2015 IEEE/ACM 37th IEEE International Conference on Software Engineering

No PAIN, No Gain? The Utility of PArallel Fault INjections

Stefan Winter*, Oliver Schwahn*, Roberto Natella†, Neeraj Suri*, Domenico Cotroneo†
*DEEDS Group, TU Darmstadt, Darmstadt, Germany
†DIETI, Federico II University of Naples, Naples, Italy
{sw|os|suri}@cs.tu-darmstadt.de, {roberto.natella|cotroneo}@unina.it

***Abstract*—Software Fault Injection (SFI) is an established technique for assessing the robustness of a software under test by exposing it to faults in its operational environment. Depending on the complexity of this operational environment, the complexity of the software under test, and the number and type of faults, a thorough SFI assessment can entail (a) numerous experiments and (b) long experiment run times, which both contribute to a considerable execution time for the tests.**

In order to counteract this increase when dealing with complex systems, recent works propose to exploit parallel hardware to execute multiple experiments at the same time. While PArallel fault INjections (PAIN) yield higher experiment throughput, they are based on an implicit assumption of non-interference among the simultaneously executing experiments. In this paper we investigate the validity of this assumption and determine the trade-off between increased throughput and the accuracy of experimental results obtained from PAIN experiments.

I. Introduction

Software Fault Injection (SFI) [1]–[3] is used to experimentally assess the robustness of software systems against faults arising from hardware devices, third-party software components, untrusted users and other sources.

Given the high complexity of modern software, SFI typically requires a significant number of experiments to cover all relevant faults for the validation of fault-tolerance, with studies reporting thousands, or even millions, of injected faults [4]–[7]. The problem of high experiment counts is exacerbated by evidence that *simultaneous fault injections*, i.e., combinations of several injected faults, need to be considered as well. Recent studies [8], [9] show that failure recovery protocols in distributed systems exhibit vulnerabilities to simultaneously occurring faults and can, hence, only be uncovered by injecting fault combinations. A "combinatorial explosion" of the number of experiments is the consequence. Similar findings were obtained in recent work on operating systems and software libraries, which showed that software faults cause the simultaneous corruption of several interface parameters as well as shared memory contents [10] and that simultaneous corruptions can uncover robustness issues which would not be found by singular corruptions [11]. Finally, the injection of several faults (namely, higher order mutations, HOMs) is being increasingly investigated in the field of mutation testing[1], as HOMs have proven effective at improving the quality of test suites [12], [13]. Despite the demonstrated utility of simultaneous fault injections, the combinatorial explosion of the number of experiments remains a considerable challenge for their applicability.

In order to cope with the high number of Fault Injection (FI) experiments, two different strategies can be adopted. The first strategy attempts to reduce the number of experiments that need to be performed. Search-based techniques and sampling strategies for large test sets (e.g., [7], [9], [14], [15]) fall into this category. The second strategy attempts to utilize the increasing computational power of modern hardware, where several experiments are executed at the same time on the same machine (*parallelization*) to better utilize the parallelism of the underlying hardware (e.g., [16]–[18]). While parallelization ("throwing hardware at the problem") is less elegant, it is an appealing solution since it is generally applicable, whereas the applicability of sampling and search-based techniques depends on domain-specific knowledge in most cases. Parallelization, therefore, seems to be a promising solution to cope with the high number of experiments, especially as it can be combined with domain-specific sampling and pruning.

Nevertheless, parallelization relies on an implicit assumption that *executing several experiments in parallel does not affect the validity of results*. We hypothesize that this assumption is not trivial. Even if the experimenter takes great care to avoid interference between experiments (e.g., by running them on separate CPUs or virtual machines), there is a number of subtle factors (such as resource contention and timing of events) that can change the behavior of the target system (e.g., faults can lead to different failure modes than those observed under sequential execution), thus invalidating the results and negating the benefits of parallelization. This is a concern especially for embedded, real-time, and systems software, which are an important target of FI experiments, and where studies have shown that faults often exhibit non-determinism and time-sensitive behavior [5], [19].

Hence, in order to conduct *efficient and accurate* parallelization, we propose PArallel fault INjections (PAIN) as a SFI framework. As SFI is applied mostly for the assessment of critical systems, a major concern that outweighs performance considerations is the confidence in the validity of the experimental results; it is of utmost importance to avoid interference of PAIN experiments that affects their outcome. In addition to

[1] SFI and mutation testing use similar techniques to achieve their objectives, but these objectives differ with mutation testing primarily targeting the quality of test suites versus fault tolerance validation for SFI. We point the interested reader to [12] for a fuller discourse.

978-1-4799-1934-5/15 $31.00 © 2015 IEEE
DOI 10.1109/ICSE.2015.67

experiment throughput, we therefore also assess the validity of results from parallel experiments. Our paper makes the following contributions:

- We introduce an experimental environment for the study of parallel FI experiments and similar system-level tests.
- We conduct extensive FI experiments on the Linux kernel under an Android emulator environment. We qualitatively and quantitatively analyze the impact of parallelism on the experiment throughput and the validity of the results.
- We provide guidelines to tune the main factors that affect experiment throughput and the validity of PAIN experiments, including the degree of parallelism and failure detection timeouts.

Following the related work in Section II we present the hypotheses that our analyses are based on, followed by the corresponding experiment design/setup discussion over Sections III and IV. Section V presents our results and provides an in-depth analysis of the factors contributing to the observed effects. Sections VI and VII summarize the lessons learned and discuss threats to the validity of our empirical study.

II. Related Work: Fault Injection & Test Parallelization

A. Perspectives on Fault Injection (FI)

FI deliberately injects faults into a software to evaluate its *fault tolerance* properties. Examples of injections are CPU faults (emulated through memory corruptions), network faults (emulated through I/O exceptions and packet corruptions), and software bugs (emulated through code mutations). Such injections expose the software under test to exceptional and stressful conditions to achieve confidence in its reliability and performance in the presence of faults.

The scientific literature reports numerous applications of this approach, in particular in embedded, real-time, and systems software, such as operating systems (OSs). Examples are:

- Ng and Chen [20] adopted FI (by emulating both hardware and software faults) to validate the reliability of their *write-back file cache*. Using FI experiments, the authors identified weak points of their file cache design, and iteratively improved it until its *corruption rate* (in the presence of OS crashes) was comparable to a *write-through cache*. Swift et al. [21] injected faults in device driver code to assess the *memory isolation* of a novel fault-tolerant OS architecture.
- Arlat et al. [5] applied FI on a microkernel OS composed from off-the-shelf components to obtain quantitative data regarding *failure modes* of the microkernel, the *error propagation* among functional system components, and the *coverage* and *latency* of error detection mechanisms.
- Several FI-based dependability benchmarks have been proposed to compare alternative components such as operating systems [4], [22], web servers [23], and DBMSs [24]. In [4], [22], [25], UNIX and Windows OSs are compared with respect to the *severity of their failures*: For each FI experiment, the behavior of the OS is classified according to a failure severity scale, reflecting the impact of the fault on the stability and responsiveness of the system. In [23], two web servers are benchmarked with respect to *performance* (e.g., throughput and latency) and *fault tolerance* (e.g., percentage of failed HTTP requests) in the presence of faults injected in the OS. In [26], the *performability* (a unified measure for performance and dependability) of a fault-tolerant system is evaluated using a stochastic model, whose parameters are obtained from fault injection experiments.

There are numerous factors that can affect the accuracy and validity of such assessments, possibly leading experimenters to false conclusions about the dependability of a system if they are disregarded. As pointed out by Bondavalli et al. [27], FI tools and experiments should be regarded as *measuring instruments* for dependability attributes and designed with the principles of *measurement theory* in mind.

For this reason, much research in this field is focused on metrological aspects of FI. Several techniques were developed to reduce the *intrusiveness* of FI tools on the target system, by taking advantage of debugging and monitoring facilities provided by hardware circuitry [28], [29], and by minimizing changes in the target software [30]. Skarin et al. [31] assessed the *metrological compatibility* (i.e., statistical equivalence) of results obtained from these alternative FI techniques. Van der Kouwe et al. [32] evaluated the *distortion* of results due to injected faults that have no effect on the system and are underrepresented. In [33], [34], the *precision* and the *repeatability* of FI is evaluated in the context of distributed systems, which are affected by non-determinism and by clock skew issues. Irrera et al. [35] assessed whether *virtualization environments* could be used for FI experiments without impacting on system metrics related to performance and resource usage: While their conclusions are positive, their experiments show that virtualization actually had a noticeable impact on some of the monitored metrics.

B. Perspectives on Test Parallelization

In the software community, the idea of test parallelization has been mostly driven by the advent of increasingly parallel system architectures, such as multiprocessor and networked systems [36]. Kapfhammer proposed parallel executions to complement sampling techniques for improving the performance of regression testing [37]. Lastovetsky used parallel computing to achieve a throughput improvement (*speedup*) of factor 6.8 to 7.7 compared to sequential execution [16] for testing of a complex CORBA implementation on several platforms. Duarte et al. developed GridUnit as an extension of the JUnit testing framework to support parallel test execution in a grid computing environment, achieving speedups ranging between 2 and 71.11 [17], [38], [39]. Parveen et al. reported a speedup of 30 for a 150-node cluster using their MapReduce-based unit testing tool HadoopUnit [40]. Oriol and Ullah ported the York Extensible Testing Infrastructure (YETI) to MapReduce architectures using Hadoop [18]. They reported a speedup of 4.76 for a fivefold increase in computational

resources. In contrast to other work, which only reported on performance, they also compared the results of sequential versus parallel tests and reported that the numbers of detected defects are equal in both cases. However, they did not specify whether the same defects were detected or just equal numbers. Other recent approaches dealt with Testing-as-a-Service (TaaS) for both dynamic tests [41], [42] and program analyses [43]–[46].

While multiple test parallelization approaches have been advocated, their primary focus was to increase test throughput. Interferences between parallel tests were not investigated in these studies, because tests were performed on individual software units rather than integrated systems, and the execution of test cases was not influenced by non-deterministic factors such as timing and resource contention [18], [38]. In other cases, tests that contended for shared resources were executed sequentially or on distinct hardware machines to conservatively avoid any interference [16]. Nevertheless, interferences between parallel experiments are a potential, yet unexplored, issue for FI experiments, since they usually target complex, integrated systems rather than individual components, and since injected faults (like real faults) can result in unforeseen and non-deterministic behavior [47], [48].

C. FI Validity in Parallel Execution Environments

While some studies have advocated the potential benefits of parallelizing FI experiments [46], [49]–[51], none of them investigated the impact of parallelism on the validity of results (i.e., whether results from parallel experiments are metrologically compatible to sequential ones). In these studies, FI experiments were executed in separate virtual machines [49], [50] and OS processes [51] to isolate the experiments. Memory protection mechanisms provided by virtualization and the OS can prevent data corruptions from propagating among experiments. For this reason, experiments are assumed to be independent from each other, and are treated as an "embarrassingly parallel" problem (i.e., experiments can be arbitrarily parallelized). Nevertheless, there are potentially adverse effects of parallelization: In fact, it is difficult to enforce perfect *performance isolation* among virtual environments [52], [53], and performance interferences (e.g., the shortage of resources or the timing of events) can significantly change the *behavior* of a system, and even affect its security [54], [55]. Thus, we investigate the interplay between parallelism, the increase of experiment throughput, and the experimental results.

III. Experimental Design

We experimentally assess the feasibility of *increasing SFI experiments throughput by parallel execution without compromising the accuracy of the results*, by addressing the following research questions (RQ).

RQ 1 *Can parallel executions of SFI experiments on the same machine increase the throughput of SFI experiments?*

RQ 2 *Can parallel executions of SFI experiments on the same machine change the results obtained from SFI experiments?*

If the answer to RQ 1 is positive and to RQ 2 negative, then SFI parallelization has no adverse effects and should be applied whenever parallel hardware is available. However, if RQ 1 is negative and RQ 2 positive, parallelization should be avoided. If both answers are negative, the decision whether to parallelize or not should be driven by other factors, such as hardware cost or complexity of the experiment setup. If both are positive, parallelism can be beneficial, but it can also potentially affect the accuracy of results. In this case, we need to investigate:

RQ 3 *If RQ 1 and RQ 2 hold, can the parallelization of experiments be tuned to achieve both a (desirable) throughput increase and avoid the (undesirable) inaccuracy of results?*

In the following we introduce the basic system and fault models for our study along with the technical terms to derive detailed hypotheses from the stated research questions.

A. System Model

We investigate the impact of parallelism in an experimental context similar to contemporary FI studies (cf. Section II-A). In particular, the SFI experiments, on which our study is based, focus on the robustness of operating system (OS) kernels against faulty device drivers. Device drivers have been shown to have high defect rates [56], [57] and at the same time their failures have severe consequences on overall system stability [58], [59]. FI into device drivers helps identifying faults that cause critical kernel failures and provides useful feedback to improve the kernel's robustness [5], [20].

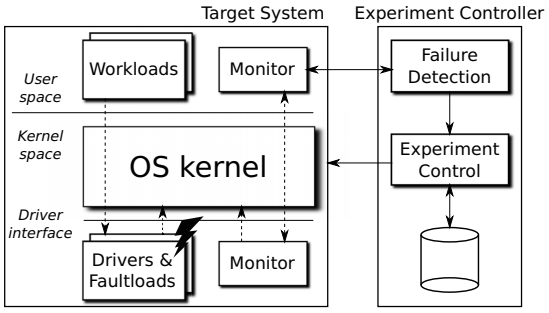

Fig. 1. System model and experiment setup

Therefore, we create faulty versions of driver code, load these faulty drivers, and execute a workload to trigger the altered code. Figure 1 depicts our setup. All experiment control logic is external to the target system, which is encapsulated in a virtual machine (VM), to prevent it from being corrupted by the effects of injected faults. To detect failures of the target system, we use *external failure detectors* that collect and analyze messages from the target system, while running in the experiment controller's separate virtual environment. Using these detectors, we are able to reliably detect:

- **System Crashes (SC)** by monitoring kernel messages emitted by the VM,
- Severe **System Errors (SE)** in a similar manner, and

- **Workload Failures (WF)** by monitoring application log information from the workloads forwarded to the system log that the VM emits.

Our setup also detects stalls (so-called *hangs*) of the target system, in which experiments neither make any progress nor produce any information about the failure. We assume that the system hangs if it does not terminate within a timeout interval, which is calculated by adding a generous safety margin to the maximum time needed by the target to produce the correct result when no fault is present. Our timeout-based external detectors *assume* hangs using these timeouts:

- During system initialization (**Init Hang Assumed, IHA**),
- After system initialization (**System Hang Assumed, SHA**), or
- During workload execution (**Workload Hang Assumed, WHA**).

As such timeout-based external detectors are known to be possibly imprecise or inefficient depending on their configuration policy [60]–[62], we additionally employ sophisticated hang detectors inspired by the approach of Zhu et al. [62]. We include two additional hang detectors in the target system: A *light detector*, executing as a user space process, monitors basic system load statistics. If these statistics indicate lack of progress, the light detector triggers a *heavy detector* executing in kernel space. The heavy detector performs a more accurate (but also more complex and time consuming) analysis and triggers a controlled system crash if a hang is detected. The tests that the light and heavy detectors apply are identical to those suggested in [62]. The used threshold values for our hardware configurations can be found in our source code at [63]. This target-internal hang detection infrastructure provides us with two additional failure modes:

- **System Hang Detected (SHD)**
- **Workload Hang Detected (WHD)**

B. The SFI Fault Model

The *fault model* in SFI experiments defines the corruptions or disturbances to be introduced in the target system. In our experiments, we consider the injection of *driver source code changes* to emulate residual defects of device drivers, in a similar way to recent studies on OS software fault tolerance [20], [21] and on dependability benchmarking [22]–[24]. We adopted the SAFE tool [7] to inject realistic code changes that were defined on the basis of actual faults found in commercial and open-source OSs [2], [64].[2]

As faulty drivers are known to constitute a severe threat to system stability, the target system in our SFI experiments has to be executed in a strictly isolated environment that (a) prevents experiments from affecting the test bench and (b) enables subsequent experiments to start from a clean environment free from effects of previous experiments. These requirements results in high experiment overheads and limit experiment throughput, which parallelism is supposed to compensate for.

[2]For space restrictions, we refer the reader to [7] for a detailed exposition on fault types.

A second issue which parallelism is supposed to mitigate is the high number of experiments to be executed, especially when combinations of multiple faults are considered. In order to investigate the impact of parallelism on high volumes of SFI experiments, we injected both single and multiple faults into driver code. Multiple faults are injected by repeatedly invoking SAFE on previously mutated driver code, in a similar way to emerging Higher-Order Mutation Testing approaches [12]. The injection of multiple faults leads to a combinatorial explosion of the number of faulty drivers to test with, and allows us to experiment with a high volume of SFI experiments.

C. Performance and Result Accuracy Measures

Performance measure: We, and other researchers [49], [50], argue that a speedup of FI experiments is desirable to achieve a better coverage of fault conditions to test with. The performance metric of interest is *experiment throughput*, measured as *average number of experiments per hour*.

The accuracy of SFI results needs to be defined in statistical terms, since the outcome of FI experiments is influenced by non-deterministic factors. This aspect may not be intuitive, and requires a more detailed discussion. In order to observe the effects of injections, the injected code needs to be *activated* during experiment execution [48]. As the abstraction from hardware configuration details and the orchestration of access to hardware devices are among the core functions of OSs, most of them do not provide a direct interface to driver functions for programmers. As a consequence, a large and complex software layer interposes between device drivers and user programs (cf. Figure 1). Some driver functions (e.g., those related to power management) may even be entirely hidden from user programs and invoked by the OS upon (possibly non-deterministic) hardware events and task scheduling.

Accuracy measures: Our measure for *result accuracy* has two aspects. First, we are interested whether *result distributions of failure modes* (Section III-A) change with increased parallelism. Second, we assess the degree of result heterogeneity across repetitions of identical experiment sets, as this indicates the reproducibility and representativeness of the results. While for the first case, a binary measure that indicates statistically significant deviations suffices, we require a comparative metric for the latter. We choose a *Chi square test for independence* to decide whether or not result distributions for parallel experiments differ significantly from result distribution for sequential experiments. To measure the variance of distribution samples we interpret the obtained distributions as vectors and calculate their *Euclidean distances from the mean*. We use the mean value of all such distances within a set of repetitions for the same setup as heterogeneity metric d.

D. Hypotheses

On the background of the introduced models and terminology we can derive precise hypotheses from the research questions stated in the beginning of Section III. We only state

the null hypotheses to be tested. The alternative hypotheses are the negation of the null hypotheses stated below.

Hypothesis H_0 1 *If the number of parallel experiment instances running on the same physical machine is increased, the experiment throughput does not increase.*

Hypothesis H_0 2 *If the number of parallel experiment instances running on the same physical machine is increased, the obtained result distribution of failure modes is independent from that increase.*

Hypothesis H_0 3 *If the number of parallel experiment instances running on the same physical machine is increased, the heterogeneity among repeated injection campaigns does not increase.*

IV. Experiment Setup and Execution

All software developed to conduct our experiments (currently 14 159 physical source lines of code[3]) is publicly available at github [63] for the reproduction and cross-validation of our results.

A. Target System

The operating system we are targeting in our analysis is the Android mobile OS [65]. We run Android 4.4.2 "KitKat" with a 3.4 kernel from Google's repositories [66] in the goldfish System-on-Chip emulator [67], which is based on the QEMU emulator/virtualizer [68] and ships with the Android SDK.

We inject faults into the MMC driver for the emulated SD card reader of the goldfish platform. The driver has 435 physical source lines of code. We are using two different synthetic workloads to trigger injected faults in the MMC driver, all of which are based on code from Roy Longbottom's Android benchmarks [69].

The first workload ("pure") performs file operations on the emulator's virtual SD card in order to (indirectly) exercise the MMC driver and faults injected there. We use code from the DriveSpeed benchmark app and configure it to stress the SD card driver for approximately 30 seconds.

A "mixed" workload adds CPU and memory load to the pure workload. The goal is to create a more diverse utilization of system resources by the emulator to cover a wider range of possible interference between emulator instances competing for shared system resources. Besides DriveSpeed, we use code from the LinpackJava and RandMem benchmarks. All benchmarks are executed as parallel threads, leaving their scheduling to the Android OS.

We use an additional thread in the workload apps to perform WF failure detection, since application failures are signaled as exceptions within Android's Dalvik runtime and need to be explicitly forwarded to the external failure detector residing outside of the emulator.

[3]generated using David A. Wheeler's SLOCCount

B. Fault Load

Applying SAFE for creating faulty versions of the MMC driver we obtain 273 mutants to test with. If we recursively apply SAFE to each of these mutants, as described in Section III-B, we obtain 70,167 *second order mutants* and, hence, a total of 70,440 faulty driver variants to test with. This drastic increase in numbers from first to second order mutants illustrates the combinatorial explosion resulting from simultaneous fault injections and higher order mutation testing.

To test the hypotheses outlined in Section III-D, we restrict ourselves to a campaign of 400 randomly sampled mutants from the set of first and second order mutants. As the outcomes of experiments are subject to non-determinism and experiment repetitions are required to establish confidence in the obtained results, we repeat this campaign three times for each considered setup to account for non-determinism (Section III-C).

C. Execution Environments

In order to conduct parallel tests on the same hardware, we replicate instances of the goldfish emulator, in which the target system is executing, on a single host machine. This parallelization by replication of emulator instances reflects the implicit assumption of non-interference we are questioning, as emulation and virtualization form the basis of recent approaches to test parallelization [41], [42], [46], [49], [50].

In order to avoid effects from a single platform to bias our results, we execute all experiments on two different platforms:

- A desktop configuration with an AMD quad-core CPU, 8 GB main memory, and a 500 GB hard drive operating at 7,200 RPM.
- A server configuration with two Intel Xeon octa-core CPUs, 64 GB main memory, and a 500 GB hard drive operating at 7,200 RPM.

To avoid differing CPU frequencies from biasing results, we have disabled frequency scaling and set all cores on both machines to constantly operate at 1.8 GHz (which was the only possible common configuration on both platforms). We have also disabled hyper threading on the Intel processors to achieve better comparability with the AMD processors, which do not provide this feature. The desktop configuration is running Ubuntu 13.10, the server configuration CentOS 6.5.

The number of parallel instances running on the same machine, which is the controlled variable in our experiments, is initially chosen as a) 1 (sequential) and b) 2N, where N is the total number of physical cores available in the machine. 2N is a common configuration to maximize hardware utilization. By launching more instances than actually available CPU cores, the cores are more likely to be utilized when some processes wait for I/O, without requiring frequent migration between cores, which would impair the effectiveness of L1 and L2 caches. These two basic settings suffice for a fundamental assessment of the hypotheses stated in Section III-D. To answer RQ 3, we expand the analysis to further degrees of parallelism.

The different factors outlined above yield 8 different configurations (2 workloads, 2 hardware platforms, 2 degrees of

parallelism). For each of these we execute our campaign of 400 experiments three times (cf. Section IV-B). In total, to test Hypotheses 1 to 3, we execute 24 campaigns with 400 experiments each. We report the results of these experiments in the following section, along with a set of additional experiments for an in-depth analysis of these results.

V. Experimental Results and Data Analysis

A. Initial Results

Table I shows the results of the 24 campaigns to test Hypotheses 1 and 2. Each row lists the results for a different setup. The HW column specifies the hardware platform, the # column the number of parallel instances of the emulator performing the experiments, and the WL column the used workload. The next ten columns contain the average number of experiments that resulted in the corresponding failure mode. In addition to the eight failure modes specified in Section III-A, there are two additional columns for possible experiment results: An **Invalid** class of experiment results for cases in which the experiment control logic had to abort the experiment due to unforeseen failures within the control logic, and a **No Failure (NF)** class for experiments that finished execution without any failure detection. Unforeseen failures include (for instance) cases when the host OS cannot fulfill requests for resource allocations. The last three columns list the throughput in experiments per hour, the average experiment duration in seconds, and the average Euclidean distance of obtained samples from the mean values as introduced in Section III-C.

From the obtained experiment throughput data, we clearly **reject Hypothesis 1**: In the parallel case the average experiment throughput is considerably higher than in the sequential case. We achieve an average speedup between 4 and 4.5 for a 8-fold increase of instances on the desktop machines and an average speedup between 9.4 and 10 for a 32-fold increase on the servers. The throughput calculation is not only based on experiment execution times, but also includes the processing overhead of the experiment control logic. From our experience, however, this overhead is small (less than one second per experiment) compared to the experiment duration.

To test Hypothesis 2, we conduct a Chi-square test for independence to assess if the observed result distributions are statistically independent from the degree of parallelism. As we perform multiple tests simultaneously on the same population, we account for the risk of false discoveries (i.e., incorrect rejections of the null hypothesis) by adjusting p-values according to the Benjamini-Hochberg procedure [70]. These adjusted p-values (p) and the corresponding test results are shown in Table II, along with the normalized Pearson coefficients (r). The normalized Pearson coefficient indicates the relative "strength" of correlation and can be used to compare the degree of correlation between parallel/sequential execution and the different result distributions. The coefficient also indicates positive or negative correlation. We have used the absolute numbers from the distributions rather than the mean values for the Chi square tests. The results in Table II indicate that there is no independence of the result distributions from parallelism in three out of four cases, and we therefore **reject Hypothesis 2**.

To test Hypothesis 3, we calculate the Euclidean distance of the mean distributions shown in Table I to each of the three distribution samples they were derived from. The mean values of these distance measures are shown in the d column of Table I. The heterogeneity of parallel result distributions is between 1.7 and 10.1 times higher than the heterogeneity of sequential results and we therefore **reject Hypothesis 3**.

From the presented results, both RQs 1 and 2 are positively answered and we proceed to address RQ 3.

B. Influence of Timeout-Values on the Result Distribution

In order to better understand the trade-off between experiment throughput and result accuracy, we take a closer look at the observed changes in the result distributions. While the numbers for the result classes Invalid, SC, SE, WHD, and SHA only marginally differ across the different setups, we see major deviations in SHD and WHA failure modes, especially illustrated by the drastic change in the distribution for 8 instances on the desktop setup with the mixed workload (cf. Table I). As both failure modes are related to hang detectors and these depend on timeouts for detection, we suspect that the increased rates for parallel experiments are false positives of these detectors and that their timeouts need adjustment in the parallel case. Indeed, the experiments exhibit longer execution times in the parallel case, as the *Experiment Duration* column of Table I shows. Compared to sequential experiments, the execution times are roughly doubled. To prevent the longer execution times from affecting results, we chose to triple the timeout values for the WHA, SHA, and IHA detectors. A high timeout value leads to unnecessarily long wait times in the case of an actual hang failure and we discuss better strategies in Section V-C. In our case, we see a relatively small fraction of hang failures and consider the longer detection time a reasonable cost for more accurate results.

After eliminating this potential source of deviations in the result distributions, we performed the parallel campaigns again with the modified setup. We focus all further analyses on experiments with the mixed workload, as we observe higher correlation coefficients for this workload in both setups. Table III shows the obtained results, which are closer to the distributions obtained for sequential runs. However, while the heterogeneity of results has decreased for the server setup, it has increased for the desktop setup and this divergence of result accuracy is also reflected by the Chi square test results in the first two rows of Table V: While the result distribution for the desktop setup still significantly correlates with the degree of parallelism, it is statistically independent for the server setup. The distribution differences that lead to this indication of diverging results are mostly due to differing SHD and WHA failure counts. As a consequence, we further look into timeout selection strategies for the corresponding detectors on this platform in Section V-C.

The improved accuracy of experimental results on the server platform raises the question, whether similar systematic causes

TABLE I
MEAN FAILURE MODE DISTRIBUTIONS, PERFORMANCE AND ACCURACY MEASURES FROM 24 EXPERIMENT CAMPAIGNS

Setup			Failure Modes										Performance and Accuracy Measures		
HW	#	WL	Invalid	NF	SC	SE	WF	SHD	WHD	SHA	WHA	IHA	Throughput (exp./h)	Experiment Duration (s)	d
Desktop	1	pure	0.00	108.67	97.00	0	182.67	6.33	0	0	0.00	5.33	16.4	219.21	0.89
Desktop	1	mixed	0.00	108.00	97.00	0	182.00	0.00	0	0	6.33	6.67	12.5	286.97	2.02
Server	1	pure	0.00	101.33	97.00	0	183.00	18.67	0	0	0.00	0.00	14.6	246.38	1.26
Server	1	mixed	0.00	114.67	97.00	0	183.00	5.33	0	0	0.00	0.00	12.2	295.36	0.63
Desktop	8	pure	0.00	42.00	97.00	0	181.33	25.67	0	0	48.67	5.33	67.1	416.23	7.34
Desktop	8	mixed	0.00	1.00	96.67	0	6.33	10.00	0	1	281.67	3.33	56.1	493.77	3.50
Server	32	pure	0.33	95.00	97.00	0	182.67	22.00	0	0	0.00	3.00	146.3	496.98	3.37
Server	32	mixed	0.00	65.00	97.00	0	179.00	5.00	0	0	48.00	6.00	115.6	616.19	6.35

TABLE II
CHI-SQUARE TEST OF INDEPENDENCE FOR PARALLELISM AND INITIAL RESULT DISTRIBUTIONS

HW	WL	p	r	Result
Desktop	pure	3.2×10^{-55}	0.45	reject
Desktop	mixed	0	0.90	reject
Server	pure	2.1×10^{-1}	0.1	do not reject
Server	mixed	4.3×10^{-41}	0.40	reject

of result divergence can be identified. In order to provoke a stronger impact of such potential effects on the result distributions, we performed additional campaigns on the server platform with 36, 40, 44, and 48 instances. The results are shown in Table IV. The comparatively large improvement in terms of throughput is probably due to a minor change in the experiment logic to regularly clean parts of the host file system from temporary files used by the experiment controller. As the Chi square tests for independence in Table V show, the result distributions for 36, 40, and 44 instances do not significantly differ from the results obtained from the sequential campaigns, whereas the distributions for 48 instances do. We also observe a strong increase of the heterogeneity measure d for the latter. Although the results for the setups with 36, 40, and 44 instances are still more heterogeneous than the results obtained from sequential experiments, we deem it acceptable and expect results to further stabilize with moderately higher numbers of campaign repetitions.

C. Selection of Timeout-Values for Parallel Experiments

As it turns out, the selection of suitable timeout values for our hang detectors is crucial to avoid false positive hang detections, and good estimates become challenging for parallel SFI experiments. Initially, we chose timeout values based on our experience with previous sequential experiments and added a generous safety margin. This turned out to be insufficient for parallel executions, as is evident from the drastically increased WHA detections in Table I. A naïve approach to avoid these false positives would be to upscale the timeout values with the number of parallel instances. However, as the degree by which execution times increase with the numbers of instances is generally unknown and, according to our results, also depends on the execution platform (hardware and OS), this entails an iterative process of trial and error until suitable timeout values are found.

A better strategy is to estimate values based on observations made during so-called *golden runs* on the intended execution platform *without injections*. Such runs should be performed for each targeted level of parallelism and relevant timing data be recorded as a baseline to derive suitable timeout values. We performed a number of such calibration runs for our parallel desktop setup with the mixed workload using a modified version of our experiment controller. During calibration runs we record the times needed for system initialization (*sysinit*) and for workload completion, as these two time values are relevant to our IHA and WHA detectors. The recorded times appear to be normally distributed. Therefore, we fitted normal distributions to our data and used their 99.99 percentiles as our timeout values, i.e., $99.99\,\%$ of experiments should execute without false hang detections if the fitted distribution matches the actual distribution of run times. However, the repetition of the original experiments for the same setup using these timeout values yielded unsatisfactory results with 106 WHA and 3 IHA detections in average. While the number of IHA detections is in the expected range for the 99.99 percentile, the number of WHA detections, which is considerably lower than with our original timeout values (cf. Table I), is still 10 times higher than with our tripled timeouts (cf. Table III). The difference in the quality of the estimated timeout values indicates that our calibration approach is suitable for estimating system initialization timeouts, but not workload timeouts. Since we used a modified version of our experiment controller to perform the calibration runs, we assume the workload behavior for real experiments to differ from our calibration observations.

To further investigate the timing behavior for real experiments, we repeated the original experiments for our desktop setup with the mixed workload and stepwise increased degrees of parallelism. To avoid interference from any spurious detections, we set extremely high timeout values for our hang detectors and extracted the system initialization and workload times from experiment logs. In order to exclude outliers in the collected data, we only include time values between the 0.5 and 99.5 percentiles in our analysis. The

TABLE III
MEAN FAILURE MODE DISTRIBUTIONS, PERFORMANCE AND ACCURACY MEASURES FROM REPEATED PARALLEL EXPERIMENTS

Setup			Failure Modes										Performance and Accuracy Measures		
HW	#	WL	Invalid	NF	SC	SE	WF	SHD	WHD	SHA	WHA	IHA	Throughput (exp./h)	Experiment Duration (s)	d
Desktop	8	mixed	0	104	97	0	181.67	5.00	0	0.67	11.33	0.33	47.0	587.25	5.41
Server	32	mixed	0	114	97	0	181.67	6.67	0	0.67	0.00	0.00	118.1	619.48	1.99

TABLE IV
MEAN FAILURE MODE DISTRIBUTIONS, PERFORMANCE AND ACCURACY MEASURES FROM HIGHLY PARALLEL EXPERIMENTS

Setup			Failure Modes										Performance and Accuracy Measures		
HW	#	WL	Invalid	NF	SC	SE	WF	SHD	WHD	SHA	WHA	IHA	Throughput (exp./h)	Experiment Duration (s)	d
Server	36	mixed	0.00	113.67	97	0	181.67	7.00	0	0.33	0.33	0.00	157.1	712.11	2.16
Server	40	mixed	0.67	113.00	97	0	180.00	8.00	0	0.67	0.67	0.00	154.1	834.14	1.98
Server	44	mixed	0.00	112.00	97	0	180.33	6.67	0	1.33	2.33	0.33	143.0	951.52	3.54
Server	48	mixed	0.67	104.67	96	0	177.67	11.00	0	2.00	5.00	3.00	102.5	1069.03	6.81

TABLE V
CHI-SQUARE TEST OF INDEPENDENCE FOR PARALLELISM AND ADDITIONAL RESULT DISTRIBUTIONS

HW	#	WL	p	r	Result
Desktop	8	mixed	6.7×10^{-7}	0.18	reject
Server	32	mixed	1.0	0.05	do not reject
Server	36	mixed	1.0	0.05	do not reject
Server	40	mixed	7.8×10^{-1}	0.08	do not reject
Server	44	mixed	2.1×10^{-1}	0.10	do not reject
Server	48	mixed	1.3×10^{-4}	0.17	reject

graph in Figure 2 visualizes the minimum and maximum times observed. With increasing parallelism, the difference between the observed minimum and maximum times also increases, with a maximum difference of about 618 seconds for the system initialization and about 323 seconds for the workload times. Moreover, the maximum times increase with increasing parallelism whereas the minimum times are almost constant.

A comparison of the observed time distributions for our calibration runs and the repeated experiment runs shows that the workload execution completes significantly faster in the calibration setup: With 8 parallel instances, for instance, the workload completes after about 98 seconds on average in the calibration, but needs about 304 seconds on average in the experiment setup.

As our results show that a dedicated calibration setup may exhibit different timing behavior than real experiments, we conclude that timing data from real experiments should be used for choosing suitable timeout values. Using the 99.99 percentile of a distribution fitted to the timing data from actual experiments, we obtained a number of hang detections comparable to the original experiments with tripled timeout values (cf. Table III), confirming the suitability of our approach. The new timeout value for the IHA detector is about 300 seconds

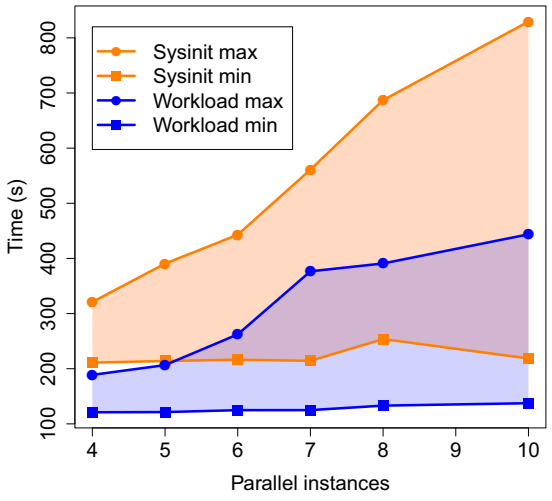

Fig. 2. Minimum and maximum times for system initialization and mixed workload execution observed during experiment runs with increasing levels of parallelism for the desktop setup.

shorter than the original tripled value, whereas the new WHA detector timeout is about 30 seconds longer.

With timeout values of comparable or even better accuracy than the previous trial and error approach, our systematic approach to calculating timeout values is preferable if it results in acceptable overhead. For the systematic timeout estimation, we used timing data from 3 complete experiment campaigns, summing up to 1200 experiments. While this overhead is acceptable for large SFI campaigns, we attempt to improve the performance of our timeout value assessment by finding a subset of the available data samples that provides us with sufficiently accurate timeout estimations. For this purpose we

randomly select subsets of varying sizes of the timing data and apply the timeout estimation approach described above. The resulting timeout values for 10 % steps in sample sizes are illustrated in Figure 3. 1.0 in the figure corresponds to 873 samples for the sysinit times and 298 samples for the workload times. This is a subset of the total experiment count, as we only have valid timing data for experiments that did not finish their execution prematurely. For instance, if in an experiment the system crashed during initialization, we have no valid sysinit time for that test case. Since the workload times have a narrower distribution (cf. Figure 2), less overall samples are needed to obtain a robust estimation than for the sysinit times. Starting from a sample size of 0.5 to 0.6, the estimated timeout values are very close to the value estimated using all available samples. Thus, the execution of only 2 experiment campaigns would have sufficed for the estimation of suitable timeout values.

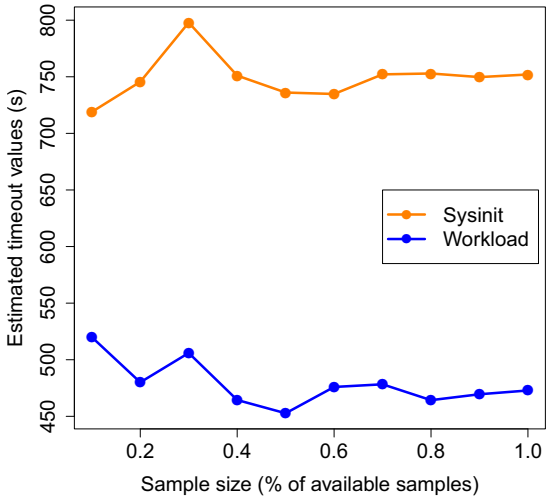

Fig. 3. Estimated timeout values for different samples sizes of observed real experiment times.

VI. Discussion

The experiments performed in this study provided interesting insights about the result accuracy and performance of FI experiments, and we summarize the main lessons that can be drawn from them in this section. A subtle aspect that turned out to be more important than we initially expected was:

> *The correct setup and tuning of parallel experiments may prove difficult and requires special care.*

We observed that the parallel execution of experiments had a significant impact on the duration and the timing behavior of individual experiments. Besides changing the mean experiment duration, parallelism also led to incorrect failure detections in our first round of experiments. Even though failure detector timeouts were calibrated on the basis of preliminary fault-free parallel runs, they turned out to be inadequate to account for non-deterministic execution delays caused by unexpected interactions among failing and non-failing VMs that were not observed in fault-free runs.

Moreover, the technical implementation of our PAIN setup (not discussed in detail in this paper for the sake of brevity), being concurrent and complex by itself, required significant efforts for testing and debugging. To achieve a reliable setup, we had to find and fix transient and subtle issues related to portability across different platforms (e.g., the desktop and the server platforms had different limits on the maximum number of processes and on other resources, causing experiment failures), resource leaks of the target system (in the case of the Android emulator, temporary files), and communication between the experiment controller and the target system, which in some cases went out of sync due to unexpected timing behavior and the loss of messages. On the basis of this experience, we advise researchers and practitioners, who design parallel FI setups, to pay attention to these aspects. As PAIN has matured to avoid a number of such issues, we hope that it provides a useful basis for future parallel experiments and experimentation frameworks.

Going back to the initial research questions, our experiments provide evidence that parallelism can influence both experiment performance and results. Executing experiments in parallel can significantly (RQ1) increase experiment throughput with a speedup factor of up to 10. Thus, FI experiments can be included among the computer applications that can benefit from parallelism. Nevertheless, we also observed that:

> *The parallel execution of FI experiments can improve experiment throughput, but can also affect the accuracy of results.*

The statistically significant differences reported in our analysis show that there are cases in which parallelism can actually change the results of experiments (RQ2), both in terms of failure mode distributions and of result stability across repetitions. Such effects, ultimately, affect the conclusions about fault tolerance properties of the target system. In our study, performance interferences between parallel instances changed the failure mode of a subset of experiments (that were originally not failing or failing differently) to hang failures of the OS and the workload. Moreover, these hang failures were not easily reproducible and led to unstable failure distributions.

Such changes in the results were observed for very high degrees of parallelism, e.g., up to three times the number of cores (48 parallel experiments) in the case of our server setup. Nevertheless, we also found that running parallel experiments at a lower degree of parallelism (e.g., 32 parallel experiments for the server setup with appropriate timeouts) do not cause statistically significant variations from the results of sequential executions. This observation indicates that parallelism does

not necessarily harm result accuracy (RQ3), if the degree of parallelism is not too high.

Therefore, to maximize the experiment throughput while preserving result accuracy, it is important to properly tune the degree of parallelism. For instance, the best throughput in the server setup with mixed workload was achieved when running 36 parallel instances (157.1 experiments per hour on average). Further increasing the number of parallel instances to 48 gradually reduced the throughput (down to 102.5 experiments per hour) and caused statistically significant inaccuracies. This behavior illustrates that result inaccuracies occur when the number of parallel instances exceeds the system's capacity:

> *Negative effects on result accuracy can be avoided, if the degree of parallelism is carefully tuned for best throughput.*

Consequently, the tester has to determine the appropriate degree of parallelism for his setup by running parallel experiments. For instance, by performing preliminary parallel runs with an increasing number of instances is a viable method for finding a suitable degree of parallelism. We have demonstrated that such runs should not use modified or even simplified versions of the intended experiment workload, as these may yield significantly different loads that, for instance, lead to different timing behavior and inaccurate assessments of resource utilization. We note that, while our study focused on FI experiments, these issues also apply to other forms of testing (e.g., unit testing frameworks using fixed timeouts per test case) and systems that exhibit non-deterministic behavior (e.g., concurrent software).

VII. Threats to Validity

For any empirical study, care must be taken when interpreting the results and drawing conclusions. The main threats to validity that we identified are the choices for the injection target, the fault model, the workload, and the accuracy measures of the experimental design.

We chose the Linux kernel as *injection target*. Even if this system is not representative of every software system, it is representative of embedded, real-time, and systems software, and as such among the most relevant targets for FI. Moreover, being a complex OS, this system includes several *non-determinism* factors whose influence we wanted to investigate, including concurrency (both in user and kernel space), I/O interactions, and the use of non-deterministic heuristics adopted in many parts of the OS, such as in task and I/O scheduling and page reclaim algorithms.

The *fault model* adopted in our experiments was based on the injection of software faults through code mutation, whose representativeness is supported by extensive analyses of real faults and mutants, and which are an accepted practice [71]–[74]. In order to investigate the utility of parallelization for coping with the high number of experiments that result from multiple FIs, we inject both 1st and 2nd order mutants. Even if the realism of multiple injections (with respect to real faults) has not been investigated in depth, multiple injections are already used in FI studies [9], [11], [12] and are representative of experiments conducted by practitioners.

As *workload*, we adopted existing performance benchmarks for the Android platform. Although benchmarks may not be representative of specific user scenarios, they are the most typical workloads in FI experiments, especially in the context of dependability benchmarks [20], [24], [75]. Moreover, performance benchmarks are stressful workloads and increase the likelihood of fault activation during experiments [76].

The *accuracy measures* in our experiments were focused on the number and the distribution of failure modes, since this is one of the most important aspects of FI experiments, which are often aimed at evaluating the likelihood of high-severity failure modes [4], [20], [22], [77]. Moreover, measurements of fault-tolerance properties, such as coverage and latency, strictly depend on the failure types observed during experiments. Thus, avoiding distortions of the failure distribution is of great importance for fault-tolerance evaluations.

VIII. Conclusion

As software systems become more complex and, at the same time, tend to exhibit sensitivity to complex fault conditions [8]–[11], the number of relevant fault conditions to test against also increases drastically. The simultaneous execution of such tests on parallel hardware has been advocated as a viable strategy to cope with rapidly increasing test counts. In this paper we have addressed the question whether such strategies can be applied to speed up Software Fault Injection (SFI) experiments by performing PArallel INjections (PAIN). Besides assessing the speedup of experiment throughput, we address the question whether PAIN affects the accuracy of SFI experiments. We define two measures of "metrological compatibility" for SFI experiments (i.e., *how accurately results from PAIN reflect results from sequential experiments*), and apply them in a case study targeting the Android OS.

Our results show that while PAIN significantly improves the throughput, it also impairs the metrological compatibility of the results. We found that result inaccuracy is related both to the degree of parallelism and to the choice of timeout values for failure detection, due to resource contention and timing of events that influence the experiments. Therefore, we provide guidelines to tune the experiments using data from preliminary experiment executions, in order to achieve the best experimental throughput while preserving result accuracy.

Acknowledgments

This research has been supported in part by DFG GRK 1362, CASED, EC-SPRIDE, EC H2020 #644579, CECRIS FP7 (GA no. 324334), and SVEVIA MIUR (PON02 00485 3487758). We thank all contributors to the PAIN implementation, most notably Michael Tretter, Manuel Benz, and Alexander Hirsch. We dedicate this paper to the memory of Fabian Vogt, whose work on an earlier version of the PAIN framework has greatly influenced its current architecture.

REFERENCES

[1] J. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman, "Predicting How Badly "Good" Software Can Behave," *IEEE Softw.*, vol. 14, no. 4, pp. 73–83, 1997.

[2] J. Durães and H. Madeira, "Emulation of Software faults: A Field Data Study and a Practical Approach," *IEEE Trans. Softw. Eng.*, vol. 32, no. 11, pp. 849–867, 2006.

[3] D. Cotroneo and R. Natella, "Fault Injection for Software Certification," *IEEE Security Privacy*, vol. 11, no. 4, pp. 38–45, 2013.

[4] P. Koopman and J. DeVale, "The Exception Handling Effectiveness of POSIX Operating Systems," *IEEE Trans. Softw. Eng.*, vol. 26, no. 9, pp. 837–848, 2000.

[5] J. Arlat, J. Fabre, M. Rodríguez, and F. Salles, "Dependability of COTS Microkernel-Based Systems," *IEEE Trans. Comput.*, vol. 51, no. 2, pp. 138–163, 2002.

[6] D. Di Leo, F. Ayatolahi, B. Sangchoolie, J. Karlsson, and R. Johansson, "On the Impact of Hardware Faults–An Investigation of the Relationship between Workload Inputs and Failure Mode Distributions," in *Proc. SAFECOMP'12*, 2012, pp. 198–209.

[7] R. Natella, D. Cotroneo, J. Durães, and H. Madeira, "On Fault Representativeness of Software Fault Injection," *IEEE Trans. Softw. Eng.*, vol. 39, no. 1, pp. 80–96, Jan. 2013.

[8] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen, and D. Borthakur, "FATE and DESTINI: A Framework for Cloud Recovery Testing," in *Proc. NSDI'11*, 2011.

[9] P. Joshi, H. S. Gunawi, and K. Sen, "PREFAIL: A Programmable Tool for Multiple-failure Injection," in *Proc. OOPSLA'11*, 2011, pp. 171–188.

[10] A. Lanzaro, R. Natella, S. Winter, D. Cotroneo, and N. Suri, "An Empirical Study of Injected versus Actual Interface Errors," in *Proc. ISSTA'14*, 2014, pp. 397–408.

[11] S. Winter, M. Tretter, B. Sattler, and N. Suri, "simFI: From single to simultaneous software fault injections," in *Proc. DSN'13*, Jun. 2013, pp. 1–12.

[12] Y. Jia and M. Harman, "Higher Order Mutation Testing," *Information and Software Technology*, vol. 51, no. 10, pp. 1379–1393, 2009.

[13] M. Papadakis and N. Malevris, "An empirical evaluation of the first and second order mutation testing strategies," in *Proc. ICSTW'10*, 2010, pp. 90–99.

[14] Y. Jia and M. Harman, "Constructing Subtle Faults Using Higher Order Mutation Testing," in *Proc. SCAM'08*, Sep. 2008, pp. 249–258.

[15] A. Siami Namin, J. H. Andrews, and D. J. Murdoch, "Sufficient Mutation Operators for Measuring Test Effectiveness," in *Proc. ICSE'08*, 2008, pp. 351–360.

[16] A. Lastovetsky, "Parallel testing of distributed software," *Information and Software Technology*, vol. 47, no. 10, pp. 657–662, 2005.

[17] A. Duarte, W. Cirne, F. Brasileiro, and P. Machado, "GridUnit: Software Testing on the Grid," in *Proc. ICSE'06*, 2006, pp. 779–782.

[18] M. Oriol and F. Ullah, "YETI on the Cloud," in *Proc. ICSTW'10*, Apr. 2010, pp. 434–437.

[19] D. Cotroneo, M. Grottke, R. Natella, R. Pietrantuono, and K. S. Trivedi, "Fault Triggers in Open-Source Software: An Experience Report," in *Proc. ISSRE'13*, 2013, pp. 178–187.

[20] W. T. Ng and P. M. Chen, "The design and verification of the rio file cache," *IEEE Trans. Comput.*, vol. 50, no. 4, pp. 322–337, 2001.

[21] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the reliability of commodity operating systems," in *Proc. SOSP'03*, 2003, pp. 207–222.

[22] J. Durães and H. Madeira, "Multidimensional characterization of the impact of faulty drivers on the operating systems behavior," *IEICE Transactions on Information and Systems*, vol. 86, no. 12, pp. 2563–2570, 2003.

[23] J. Durães, M. Vieira, and H. Madeira, "Dependability Benchmarking of Web-Servers," in *Computer Safety, Reliability, and Security*, ser. Lecture Notes in Computer Science, vol. 3219, 2004, pp. 297–310.

[24] M. Vieira and H. Madeira, "A dependability benchmark for OLTP application environments," in *Proc. VLDB'03*, 2003, pp. 742–753.

[25] A. Albinet, J. Arlat, and J.-C. Fabre, "Characterization of the impact of faulty drivers on the robustness of the linux kernel," in *Proc. DSN'04*, 2004, pp. 867–876.

[26] A. Bondavalli, S. Chiaradonna, D. Cotroneo, and L. Romano, "Effective fault treatment for improving the dependability of COTS and legacy-based applications," *IEEE Trans. Dependable Secure Comput.*, vol. 1, no. 4, pp. 223–237, 2004.

[27] A. Bondavalli, A. Ceccarelli, L. Falai, and M. Vadursi, "Foundations of measurement theory applied to the evaluation of dependability attributes," in *Proc. DSN'07*, 2007, pp. 522–533.

[28] J. Carreira, H. Madeira, and J. G. Silva, "Xception: a technique for the experimental evaluation of dependability in modern computers," *IEEE Trans. Softw. Eng.*, vol. 24, no. 2, pp. 125–136, 1998.

[29] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson, "GOOFI: Generic Object-Oriented Fault Injection Tool," in *Proc. DSN'01*, 2001, pp. 83–88.

[30] D. Stott, B. Floering, Z. Kalbarczyk, and R. Iyer, "A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors," in *Proc. IPDS'00*, 2000, pp. 91–100.

[31] D. Skarin, R. Barbosa, and J. Karlsson, "Comparing and validating measurements of dependability attributes," in *Proc. EDCC'10*, 2010, pp. 3–12.

[32] E. van der Kouwe, C. Giuffrida, and A. S. Tanenbaum, "Evaluating Distortion in Fault Injection Experiments," in *Proc. HASE'14*, 2014.

[33] R. Chandra, R. M. Lefever, K. R. Joshi, M. Cukier, and W. H. Sanders, "A global-state-triggered fault injector for distributed system evaluation," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 7, pp. 593–605, 2004.

[34] D. Cotroneo, R. Natella, S. Russo, and F. Scippacercola, "State-Driven Testing of Distributed Systems," in *Proc. OPODIS'13*, 2013, pp. 114–128.

[35] I. Irrera, J. Durães, H. Madeira, and M. Vieira, "Assessing the Impact of Virtualization on the Generation of Failure Prediction Data," in *Proc. LADC'13*, 2013, pp. 92–97.

[36] E. Starkloff, "Designing a parallel, distributed test system," in *Proc. AUTOTESTCON'00*, 2000, pp. 564–567.

[37] G. M. Kapfhammer, "Automatically and Transparently Distributing the Execution of Regression Test Suites," in *Proc. ICTCS'01*, 2001.

[38] A. N. Duarte, W. Cirne, F. Brasileiro, and P. Duarte De Lima Machado, "Using the Computational Grid to Speed up Software Testing," in *Proc. SBES'05*, 2005.

[39] A. Duarte, G. Wagner, F. Brasileiro, and W. Cirne, "Multi-environment Software Testing on the Grid," in *Proc. PADTAD'06*, 2006, pp. 61–68.

[40] T. Parveen, S. Tilley, N. Daley, and P. Morales, "Towards a distributed execution framework for JUnit test cases," in *Proc. ICSM'09*, Sep. 2009, pp. 425–428.

[41] L. Yu, L. Zhang, H. Xiang, Y. Su, W. Zhao, and J. Zhu, "A Framework of Testing as a Service," in *Proc. MASS'09*, Sep. 2009, pp. 1–4.

[42] L. Yu, W.-T. Tsai, X. Chen, L. Liu, Y. Zhao, L. Tang, and W. Zhao, "Testing as a Service over Cloud," in *Proc. SOSE'10*, Jun. 2010, pp. 181–188.

[43] M. Staats and C. Păsăreanu, "Parallel Symbolic Execution for Structural Test Generation," in *Proc. ISSTA'10*, 2010, pp. 183–194.

[44] G. Candea, S. Bucur, and C. Zamfir, "Automated Software Testing as a Service," in *Proc. SoCC'10*, 2010, pp. 155–160.

[45] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea, "Cloud9: A Software Testing Service," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 4, pp. 5–10, Jan. 2010.

[46] R. Mahmood, N. Esfahani, T. Kacem, N. Mirzaei, S. Malek, and A. Stavrou, "A whitebox approach for automated security testing of Android applications on the cloud," in *Proc. AST'12*, 2012, pp. 22–28.

[47] J. Gray, "Why do computers stop and what can be done about it?" Tandem Computers, Tech. Rep. TR-85.7, 1986.

[48] M. Grottke and K. Trivedi, "Fighting Bugs: Remove, Retry, Replicate, and Rejuvenate," *IEEE Computer*, vol. 40, no. 2, pp. 107–109, 2007.

[49] T. Banzai, H. Koizumi, R. Kanbayashi, T. Imada, T. Hanawa, and M. Sato, "D-Cloud: Design of a Software Testing Environment for Reliable Distributed Systems Using Cloud Computing Technology," in *Proc. CCGRID'10*, 2010, pp. 631–636.

[50] T. Hanawa, T. Banzai, H. Koizumi, R. Kanbayashi, T. Imada, and M. Sato, "Large-Scale Software Testing Environment Using Cloud Computing Technology for Dependable Parallel and Distributed Systems," in *Proc. ICSTW'10*, Apr. 2010, pp. 428–433.

[51] R. Banabic and G. Candea, "Fast black-box testing of system recovery code," in *Proc. EuroSys'12*, 2012, pp. 281–294.

[52] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat, "Enforcing performance isolation across virtual machines in xen," in *Proc. Middleware'06*, 2006, pp. 342–362.

[53] G. Somani and S. Chaudhary, "Application performance isolation in virtualization," in *Proc. CLOUD'09*, 2009, pp. 41–48.

[54] Q. Huang and P. P. Lee, "An experimental study of cascading performance interference in a virtualized environment," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 4, pp. 43–52, 2013.

[55] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini, "DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments," in *Proc. USENIX ATC'13*, 2013, pp. 219–230.

[56] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An Empirical Study of Operating Systems Errors," in *Proc. SOSP'01*, 2001, pp. 73–88.

[57] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller, "Faults in linux: ten years later," in *Proc. ASPLOS'11*, 2011, pp. 305–318.

[58] D. Simpson, *Windows XP Embedded with Service Pack 1 Reliability*. [Online]. Available: http://msdn.microsoft.com/en-us/library/ms838661(WinEmbedded.5).aspx.

[59] A. Ganapathi, V. Ganapathi, and D. Patterson, "Windows XP Kernel Crash Analysis," in *Proc. LISA'06*, 2006, pp. 12–22.

[60] D. Cotroneo, R. Natella, and S. Russo, "Assessment and Improvement of Hang Detection in the Linux Operating System," in *Proc. SRDS'09*, Sep. 2009, pp. 288–294.

[61] A. Bovenzi, M. Cinque, D. Cotroneo, R. Natella, and G. Carrozza, "OS-Level Hang Detection in Complex Software Systems," *Int. J. Crit. Comput.-Based Syst.*, vol. 2, no. 3/4, pp. 352–377, Sep. 2011.

[62] Y. Zhu, Y. Li, J. Xue, T. Tan, J. Shi, Y. Shen, and C. Ma, "What Is System Hang and How to Handle It," in *Proc. ISSRE'12*, 2012, pp. 141–150.

[63] DEEDS/TUD and Mobilab/UniNa, *PAIN Software Framework*, https://github.com/DEEDS-TUD/PAIN.git.

[64] J. Christmansson and R. Chillarege, "Generation of an Error Set that Emulates Software Faults based on Field Data," in *FTCS*, 1996, pp. 304–313.

[65] Google Inc., *Android – Discover Android*. [Online]. Available: http://www.android.com/about/.

[66] ——, *android Git repositories*. [Online]. Available: https://android.googlesource.com/.

[67] ——, *Android Emulator*. [Online]. Available: http://developer.android.com/tools/help/emulator.html.

[68] F. Bellard, *Qemu*. [Online]. Available: http://wiki.qemu.org/Main_Page.

[69] R. Longbottom, *Roy Longbottom's Android Benchmark Apps*. [Online]. Available: http://www.roylongbottom.org.uk/android%20benchmarks.htm.

[70] Y. Benjamini and Y. Hochberg, "Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 57, no. 1, pp. 289–300, 1995.

[71] J. Andrews, L. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" In *Proc. ICSE'05*, 2005, pp. 402–411.

[72] H. Do and G. Rothermel, "On the use of mutation faults in empirical assessments of test case prioritization techniques," *IEEE Trans. Softw. Eng.*, pp. 733–752, 2006.

[73] J. Durães and H. Madeira, "Emulation of Software faults: A Field Data Study and a Practical Approach," *IEEE Trans. Softw. Eng.*, vol. 32, no. 11, pp. 849–867, 2006.

[74] R. Natella, D. Cotroneo, J. A. Durães, and H. S. Madeira, "On Fault Representativeness of Software Fault Injection," *IEEE Trans. Softw. Eng.*, vol. 39, no. 1, pp. 80–96, 2013.

[75] K. Kanoun and L. Spainhower, *Dependability Benchmarking for Computer Systems*. Wiley-IEEE Computer Society, 2008.

[76] T. Tsai, M. Hsueh, H. Zhao, Z. Kalbarczyk, and R. Iyer, "Stress-based and path-based fault injection," *IEEE Trans. on Computers*, vol. 48, no. 11, pp. 1183–1201, 1999.

[77] A. Avizienis, J. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Trans. Dependable Secure Comput.*, vol. 1, no. 1, pp. 11–33, 2004.