Towards a State-driven Workload Generation Framework for Dependability Assessment

Domenico Cotroneo, Francesco Fucci, Roberto Natella
Dipartimento di Informatica e Sistemistica
Università degli Studi di Napoli Federico II
Via Claudio 21, 80125, Naples, Italy
Email: {cotroneo,francesco.fucci,roberto.natella}@unina.it

***Abstract*—Assessing dependability of complex software systems through fault injection is an elaborate process, since their fault tolerance is influenced by the *state* in which they operate. This paper focuses on the definition of *state-driven workloads* in fault injection experiments, that is, workloads that bring a system in a given target state to be evaluated. We discuss a framework for the automated generation of state-driven workloads, and provide a preliminary evaluation in the context of the Linux 2.6 OS.**

***Keywords*-Fault Injection, Fault Tolerance, Stateful Systems**

I. Introduction

Fault injection, that is, the deliberate introduction of faults into a system, is an approach for providing evidences that a system is tolerant to faults in the environment and in the system itself. The increasing complexity of software systems makes fault injection an elaborate process, since these systems can operate in several different conditions, namely *states*, which have influence on their dependability. The importance of the state for dependability assessment purposes is emphasized by several studies on dependability assessment of stateful complex systems, such distributed filesystems [1], DBMSs [2], and multicast and group membership protocols [3], [4], [5]. In fact, when complex software systems are evaluated both *the activation and manifestation of faults*, as well as *the ability of the system to tolerate them*, depend on the system state, such as the current step of a numerical algorithm or of a distributed protocol [6], [7], [8]. For instance, this is the case of Mandelbugs [9], that is, a class of software faults that manifest themselves depending on the system state and on complex interactions with the hardware, the OS and other software in the system. Therefore, fault injection experiments have to be carefully planned by including the system state [10].

To take into account the state, a fault injection experiment should adopt an appropriate *workload*, which is the set of requests that are being submitted to the system when a fault is injected. A *state-driven workload*, i.e., a workload that brings the system in a given state during the experiment, is important to assure the significance and the efficiency of experiments, by *avoiding faults that do not actually manifest themselves* and by *covering every state in which fault tolerance mechanisms need to be tested*.

The generation of workloads for complex software systems is an open issue. Past studies proposed the generation of synthetic randomly-generated workloads (e.g., random CPU and I/O operations) according to a given distribution [11], [12], but this approach does not allow to bring a system into certain states that can be hard-to-reach, which therefore cannot be exercised and analyzed. Other fault injection approaches rely on hand-written workloads developed by testers [13], [14], [10], [15]. However, complex software systems are difficult to control since the relationship between the workload and states is complex and non-deterministic, due to the effect of random factors such as process scheduling and I/O delays. Therefore, it is still a difficult and time-consuming task to define a workload that brings the system in a desired state.

This paper is a first step towards the automated generation of state-driven workloads in complex software systems, which is an open research issue. We discuss a general framework for the purpose, and present a preliminary experiment on a complex system. The proposed framework is based on a closed-loop paradigm: a *workload generator* explores the space of possible workloads, and a positive feedback is provided to the workload generator if the system is approaching the target state, until the system converges to this state. The approach is fully automated, as it is only based on the feedback it receives from the controlled system, and does not rely on *a priori* knowledge about the relationship between workloads and states. The feasibility of the approach and its ability to reach a target state were evaluated in a preliminary case study, in which the workload generator is adopted to control the state of the Linux 2.6 SMP scheduler.

The paper is organized as follows. Section II discusses relevant fault injection techniques and tools for stateful software systems. Section III describes the proposed state-driven workload generation framework, which is further detailed in Section IV. Section V discusses a case study and some preliminary results. Section VI concludes the paper.

II. Related work

Dependability attributes of stateful systems have been studied in fault injection studies since a long time. For this reason, theoretical frameworks have been proposed in past works to make the fault injection process systematic. In [16], [17], [18], formal testing approaches for fault-tolerant systems are proposed, based on *state models* that describe the expected behavior with respect to normal inputs and faults (e.g., communication or memory faults). The model is used to generate

 ISBN: 978-1-61208-212-7

and monitor events for testing the actual implementation of the system, and to assess the coverage of the tests. The role of the system state was also emphasized by [19], in which faulty inputs are injected at the interface between drivers and the OS. It is showed that if faults are injected at different times during a sequence of interface function calls, then a higher number of vulnerabilities is discovered than injecting at the first occurrence of the target function call.

Tools have also been developed to aid testers in fault injection experiments in stateful systems. FTAPE [11], [12], a tool for injecting CPU, memory, and I/O faults in fault-tolerant computers, provides support for generating synthetic stressful workloads, based on the observation that a stressful workload is able to increase the percentage of injected faults that are activated and that propagate through the system, and thus are useful to assess the system response to faults. Faults are injected when the stress level of CPU, memory, or I/O (i.e., CPU utilization and memory and I/O throughput) are higher or lower than a threshold. To control the system state, FTAPE spawns a set of one or more processes that are CPU-intensive (by performing arithmetic operations), memory-intensive (by performing sequential reads and writes to large memory areas), and I/O-intensive (by repeatedly performing file opens, reads, writes, and closes), and lets testers to specify the distribution of CPU, memory, and I/O operations.

The ORCHESTRA tool [13], [20], aimed at testing distributed real-time systems, adopts a *script-driven probing* approach, in which messages exchanged by a process are intercepted and analyzed by a *protocol fault injection* layer (PFI) in the OS kernel, which identifies the current state of the protocol under test and corrupts/delays messages to perform fault injection. In order to track the protocol and to trigger fault injection, the PFI layer executes a *script program* provided by the tester, which describes the protocol under test using a state machine specification. In a similar way, the FCI fault injection framework for grid computing systems [15] provides a language specifically developed for specifying *fault injection scenarios*, which is used by the tester to describe *commands* to be sent to processes in the system (e.g., for stopping, halting, or resuming process execution) and *guard conditions* that trigger commands. NFTAPE [14] is a *portable* fault injection tool for distributed systems (i.e., it allows to easily customize fault injection for different systems and fault models), which introduced the concept of *LightWeight Fault Injector*, i.e., a small program running in the target system that is invoked by a remote controller to inject a fault, and that embeds the logic needed to implement a fault model for a given target platform.

The Loki tool [10] addressed the important problem of performing fault injection based on the *global state*, i.e., the condition that triggers fault injection is based on several nodes of the distributed system. Due to the problem of *clock* synchronization at each node and to message delays, a fault may be injected in a global state that is different than the desired target state. To mitigate this problem, Loki performs an off-line analysis of execution traces in order to discard experiments in which fault injection is likely to have been triggered in a wrong state.

A limitation of the tools mentioned above is that the workload has to be manually tuned in order to bring the system in a desired state. In the case of FTAPE, the tester selects the distribution of CPU, memory, and I/O operations generated by the synthetic workload. In the case of other tools, such as ORCHESTRA, FCI, NFTAPE, and Loki, the tester specifies a fault injection scenario by means of state machines, which are used by the fault injection tool to track the current state and trigger a fault when a desired state is reached. These approaches assume that the system is excited by a workload able to bring the system in the target state. However, devising such a workload is a tricky task for complex systems, since the tester has to carefully define the timing and the order of messages or inputs to be sent to the system. This problem is further complicated by the inherent non-determinism of complex systems, which makes difficult to bring the system in the target state and does not assure a correct execution of the fault injection experiment. This paper represents a further step towards solving this research problem.

III. Overview

Our framework for workload generation is composed by three subsystems, namely: (i) the System Under Test (SUT), which is the target of the experimental dependability evaluation, (ii) the Workload Generator (WG), which submits inputs to the SUT in order to provide a workload and monitors its behavior, and (iii) a Fault Injector (FI), which is adopted to inject faults into the SUT. The problem considered in this paper, namely *state-driven workload generation*, can be formulated as follows: *given an initial state s_0 of the SUT, and a goal state s_g in which an experiment has to be performed, the WG has to find a sequence of actions that drives the SUT from s_0 to s_g*. To this purpose, the WG and the SUT are put in a closed-loop configuration (shown in Fig. 1), in which the WG sends inputs to the SUT and collects information about its state. When the SUT is in the target state s_g, the WG triggers the FI, and then the fault injection experiment is performed.

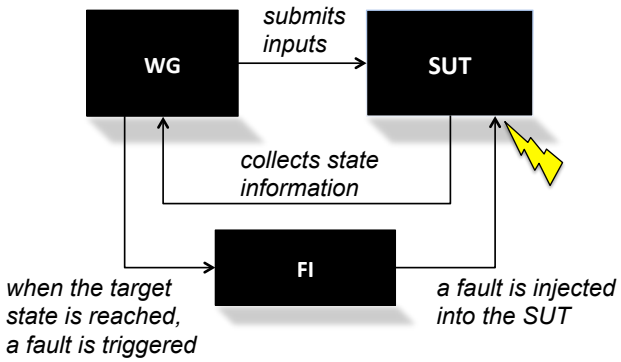

Fig. 1: Overview of the workload generation framework.

The WG evaluates how the SUT is behaving by computing a feedback function D based on state information. An increase

of the feedback value means that the action a_t has driven the SUT to a state "close" to the goal state; otherwise, the function decreases. Actions to be performed by the WG may consist in messages or commands sent to the SUT, or in variations in the rate or type of inputs generated by the WG. The WG aims to find a sequence of actions $s =< a_1 \dots a_t >\in S$ that brings the SUT *as close as possible to the goal state*, i.e., after performing the actions in s, D reaches a peak value at time t:

$$s = \arg\max_{\hat{s}\in S} D(t) \ . \quad (1)$$

An example of stateful system in which such a framework could be deployed is represented by a *File System*, since several past studies on fault injection and robustness testing have investigated dependability of File Systems with respect to faulty disks or applications [1], [21], [22]. In this kind of system, the state could include aspects related to I/O transactions, such as the amount of cached data that still has to be flushed to disk and the available disk bandwidth. A dependability analysis of a File System could aim to assess the probability of persistent data corruption due to faults, which in turn depends on the state of the system (e.g., the amount of cached data). In order to bring the File System in a given target state, the Workload Generator should perform I/O operations or instantiate new I/O-bound processes until the target is reached. Tuning a state-driven workload in such a scenario can be a tricky task due to complex interactions between the File System, other OS subsystems, user applications, and I/O devices, and automated workload generation is useful to relieve the tester of this task.

IV. The proposed framework

In this section, we will give a description of the framework focusing on its key elements: (i) state modeling, (ii) actions, (iii) reward function, (iv) search algorithm. The framework is meant to be tailored to the specific SUT, by using an appropriate definition of these elements. We discuss how these aspects should be defined by the tester, and will provide an example in the next section. These elements are orchestrated by the loop shown in Algorithm 1, which takes as inputs the target state s_g. Furthermore, the tester can choose a search algorithm to update the command sequence S, and a tolerance value ϵ that represents the stop condition of the search algorithm.

Algorithm 1 WGMainLoop(s_g,ϵ)

```
1: while D(s_g) − D(s_c) ≥ ε and T ≤ MAX_TIME do
2:     update the action sequence S
3:     apply the last action in sequence S on the SUT
4:     update the current state s_c
5: end while
```

A. State

The problem of the state definition is of paramount importance in our framework, and it is strictly dependent on the SUT. The framework is aimed at complex and "black-box" systems, for which a detailed knowledge about its internals is not available. Therefore, we consider as "state" a vector of *variables* that reflect the state of a subset *resources* or *data structures* in the system that are relevant for the dependability and performance of the SUT. Since several studies on field failure data have shown that fault activation and propagation is influenced by stress conditions [23], [24], the definition of state may include the usage of internal resources such as buffers, queues and communication channels, and performance measures such as the throughput of the system [25]. Moreover, the state can be defined based on the objectives of the fault injection and from the requirements of the system. For instance, in [9] fault injection is adopted in the context of a fault-tolerant distributed system based on a warm-replication mechanism to copy the state of a process to a backup replica: in this scenario, the evaluation of fault tolerance takes into account the *number of requests in the queues of the process*, which affects the amount of data that has to be copied to the backup replica and ultimately on the effectiveness of fault tolerance, and it is included in the state.

B. Actions

In general terms, actions are changes in the workload generated by the WG. They may consist in individual messages or commands sent to the SUT, or may represent parameters of a synthetic workload (as in the case of FTAPE [11]). The choice of the set of actions to adopt is dependent on the SUT, since the workload exercises the system through its interface to users and to other systems. Moreover, the definition of actions is also affected by the state definition, since the actions should be able to modify the state of the SUT. For instance, in the case of a web server, actions may change of the rate and kind of HTTP requests [26].

C. Feedback function

The WG selects actions to be performed on the basis of the feedback from the SUT, by means of the feedback function D. This function embeds the fact that a command that drives the system in a state s_t close to s_g gives a positive feedback; the function assumes its maximum value in $s_t = s_g$. The function should compute a scalar value that could be analyzed by the search algorithm, by accounting for the current value of state variables and their target value. For instance, if the WG aims to maximize the throughput of the system during the test, the function could return the current throughput. If more than one state variable is involved, the feedback function could compute a weighted sum of the factors that have to be tuned by the WG.

D. Search Algorithms

The core of the WG is based on search algorithms. In fact, the workload generation problem is an optimization problem in a discrete space, since the action space is typically discrete. It is known that this problem is NP-hard, therefore the search algorithm should be based on some kind of heuristic. The choice for the action to perform is based on the values returned

 ISBN: 978-1-61208-212-7

by the D function. If the last actions provided an increase of the D function (e.g., $D(s_t) - D(s_{t-1}) > 0$ for the action performed at time t), then the algorithm should take into account those actions to build the action sequence S; otherwise, the search algorithm could consider to discard that action in the sequence, and to try a new action. The tester can adopt well-known algorithms in the fields of combinatorial optimization and artificial intelligence, such as simulated annealing, genetic algorithms, and A*.

V. Evaluation

In this section we illustrate the proposed workload generation framework in the context of a case study and provide a preliminary evaluation. We consider a scenario of a fault injection campaign targeting a Linux-based system, in which experiments have to be performed when the CPU and I/O usage (which represents the state of the system) is equal to a given value. CPU and I/O are exercised by a synthetic workload consisting of CPU-bound and I/O-bound processes, which stress the system through the OS interface. Several fault injection studies were conducted in scenarios similar to ours [27], [11], [12]. In order to control CPU and I/O usage, the WG instantiates CPU-bound and I/O-bound processes, and tunes the number and type of processes using a search algorithm (see Section IV). We evaluate the ability of the proposed framework to find the best mix of CPU-bound and I/O-processes for reaching a desired level of CPU and I/O usage.

A. System under test

The SUT consists of the Linux OS running in a quad-cpu system. The state of this system (the average utilization of CPU and I/O) can be controlled by spawning synthetic processes that perform CPU- and I/O-intensive operations. However, it is tricky to tune the relative amount of CPU and I/O operations that brings the average CPU and I/O usage to a desired level, since there is a mutual relationship between CPU and I/O operations. In fact, I/O-bound processes often require to use the CPU for short time periods, in order to prepare I/O commands and to send or to retrieve data; therefore, CPU-bound processes may delay I/O-bound processes and affect I/O usage, and I/O-bound processes contribute to CPU usage. CPU and I/O usage is mainly affected by the *process scheduler* of the OS, which selects the order in which CPU- and I/O-bound processes are executed.

In order to evaluate the proposed framework, we adopted *Linsched* [28], a simulator of the Linux OS in multi-cpu environments. It is important to note that Linsched is based on the *actual source code* of the Linux kernel (v. 2.6.35), and that it allows to simulate process execution with high accuracy. In particular, Linsched includes the whole source code of the Linux process scheduler, namely the *Completely Fair Scheduler* (CFS) [29], and allows to evaluate the effect of process scheduling on CPU and I/O usage. Fig. 2 shows the scenario considered in our experiments, which consists of 4 CPUs and an I/O device each associated with a process queue. The system state is defined by two variables, that is, *the average number of processes in CPU queues (runqueues) and in the I/O queue* respectively. In order to control the system state, the WG generates CPU bound and I/O-bound processes, which are allocated to a CPU by a load-balancing algorithm in the Linux kernel, and each CPU is managed using the CFS algorithm. When a process performs an I/O operation, it is moved in the I/O queue, which is managed using a First-In-First-Out algorithm. The complex relationship between CPU and I/O usage can be seen in Fig. 3, which shows the average length of each queue as a function of the number of CPU bound and I/O-bound processes: these functions are non-linear, and where one of the functions increases, the other one decreases.

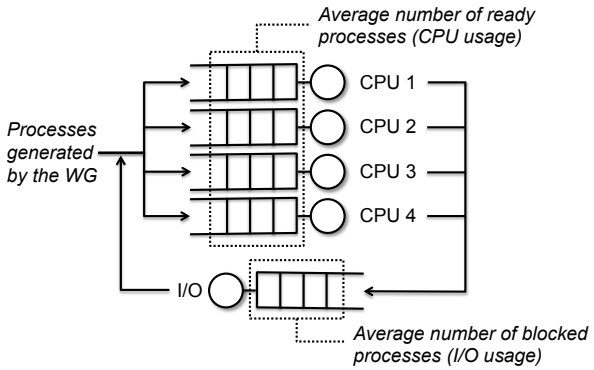

Fig. 2: Execution scenario.

B. Instantiating the WG framework

Following the proposed framework, we have defined the state s as a vector of two components: (i) the average number of the processes in the runqueues (ii) the average number of processes in I/O queue. The function D is defined as:

$$D(s_t) = -||s_t - s_g||$$

where $||.||$ is the euclidean norm between the state vectors s_t and s_g. The processes that can be generated by the WG are of three types, as shown in TABLE I. The first column is the process type, that can be CPU-bound, I/O-bound or Mixed; the second and the third column provide the average time that a process of each type spends in the *running state* (i.e., it is using the CPU) and in the *blocked state* (i.e., it is waiting for the completion of an I/O operation), respectively. The WG starts its execution using a process set filled with n_1 CPU-bound processes, n_2 I/O-bound processes, and n_3 Mixed processes (we assume $n_1 = n_2 = n_3 = 10$ as initial value) and we have set ϵ to 0.4. The actions of the WG consist in increasing or decreasing (by one, five, or ten) the number of processes of each type (n_1, n_2, and/or n_3). For every action a in the action set A, we refer to the action opposite to a as a', which removes the effects of a. In the case that such actions are not available or applicable for the target system, the WG should at each iteration (i) reset the system state, and (ii) apply the whole sequence S. Finally, we have chosen the Simulated

 ISBN: 978-1-61208-212-7

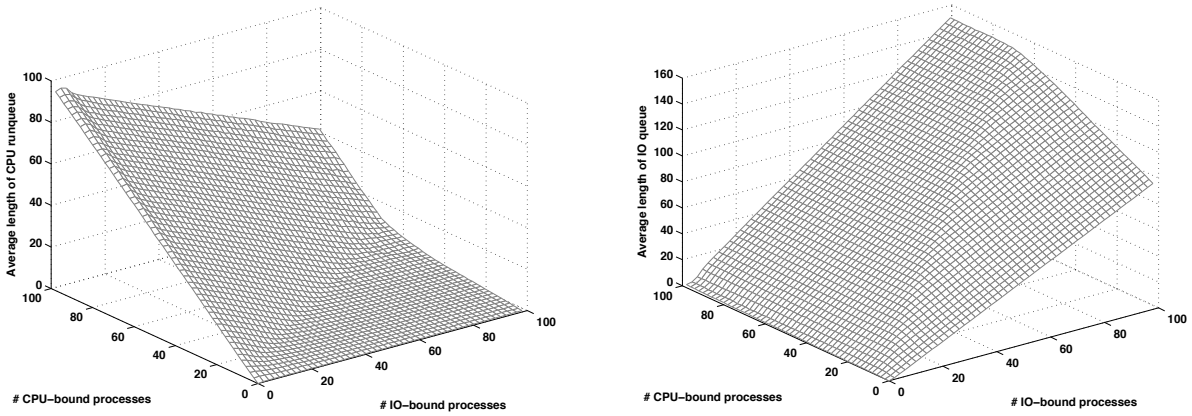

Fig. 3: Average length of CPU and I/O queues in function of the number of CPU- and I/O-bound processes in the system.

Annealing [30] as search algorithm (Algorithm 2), which is executed in the context of the WG main loop (Algorithm 1). It randomly chooses an action, and evaluates D in the new state s_n. If the action produced an increase of D compared to the previous state, then the action is appended to the action sequence. If D decreases, then the action is not included in the sequence and the system is reverted to its previous state using the action a'. However, the algorithm may still include action a in the sequence and remain in the state s_n (even if D decreased) with probability $e^{-\Delta E/T}$ (decreasing with time).

Algorithm 2 SIMULATEDANNEALING(s_c)

1: $a \leftarrow random\_select(A)$
2: do a on SUT and get the next state s_n
3: $\Delta E \leftarrow R(s_n) \leftarrow D(s_n) - D(s_c)$
4: **if** $\Delta E < 0$ **and** $random(1 - e^{-\Delta E/T})$ **then**
5: do a' on SUT
6: **end if**

TABLE I: Process types instantiated by the WG.

Type	Mean CPU time	Mean I/O time
CPU-bound processes	200 ticks	20 ticks
I/O-bound processes	5 ticks	20 ticks
Mixed processes	20 ticks	20 ticks

C. Experimental results

In our experiments, we evaluated the framework using 9 different goal states, by selecting different realistic conditions in which the SUT could be operating. The target values for the average length of CPU queues were respectively $low = 10$, $medium = 50$, and $high = 100$. In a similar way, the target values for the average length of the I/O queue were respectively $low = 10$, $medium = 50$, and $high = 100$. At every iteration of Algorithm 1, the system is simulated for 10 seconds, and the state of the system is collected by computing the average number of ready and blocked processes. We evaluate the number of iterations for reaching the goal state.

In every experiment, the WG reached the target state. The distance from the target state approaches to zero in at most 250 iterations, as shown in Fig. 4. The worst case is represented by the target state $(100, 100)$, which is the state farther from the initial state (i.e., the state obtained using the initial mix of processes n_1, n_2, and n_3). It could possible to improve the speed of convergence using a different initial mix of processes. For instance, a tester could consider a set of several random process mixes, and the Workload Generator can choose as initial mix the one that is closer in the state space to the target state. It is important to note that this is a general problem of search and optimization algorithms. Figure 5 shows how the search algorithm moves towards the target state. The oscillations are due to the random choices made by the search algorithm, which in the long term brings the system in the target state (the filled dot in the figure).

VI. CONCLUSION

In this paper, we discussed a framework for automatically generating a workload to reach a target state defined by the tester. In this framework, a Workload Generator interacts with the System Under Test in a closed loop, to iteratively search for a sequence of actions that brings the System Under Test in the target state. The framework has been evaluated in a real complex system, namely the Linux 2.6 OS scheduler. From the results, we found that the Workload Generator is able to reach the target state in every experiment in a reasonable number of iterations. Future work in this area can be focused on the application of the approach to other systems, with the aim of evaluating and improving its effectiveness and portability. Another area worth of investigation is the adoption of the framework in the context of dependability benchmarking.

ACKNOWLEDGMENT

This work has been supported by the projects "Embedded Systems in Critical Domains" (CUP B25B09000100007)

 ISBN: 978-1-61208-212-7

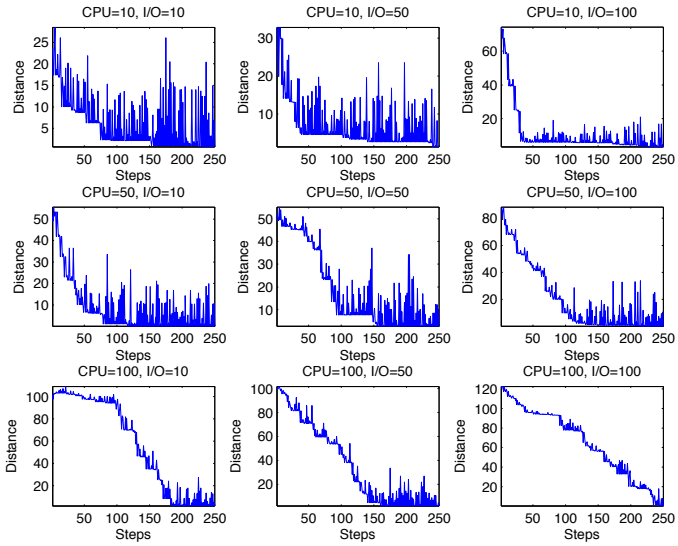

Fig. 4: Speed of convergence of the Workload Generator.

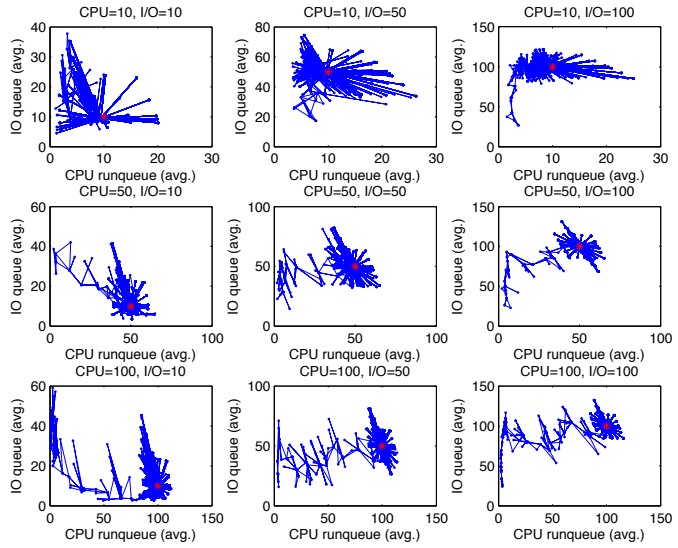

Fig. 5: State space explored by the Workload Generator.

within the framework of ”POR Campania FSE 2007-2013” and ”Iniziativa Software CINI-Finmeccanica”.

REFERENCES

[1] R. Lefever, M. Cukier, and W. Sanders, “An experimental evaluation of correlated network partitions in the Coda distributed file system,” in *Proc. Intl. Symp. Reliable Distributed Systems*, 2003, pp. 273–282.

[2] M. Vieira and H. Madeira, “A dependability benchmark for OLTP application environments,” in *Proc. 29th Intl. Conf. on Very Large Data Bases*, 2003, pp. 742–753.

[3] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J. Fabre, J. Laprie, E. Martins, and D. Powell, “Fault injection for dependability validation: A methodology and some applications,” *IEEE Trans. Software Eng.*, vol. 16, no. 2, pp. 166–182, 1990.

[4] K. Joshi, M. Cukier, and W. Sanders, “Experimental evaluation of the unavailability induced by a group membership protocol,” *Proc. European Dependable Computing Conf.*, pp. 644–648, 2002.

[5] B. Helvik, H. Meling, and A. Montresor, “An approach to experimentally obtain service dependability characteristics of the Jgroup/ARM system,” *Proc. European Dependable Computing Conf.*, pp. 179–198, 2005.

[6] E. Czeck and D. Siewiorek, “Observations on the Effects of Fault Manifestation as a Function of Workload,” *IEEE Trans. Computers*, vol. 41, no. 5, pp. 559–566, 1992.

[7] R. Chillarege and R. Iyer, “An experimental study of memory fault latency,” *IEEE Trans. Computers*, vol. 38, no. 6, pp. 869–874, 1989.

[8] J. Meyer and L. Wei, “Analysis of workload influence on dependability,” in *Proc. Intl. Fault-Tolerant Computing Symp.*, 1988, pp. 84–89.

[9] R. Natella and D. Cotroneo, “Emulation of transient software faults for dependability assessment: A case study,” in *Proc. European Dependable Computing Conf.*, 2010, pp. 23–32.

[10] R. Chandra, R. Lefever, K. Joshi, M. Cukier, and W. Sanders, “A Global-State-Triggered Fault Injector for Distributed System Evaluation,” *IEEE Trans. Parallel and Distributed Sys.*, vol. 15, no. 7, pp. 593–605, 2004.

[11] T. Tsai and R. Iyer, “Measuring fault tolerance with the FTAPE fault injection tool,” *Quantitative Evaluation of Computing and Communication Systems*, pp. 26–40, 1995.

[12] T. Tsai, M. Hsueh, H. Zhao, Z. Kalbarczyk, and R. Iyer, “Stress-Based and Path-Based Fault Injection,” *IEEE Trans. Computers*, vol. 48, no. 11, pp. 1183–1201, 1999.

[13] S. Dawson, F. Jahanian, T. Mitton, and T. Tung, “Testing of fault-tolerant and real-time distributed systems via protocol fault injection,” in *Proc. Fault Tolerant Computing Symp.*, 1996, pp. 404–414.

[14] D. Stott, B. Floering, D. Burke, Z. Kalbarczpk, and R. Iyer, “Nftape: a framework for assessing dependability in distributed systems with lightweight fault injectors,” in *Proc. IEEE Intl. Computer Performance and Dependability Symp.*, 2000, pp. 91–100.

[15] W. Hoarau and S. Tixeuil, “A language-driven tool for fault injection in distributed systems,” in *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, 2005, pp. 194–201.

[16] D. Avresky, J. Arlat, J. Laprie, and Y. Crouzet, “Fault Injection for Formal Testing of Fault Tolerance,” *IEEE Trans. Reliability*, vol. 45, no. 3, pp. 443–455, 1996.

[17] A. Arazo and Y. Crouzet, “Formal Guides for Experimentally Verifying Complex Software-Implemented Fault Tolerance Mechanisms,” in *Proc. Intl. Conf. on Eng. of Complex Computer Systems*, 2001, pp. 69–79.

[18] A. Ambrosio, F. Mattiello-Francisco, V. Santiago, W. Silva, and E. Martins, “Designing Fault Injection Experiments Using State-Based Model to Test a Space Software,” *Lecture Notes in Comp. Science*, vol. 4746, pp. 170–178, 2007.

[19] A. Johansson, N. Suri, and B. Murphy, “On the Impact of Injection Triggers for OS Robustness Evaluation,” in *The 18th Intl. Symp. on Software Reliability Eng.*, 2007, pp. 127–136.

[20] S. Dawson, F. Jahanian, and T. Mitton, “Experiments on six commercial TCP implementations using a software fault injection tool,” *Software Practice and Experience*, vol. 27, no. 12, pp. 1385–1410, 1997.

[21] V. Prabhakaran, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, “Model-based failure analysis of journaling file systems,” in *Proc. Intl. Conf. Dependable Systems and Networks*, 2005, pp. 802–811.

[22] D. Cotroneo, D. Di Leo, R. Natella, and R. Pietrantuono, “A case study on state-based robustness testing of an operating system for the avionic domain,” *Lecture Notes in Comp. Science*, vol. 6894, pp. 213–227, 2011.

[23] M. Sullivan and R. Chillarege, “Software Defects and their Impact on System Availability: A Study of Field Failures in Operating Systems,” in *Proc. Intl. Fault-Tolerant Computing Symp.*, 1991, pp. 2–9.

[24] I. Lee and R. Iyer, “Software dependability in the Tandem GUARDIAN system,” *IEEE Trans. Software Eng.*, vol. 21, no. 5, pp. 455–467, 1995.

[25] R. Jain, *The art of computer systems performance analysis*. John Wiley & Sons, 2008.

[26] R. Matias *et al.*, “An experimental study on software aging and rejuvenation in web servers,” in *Proc. Computer Software and Applications Conf.*, vol. 1, 2006, pp. 189–196.

[27] W.-I. Kao, R. Iyer, and D. Tang, “FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior under Faults,” *IEEE Trans. Software Eng.*, vol. 19, no. 11, pp. 1105–1118, 1993.

[28] J. Calandrino, D. Baumberger, T. Li, J. Young, and S. Hahn, “Linsched: The linux scheduler simulator,” in *Proc. Intl. Conf. on Parallel and Distributed Comp. and Comm. Systems*, 2008, pp. 171–176.

[29] C. Wong, I. Tan, R. Kumari, and F. Wey, “Towards achieving fairness in the linux scheduler,” *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 34–43, 2008.

[30] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi, “Optimization by simulated annealing,” *Science*, vol. 220, pp. 671–680, 1983.

 ISBN: 978-1-61208-212-7