Stress Testing With Influencing Factors to Accelerate Data Race Software Failures

Kun Qiu, Zheng Zheng, *Se i Me be IEEE*, Kishor S. Trivedi, *Life Fe , IEEE*, and Beibei Yin

Abstract

possibilities. However, even with the help of the developed tools, the extensive application of these techniques on large systems is hindered by issues with the time overhead. Therefore, how to handle potential data race failures and improve systems' reliability/availability remains an important research topic.

Fault mitigation techniques, such as the recovery methods [9], are always introduced in real safety-critical systems to make the potential failure less severe and to improve the systems' reliability/availability. Due to the elusive behavior of data race software failures, the practices of fault mitigation and quantitative reliability analysis tend to be proceeded by distinguishing them from deterministic and easy-to-reproduce cases. Grottke *e a*. divided bugs into bohrbugs, which can be certainly reproduced, and mandelbugs, which have nondeterministic manifestation behaviors, and proposed distinguished fault mitigation methods [9]–[12]. As a typical example of mandelbug, data race bugs could be mitigated by the recovery operation schemes, such as reconfiguring the environmental conditions and retrying the fault inputs. Stochastic process models, such as continuous Markov chain (CTMC), semi-Markov process (SMP), Petri Net, etc., are used to assess the quantitative reliability/availability of a system considering potential data race failures and mitigation techniques since these models can easily describe complex behaviors [12], [13].

Solving and optimizing these stochastic process models highly rely on the time to failure (TTF) or mean TTF (MTTF) metric to be known in the first place [12], [14], [15]. TTF (or MTTF) denotes the (average) time, in which an application can normally run for its given workloads. However, the task of estimating the TTF/MTTF is impractical for data race software failures because the failure observation time is too long to collect sufficient samples. Therefore, how to accelerate the data race failure and reduce the TTF/MTTF estimation time is an important topic to be studied.

To the state of the art, the stress testing method, such as accelerating life testing (ALT), has been successfully used for reducing the TTF/MTTF estimation time when a system suffering from aging-related bugs (ARBs) [16]–[18]. The stress testing method uses the TTF/MTTF data collected under stressed conditions to predict the one under target conditions according to certain models. Facing the same problem, the feasibility of using the stress testing method to deal with failures caused by data race bugs has not been systematically studied. Two critical issues need to be explored to facilitate the stress testing method as follows.

1) How to accelerate the process of detecting data race software failures? We need to determine the influencing factors as the stress condition to reduce data race failures' manifestation time and reduce the essential time.
2) How to effectively and efficiently estimate the TTF or MTTF caused by data race? We need to determine a model that can relate the TTF/MTTF under different stress conditions.

In this paper, we present an empirical study of using stress testing with influencing factors to reduce the TTF/MTTF estimation time. Stress testing is commonly conducted to flush out data race conditions by ceaselessly performing certain workloads for a long period of time [5]–[7]. However, according to existing research [5] and our testing experiences, we find that the purely stress testing practices are neither efficient nor effective because most of schedule possibilities are still not covered after iterating workloads for many loops. Aiming to overcome deficiencies of stress testing, we additionally control programs' external executing environments, known as influencing factors, to increase the possibilities of encountering data race preconditions. Based on the software failure mechanism of data races, we select three influencing factors that could influence programs' internal execution processes and then increase the risk of triggering the data race bugs.

Furthermore, we empirically study the relationship models between TTF/MTTF and the influencing factors. The relationship model is a quantified strength function that relates the response variable, i.e., TTF/MTTF, with the explanatory variable, i.e., the influencing factor [19], [20]. With this relationship model, the stressed conditions' TTF/MTTF can be extrapolated back to predict other operational (or nonstressed) condition values. Considering that the relationship model plays a pivotal role in solving the proposed issues, we employ a regression analysis method to delve into its formula in this paper.

A methodology including experiments and regression analysis is presented to study the influences and the relationship model for the proposed influencing factors. This methodology combines the design of experiments (DOE) [21] and the stress testing analysis [19], [20]. We conduct the methodology on six different desktop/server applications, which suffer from real data race failures.

This paper reveals seven interesting findings that provide useful guidelines for software data race testing, software reliability modeling, and evaluating, as shown in the following.

1) *Fi di g #1:* All three proposed influencing factors can be used in the stress testing to reduce the expected time to data race failure for the experimental applications. For the applications we tested, we could save the reproducing time for at least 15.9 times. For example, we did not manifest a failure hidden in an application known as *Pb i* when retrying the failure workload 50 000 times but did observe the failure with an average of 628.2 reattempts.
2) *Fi di g #2:* The stress testing condition, ⟨*High* memory limitation, *High* concurrency level, *M i e* parallel level⟩, is the optimized condition to minimize the time to data race failure.[1] *Fi di g #1 a d #2* indicate a lightweight probabilistic stress testing method to detect and reproduce race conditions during development phrase.
3) *Fi di g #3:* The MTTF varies with different influencing factor settings. In addition, the variations caused by different factors for the same data race suffering application are different. This finding gives the formal statistical evidence that the *e i ε e a di e ε i* mitigation method proposed in [11] and [12] can be applied for data race failures. *E i ε e a di e ε i* approach expects to im-

[1]The terms in this finding are defined in Section II.

prove systems' availability by reconfiguring applications' environmental conditions and retrying the failure inputs.

4) *Fi di g #4: P e de* best fits the relationship between MTTF and the influencing factor explanatory variable.[2]
5) *Fi di g #5: Weib di ib i de* best fits the TTF distribution model of data race failures.[3]
6) *Fi di g #6: P e Weib di ib i de* best fits the relationship model between TTF and the influencing factor explanatory variable.
7) *Fi di g #7:* The MTTF and TTF can be accurately estimated by using the proposed influencing factors as explanatory variables. *Fi di g #4–#7* provide the TTF/MTTF distribution and estimation information, which are necessary for performing quantitative reliability/availability analysis and designing fault mitigation techniques.

This paper is organized as follows. Section II presents the influencing factors for data race software failures. Section III presents the experimental methodology employed in this paper. Section IV organizes the experimental data analysis results and seven findings into the answers to three research questions. Sections V lists potential threats to the validity of this paper. Section VI shows the related work. Finally, conclusions and future work are given in Section VII.

II. Influencing Factors

In this section, we describe the influencing factors and discuss how the influencing factors affect the occurrence of data race failures. The factors are expected to accelerate data race failures by impacting the failure manifestation process. Recall that a data race occurs when there are memory accesses in a program that satisfy all the following conditions:

1) operating on the same memory location;
2) performing concurrently or in parallel;
3) writing (or modifying) operations;
4) not protected by synchronization mechanisms.

Therefore, the main idea of influencing a program's internal execution and accelerating the occurrence of data race is to increase the risk of breaking up unprotected synchronization of writing (or modifying) operations on the shared locations. Based on this motivation, we select three influencing factors as follows.

1) Memory limitation, which is the maximum resident set size (*RSS*) of physical memory available for the running program.
2) Concurrency level, which is the number of concurrent visiting clients/users of the program.
3) Parallel level, which is the number of processors available for the program.

In the following, we will explain how these three influencing factors affect the internal execution behavior of a program and increase the probability of triggering data race conditions.

[2]The mathematical format of *P e de* is defined in Section III-B3.

[3]The *df* of *Weib di ib i* and *P e Weib di ib i* are given in Section IV-C.

A. Me Li i a i

The memory limitation controls the maximum physical memory set size a to-be-tested application can access. The physical memory limitation can increase the frequency of thread context switching [22] and, then, indirectly increase the threads' risk of being interrupted from the synchronization parts. In a thread context switch, the accessing of a processor is switched from one thread to another. According to the principle of an operating system (OS) [22], [23], if a processor tries to read from a virtual memory address that is not currently mapped to a RAM address, a page fault occurs; then the running thread is pre-empted and suspended to the waiting state, during which its required data is mapped into the RAM from the disk by the OS. Simultaneously, another thread in the ready state starts to run. Thus, if the available physical memory is limited, the frequency of page fault is increased, thereby increasing the interaction probability among concurrent threads, which increases the probability of encountering unprotected synchronization operations. Therefore, we select memory limitation as the first influencing factor.

Memory limitations are an indirect and lightweight method to control the frequency of context switches. The alternative method to this aim is inserting instruments in threads/processes and scheduling them deliberately. Such intrusive approach could slow down the system and faces space explosion problems, hence is hard to be employed for large scaled applications. However, too harsh memory limitation stress could lead to the thrashing problem [24], in which pages kept on swapping in and out of RAM alternatively. Thrashing could lead to longer workload execution time and fail to reduce the testing time. Therefore, we must reserve a certain margin so that thrashing has no significant impact on the experimental time.

B. C c e c Le e

The concurrency level refers to the number of different kernel threads or processes to be executed out-of-order or in partial order based on which user-level tasks are mapped. According to our prior experimental experience and literature studies [25], [26], data race software failures occur more frequently with higher concurrent workloads. Higher concurrency level workloads increase the complexity and intensity of context switching for racing the shared resources. Therefore, they could increase the possibility of hitting certain thread orders that can break the synchronization and thus trigger a data race failure. Due to the aforementioned reasons, the concurrency level is selected as the second influencing factor in this paper.

C. Pa a e Le e

The parallel level controls the number of processors that can be accessed by the testing application. A data race occurs when two or more threads are executing modifications on a shared memory location without synchronization. For a single-processor-equipped system, the threads are scheduled to alternatively access the processor, and the shared memory location is accessed by only one threat at a time. While for a multiple-processor case, the spawned threads can run on

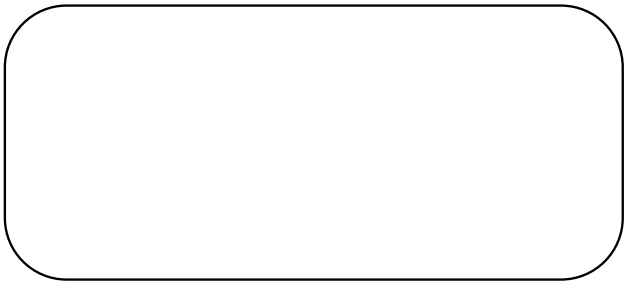


TABLE I
EXPERIMENTAL PLAN FOR THE FIRST STAGE

Index	Memory	Concurrency	Parallel	# of Replicates
EXP1	Low	Low	Single	5
EXP2				

TABLE II
EXPERIMENTAL APPLICATIONS AND EXPERIMENTAL SETTINGS FOR THE FIRST STAGE

TABLE III
EXPERIMENTAL SETTINGS FOR THE SECOND STAGE

App.	Memory limitation	Concurrency level	# of Replicates
AirNine	1,000K, 2,000K, 4,000K, 6,000K, 8,000K,10,000K.	2, 5, 10, 15, 20, 25.	15

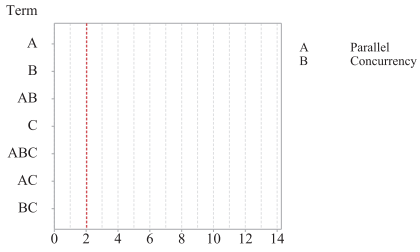
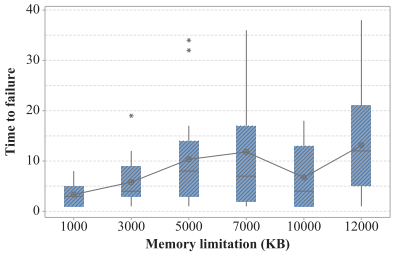
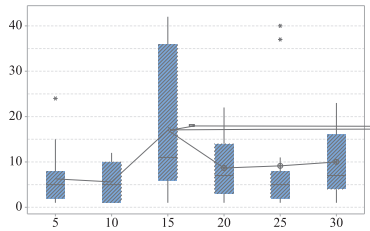

Term
A
B
AB
C
ABC
AC
BC
0
2
4
6
8
10
12
14
A Parallel
B Concurrency

TABLE IV
MEAN VALUES OF RESPONSE VARIABLE IN THE FIRST STAGE OF EXPERIMENTS

App.	***EXP1***	***EXP2***	***EXP3***	***EXP4***	***EXP5***	***EXP6***	***EXP7***	***EXP8***
Airline	*(censored)*	13.8	4838.0					


Time to failure
40
30
20
10
0
1000
3000
5000
7000
10000
12000
Memory limitation (KB)

(a)


40
30
20
10
0
5
10
15
20
25
30

[redacted]

App.	Inf.

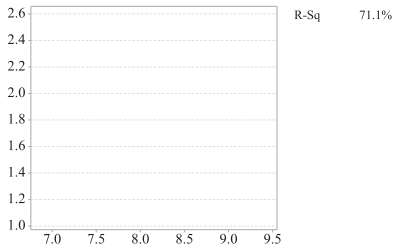

2.6
2.4
2.2
2.0
1.8
1.6
1.4
1.2
1.0
7.0
7.5
8.0
8.5
9.0
9.5
R-Sq
71.1%

TABLE VII
LOCAL DISTRIBUTION ANALYSIS RESULTS

App.	Inf.	Level	*Weibull*	*LogNormal*	*Exponential*	*Normal*
Airline	M. L.	L1	**1.357**	1.507	1.631	**1.471**
		L2	**1.312**	**1.115**	1.685	1.929
		L3				
		L4				

TABLE VIII
TTF REGRESSION ANALYSIS RESULTS

App.	Inf.	Param.	Estimator	Std. Error	95% *Normal* CI	
					Lower-B	Upper-B
		$log(\beta_0)$				
Airline	M. L.					

shows the -standardized residuals plot with respect to the standardized *S a e E e e Va e* distribution. Since all of the -standardized residuals fall into the 95% CI lines, we can conclude that *E e ia Weib* is accurate.

The MTTF for *M i a* by its CL, concurrency level variable, is obtained by (4), and is given as follows:

$$\mathrm{MTTF}_{M\ i\ a}(\mathrm{CL}) = 1.019113 \times \ ^{12.0311-0.14866\times \mathrm{CL}}. \quad (20)$$

Di c i : Wha a e he e a i be ee he MTTF f -a b ai ed i Sec i IV-B, a d h e i hi ec i ? Two types of MTTF formulas are obtained in this paper. In Section IV-B, the MTTF formula (see Table VI) is estimated with the mean values of the response variable, which require a large number of samples to support the correctness of the statistical result considering the central limit theorem's prerequisite conditions. Comparatively, in Section IV-C, the MTTF formula is calculated based on the TTF's *df*, which requires the prior knowledge of the TTF's distribution model. Although the MTTF formulas are obtained in two ways, we have the following two observations. First, the corresponding applications' MTTF mathematical forms are all the same. For example, for *Ai i e* application, its MTTFs are both in the form of *P e de* . Second, the differences between the corresponding intercept parameters and the corresponding slope parameters for the six applications are quite negligible.

On the basis of above two observations, we obtain the following *i ica i* . If the number of samples is sufficiently large, the MTTF can be estimated by the mean values of the response variable as shown in Section IV-B, which is easier to calculate. If the samples are hard to be collected to support a statistical analysis, the MTTF could be more accurately estimated by taking advantage of the prior knowledge of the TTF's distribution model, which is a *Weib* distribution as presented in *Fi di g #4*.

V. Threats to Validity

In this section, we will discuss the threats to validity from two aspects.

A. E e a Va idi

Although we perform our empirical studies on six different applications suffering from real data race failures, we still cannot claim that the effectiveness of our findings can be generalized to all software. However, to ensure the generalizability, we select the applications from four different repositories, namely *SIR* [34], *Radbe ch* [36], *SCTbe ch a k* [37], and *M SQL* bug website [35]. The applications cover different working scenarios, e.g., desk applications (*Pb i* and *M i a*), server programs (*M SQL* and *Me cached*), and business software (*Ai i e* and *Acc*). Additionally, our applications cover three common programming languages, C (*Me cached*), C++ (*Pb i* , *M SQL*, and *M i a*), and Java (*Ai i e* and *Acc*). Considering these three languages have different primitive mechanisms to accomplish synchronization for concurrent threads, we believe that the findings obtained from the selected applications can be applied extensively for a wide range of situations.

The applications we tested are deployed on a single machine; thereby we cannot claim that all findings are suitable for applications deployed on HPC systems [28], such as on clouds. The HPC applications deal with huge sets of data and complex algorithm on hundreds of computers. However, the concurrency level and parallel level should also be used for the accelerating factors for HPC applications. Because high number of clients should increase the complexity of scheduling results; High number of deployed machines should increase the probability of modifying the unprotected shared resources simultaneously. Therefore, taking into account the synchronization mechanism similarities between the desktop/server and HPC applications, we may be able to generalize part of our findings, for example, *Fi di g #1, #2, #3, a d #5*, to HPC applications.

B. I e a Va idi

Since the time to a data race failure relies strongly on the schedulers' scheduling results, which can be impacted by other running processes spawned by the OS or other concurrent running applications, the current testing environments are another threat. To alleviate this problem, we perform three operations in our experiments to minimize the influences. First, we manually reboot the machine after the samples are collected from different stress settings. Second, we add a break time between two adjacent workload iterations under the same stress setting. Because a process's scheduling priority can be affected by its preceding ones in the queue, we use the time break to minimize this impact. Third, we will not perform any other unnecessary tasks on the testing system during the experiment.

Using the virtual machines could contribute to another threat. The side effects of using a virtual machine are the various unknown configurations different from real machines. For example, compared to a virtual machine, a real machine always has faster I/O bandwidth for the same files and the same websites. Those hidden differences could affect the scheduling results and generate different TTF observations. However, such unknown configurations cannot affect the influencing factors and their relationship with TTF/MTTF. The influencing factors are selected based on the motivation that they have physical impacts on OS schedulers. The relationship model is a quantitative description of those impacts. The impacts, which are presented in Section II, are independent of hardware machines. Therefore, our findings can be applied to both virtual machine and real machine scenarios. In addition, virtual machines are commonly used as experimental platforms when studying reliability characteristics, such as failure rate, for software failures caused by data race bugs in previous studies, such as in [26] and [37].

VI. Related Work

In this section, we review and associate our findings with related work.

A. Da a Race Fa Mi iga i Tech i e

Various approaches have been proposed to improve the efficiency and effectiveness of testing for flushing out data race bugs. The studies [5]–[7], [42] concentrate on detecting data

race bugs by explicitly controlling the OS's scheduler, aiming at covering all the possible schedule possibilities. A serious limitation, despite considerable effort spent to overcome, is that they are not suitable for large-scale systems because of their complexity explosion and time overhead issues. The approach of using stress testing by controlling programs' external environmental conditions is inspired by research works [26], [43], and [44], in which the environmental-dependent bugs are discussed. Besides, the approach employed in this paper, which is controlling programs' environmental conditions, can effectively alleviate the complexity and time overhead issues for stress testing methods, compared with controlling OS's scheduler. In this paper, *Fi di g #1* confirms the effectiveness of using environmental factors to increase the occurrence of data races. *Fi di g #2* presents an optimal combination of environmental conditions, which can improve the efficiency of debugging and reproducing race conditions.

VII. Conclusion

In this paper we presented a study of using the stress testing with influencing factors to accelerate the manifestation process of data race software failures. In addition, it explored how to estimate the time to data race failure or mean time to data race failure for a system. Three influencing factors, namely memory limitation, concurrency level, and parallel level, were proposed to accelerate the failure process caused by a data race bug. The statistical analysis results of experiments confirmed the validity of the factors in terms of accelerating data race software failures. Therefore, the proposed influencing factors can be used to optimize the practices of software testing/debugging for data race failures.

Furthermore, the regression analysis showed that TTF/MTTF can be accurately estimated by treating the influencing factors as explanatory variables. Moreover, the best-fitting relationship models between TTF/MTTF and the proposed influencing factors are *P e Weib di ib i de* and *P e de*. These findings can facilitate the reliability evaluation for software systems suffering from data race failures.

In the future work, we plan to systematically study the influencing factors and stress testing approaches for software failures caused by concurrency bugs, including both data race and deadlock. We will analyze the similarities and differences of factors affecting different types of failures and their relationships with TTF or MTTF.

References

[1] M. Grottke, A. P. Nikora, and K. S. Trivedi, "An empirical investigation of fault types in space mission system software," in *P c. IEEE/IFIP I . C f. De e dab e S . Ne .*, 2010, pp. 447–456.

[2] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan, "Efficient and precise datarace detection for multithreaded object-oriented programs," *ACM SIGPLAN N ice*, vol. 37, no. 5, pp. 258–269, 2002.

[3] P. Fonseca, C. Li, V. Singhal, and R. Rodrigues, "A study of the internal and external effects of concurrency bugs," in *P c. IEEE/IFIP I . C f. De e dab e S . Ne .*, 2010, pp. 221–230.

[4] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Ta . C . S .*, vol. 15, no. 4, pp. 391–411, 1997.

[5] S. Park, S. Lu, and Y. Zhou, "CTrigger: Exposing atomicity violation bugs from their hiding places," *ACM SIGARCH C . A chi . Ne*, vol. 37, no. 1, pp. 25–36, 2009.

[6] M. Musuvathi, S. Qadeer, T. Ball, M. Musuvathi, S. Qadeer, and T. Ball, "Chess: A systematic testing tool for concurrent software," Microsoft Res., Microsoft Corporation, Redmond, WA, USA, Tech. Rep. MSR-TR-2007–149, 2007.

[7] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam, "Maple: A coverage-driven testing tool for multithreaded programs," in *P c. ACM SIGPLAN N ice*, vol. 47, no. 10, pp. 485–502, 2012.

[8] B. Wester, D. Devecsery, P. M. Chen, J. Flinn, and S. Narayanasamy, "Parallelizing data race detection," *ACM SIGARCH C . A chi . Ne*, vol. 41, no. 1, pp. 27–38, 2013.

[9] J. Alonso, M. Grottke, A. P. Nikora, and K. S. Trivedi, "An empirical investigation of fault repairs and mitigations in space mission system software," in *P c. 43 d A . IEEE/IFIP I . C f. De e dab e S . Ne .*, 2013, pp. 1–8.

[10] M. Grottke and K. S. Trivedi, "Fighting bugs: Remove, retry, replicate, and rejuvenate," *C e*, vol. 40, no. 2, pp. 107–109, 2007.

[11] K. S. Trivedi, M. Grottke, and E. Andrade, "Software fault mitigation and availability assurance techniques," *I . J. S . A a ce E g. Ma age.*, vol. 1, no. 4, pp. 340–350, 2010.

[12] M. Grottke, D. S. Kim, R. Mansharamani, M. Nambiar, R. Natella, and K. S. Trivedi, "Recovery from software failures caused by mandelbugs," *IEEE Ta . Re .*, vol. 65, no. 1, pp. 70–87, Mar. 2016.

[13] K. S. Trivedi and A. Bobbio, *Re iabi i a d A ai abi i E gi ee i g: M de i g, A a i, a d A ica i*. Cambridge, U.K.: Cambridge Univ. Press, 2017.

[14] G. A. Hoffmann, K. S. Trivedi, and M. Malek, "A best practice guide to resources forecasting for the apache webserver," in *P c. IEEE 12 h Pac. Ri I . S . De e dab e C .*, 2006, pp. 183–193.

[15] K. Trivedi, D. Wang, D. J. Hunt, A. Rindos, W. E. Smith, and B. Vashaw, "Availability modeling of sip protocol on IBM WebSphere," in *P c. 14 h IEEE Pac. Ri I . S . De e dab e C .*, 2008, pp. 323–330.

[16] R. Matias, Jr., K. S. Trivedi, and P. R. Maciel, "Using accelerated life tests to estimate time to software aging failure," in *P c. IEEE 21 I . S . S f . Re . E g.*, 2010, pp. 211–219.

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

[18] J. Zhao, Y. Jin, K. S. Trivedi, and R. Matias, Jr., "Injecting memory leaks to accelerate software failures," in *P c. IEEE 22 d I . S . S f . Re . E g.*, 2011, pp. 260–269.

[19] W. B. Nelson, *Acce e a ed Te i g: S a i ica M de , Te P a , a d Da a A a i*, vol. 344. New York, NY, USA: Wiley, 2009.

[20] W. Q. Meeker and L. A. Escobar, *S a i ica Me h d f Re iabi i Da a*. New York, NY, USA: Wiley, 2014.

[21] D. C. Montgomery, *De ig a d A a i f E e i e*. New York, NY, USA: Wiley, 2017.

[22] M. Kifer and S. Smolka, *I d c i O e a i g S e De ig a d I e e a i : The OSP 2 A ach*. New York, NY, USA: Springer, 2007.

[23] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *O e a i g S e : Th ee Ea Piece*, vol. 151. Wisconsin, MI, USA: Arpaci-Dusseau Books, 2014.

[24] R. Bryant, O. David Richard, and O. David Richard, *C e S e : A P g a e' Pe pec i e*, vol. 281. Upper Saddle River, NJ, USA: Prentice-Hal, 2003.

[25] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," *ACM SIGPLAN N ice*, vol. 43, no. 3, pp. 329–339, 2008.

[26] D. G. Cavezza, R. Pietrantuono, J. Alonso, S. Russo, and K. S. Trivedi, "Reproducibility of environment-dependent software failures: An experience report," in *P c. IEEE 25 h I . S . S f . Re . E g.*, 2014, pp. 267–276.

[27] R. V. Lenth, "Quick and easy analysis of unreplicated factorials," *Tech - e ic*, vol. 31, no. 4, pp. 469–473, 1989.

[28] A. Goscinski, M. Brock, and P. C. Church, "High performance computing clouds," *C d C i g: Me h d g , S e a d A ica i*. Boca Raton, FL, USA: CRC Press, 2012, pp. 221–259.

[29] P. E. McKight and and J. Najab, "Kruskal–Wallis test," *C i i E c c - edia f P ch g*. New York, NY, USA: Wiley, 2010.

[30] A. J. Dobson and A. Barnett, *A I d c i Ge a i ed Li ea M de*. Boca Raton, FL, USA: CRC Press, 2008.

[31] R. Corporation, "Accelerated life testing reference," 2015. [Online]. Available: http://www.synthesisplatform.net/references/Accelerated_Life_Testing_Reference.pdf

[32] W. Mendenhall, R. J. Beaver, and B. M. Beaver, *I d c i P babi i a d S a i ic*. Boston, MA, USA: Cengage Learning, 2012.

[33] S. Park, R. W. Vuduc, and M. J. Harrold, "Falcon: Fault localization in concurrent programs," in *P c. 32 d ACM/IEEE I . C f. S f . E g.*, 2010, vol. 1, pp. 245–254.

[34] S. Khurshid, "Software-artifact infrastructure repository," Jan. 2018. [Online]. Available: http://sir.unl.edu/content/sir.php

[35] E. Schoenfelder, "Bug #38691 segfault/abort in update . . . join while flush tables with read lock," Aug. 2008. [Online]. Available: https://bugs.mysql.com/bug.php?id=38691

[36] N. Jalbert, C. Pereira, G. Pokam, and K. Sen, "Radbench: A concurrency bug benchmark suite," *H Pa*, vol. 11, pp. 2–2, 2011.

[37] P. Thomson, A. F. Donaldson, and A. Betts, "Concurrency testing using schedule bounding," *ACM SIGPLAN N ice*, vol. 49, no. 8, pp. 15–28, 2014.

[38] P. Menage, "cgroup documentation," 2004. [Online]. Available: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt

[39] Dormando, "Memcached configuring server," 2016. [Online]. Available: https://github.com/memcached/memcached/wiki/ConfiguringServer

[40] P. McCullagh, "Generalized linear models," *E i J. O e i Re* ., vol. 16, no. 3, pp. 285–292, 1984.
[41] T. W. Anderson and D. A. Darling, "A test of goodness of fit," *J. A e i S a i . A c.*, vol. 49, no. 268, pp. 765–769, 1954.
[42] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte, "A randomized scheduler with probabilistic guarantees of finding bugs," *ACM SIGPLAN N ice* , vol. 45, no. 3, pp. 167–178, 2010.
[43] S. Chandra and P. M. Chen, "Whither generic recovery from application