lower	***Upper***
ML Estimate	365.48	337.92	395.28
Observed	343.57		

QALT Execution (cont'd)

- We evaluate the proposed method with respect to the reduction of experimentation time.
- First, we calculate the total time spent to execute all tests (replications) for all stress levels:

$$te = \sum_{j=1}^{m} \sum_{i=1}^{r_j} TTF_{i_j}$$

where

te is the total experiment time,
m is the number of stress levels in the experimental plan,
r_j is the number of test replications executed in the *j*th level of stress,
TTF_{i_j} is the *i*th observed failure time in the *j*th level of stress.

QALT Execution (cont'd)

- Based on the sample of failure times from the SUT, the value of ***te*** is 105,600 requests.
- The observed mean time to aging-related failures for the SUT at use condition is 343.57 (or 34,357 requests).
  - considering 21 replications, as used in our approach, we have a total experimentation time of 721,497 requests.
- In summary:
  - 21 tests (w/o acceleration) = 721,497 requests
  - 21 tests (w/ acceleration) = 105,600 requests
- A reduction of the experimental time by a factor close to 7.

Two Other Examples of using ALT for Software Systems

- Software rejuvenation scheduling using accelerated life testing, J. Zhao, Y. Jin, K. Trivedi, R. Matias Jr., and Y. Wang, ACM Journal on Emerging Technologies in Computing Systems, 2014.

- Stress Testing With Influencing Factors to Accelerate Data Race Software Failures. Qiu, Zheng, Trivedi, et al. IEEE Transactions on Reliability, 2019.

Conclusion

- Research in software aging has been limited to measure aging effects without observing failures
  - mainly because the long period of time required to observe aging-related failures
- We have demonstrated the feasibility of applying QALT to reduce the time to failures in systems suffering from software aging
  - we use the concept of “aging factors” as “stress variables”, which enables the use of QALT to software systems

Selected References

- **Handbook on Software Aging and rejuvenation,** T. Dohi, K. Trivedi & A. Avritzer (eds.), World scientific, 2020
- **Reliability and Availability: Modeling, Analysis, Applications,** K. Trivedi & A. Bobbio, Cambridge University Press, 2017
- **Accelerated testing: statistical method, test plans, and data analysis,** B. N. Nelson, New Jersey: Wiley, 2004.
- An Experimental Study on Software Aging and Rejuvenation in Web Servers. R. Matias, P. J. Freitas Filho, COMPSAC, 2006.
- Using Accelerated Life Tests to Estimate Time to Software Aging Failure, R. Matias, K. Trivedi, P. Maciel , ISSRE, 2010.
- Accelerated Degradation Tests Applied to Software Aging Experiments, R. Matias, K. Trivedi, P. Filho and P. Barbetta, IEEE Trans. on Reliability, 2010.
- Software rejuvenation scheduling using accelerated life testing, J. Zhao, Y. Jin, K. Trivedi, R. Matias Jr., and Y. Wang, ACM Journal on Emerging Technologies in Computing Systems, 2014.
- Understanding the impacts of influencing factors on time to a datarace software failures, K. Qiu, Z. Zheng, K.S. Trivedi, B. Yin, ISSRE 2017
- Stress Testing With Influencing Factors to Accelerate Data Race Software Failures. Qiu, Zheng, Trivedi, et al. IEEE Transactions on Reliability, 2019.