Fault Injection Framework for System Resilience Evaluation

Fake Faults for Finding Future Failures

Thomas Naughton, Wesley Bland[*], Geoffroy Vallée, Christian Engelmann, and Stephen L. Scott[†]
Oak Ridge National Laboratory
Computer Science and Mathematics Division
Oak Ridge, TN 37831, USA
{naughtont,blandwb,valleegr,engelmannc,scottsl}@ornl.gov

ABSTRACT

As high-performance computing (HPC) systems increase in size and complexity they become more difficult to manage. The enormous component counts associated with these large systems lead to significant challenges in system reliability and availability. This in turn is driving research into the resilience of large scale systems, which seeks to curb the effects of increased failures at large scales by masking the inevitable faults in these systems. The basic premise being that failure must be accepted as a reality of large scale system and coped with accordingly through system resilience.

A key component in the development and evaluation of system resilience techniques is having a means to conduct controlled experiments. A common method for performing such experiments is to generate synthetic faults and study the resulting effects. In this paper we discuss the motivation and our initial use of software fault injection to support the evaluation of resilience for HPC systems. We mention background and related work in the area and discuss the design of a tool to aid in fault injection experiments for both user-space (application-level) and system-level failures.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Reliability*; C.4 [**Computer Systems Organization**]: Performance of Systems—*Fault tolerance*

General Terms

Experimentation, Reliability

Keywords

Fault injection, Resilience

[*]University of Tennessee; Electrical Engineering and Computer Science Department; Knoxville, TN 37996, USA.

[†]This work was supported by the U.S. Department of Energy, under Contract DE-AC05-00OR22725.

Resilience'09, June 9, 2009, Munich, Germany.

1. INTRODUCTION

Errors are an un-avoidable aspect of high-performance computing systems, and their management accounts for a great deal of effort at all levels of the execution environment. While errors occur they are only significant if they interrupt productive work, i.e., produce a failure. The goal of resilient systems is to mask these failures to enable applications to continue execution at an acceptable level.

Currently, there is much discussion surrounding the resilience of large scale systems. A central facet being how to detect and react to the enormous increase in errors that such large systems face. What errors occur and how they ultimately manifest as failures in the system is a topic of extensive investigation. Many projects are looking at log analysis to identify faults in order to develop prediction techniques, which might be used to avoid future failures or help identify the root causes of failures. However, several basic questions persist, like "what is failing?" and "how well does the system mask failures from applications?" As the systems increase in size, a systematic method for evaluating how the overall system copes with failures is a crucial component in resilience research.

The use of software based fault-injection provides such a vehicle for experimentation related to fault-tolerance, to evaluate the level of detection and recovery supported by current software environments used on the HPC platforms. This facility provides a methods to perform controlled experimentation. For example, to evaluate the effectiveness of current failure prediction and failure mining techniques. The end objective being to gain a better understanding of how systems are coping with known issues. The approach allows for iterative improvement into the quality of the simulated faults, i.e., fidelity with respect to real-world errors. This synthetic approach to failures will provide a basis for looking at existing detection and reaction techniques, some of which may not currently be employed in HPC environments, i.e., borrowing from other domains like embedded/portable systems [2].

In this paper, we present preliminary work on the use of software fault injection to aid research in resilient HPC systems. We provide a brief overview of techniques and highlight prior work that has made use of fault injection. We also discuss our current efforts in the creation of a tool to aid in fault injection experiments for HPC systems. The basic architecture is given and the associated criteria/requirements.

2. BACKGROUND

Fault injection is the purposeful introduction of faults (or errors) into a target. These intentional faults can be introduced through hardware or software methods [5]. The hardware based approach obviously requires specialized equipment to bother introduce the fault, and to detect the effects. In a software based approach, or software implemented fault injection (SWIFI), there is more flexibility in terms of how to implement and detect the faults. However, the faults are limited in scope to those that are accessible via software [5]. For example, radiation induced memory "soft errors" can be injected via hardware/environmental approaches but can only be simulated through software by techniques like bit-flips, etc.

The distinction between faults, errors, and failures is fundamental to work on fault-injection. In [1] this relationship in clarified, and is paraphrased here:. a *fault* is a defect that exists in a service but may be "active" or "dormant"; the *error* occurs when a fault becomes active; if the error is not suppressed and becomes visible outside the service a *failure* occurs. Therefore, for fault-injection experiments we will identify what services to perturb in order to evaluate the resilience of the service, and overall system, to failures.

3. FAULT INJECTION FRAMEWORK

Despite the common use of fault injection techniques for testing and evaluation, there exist very few publicly available open source tools. To aid in our resilience studies, we have begun development of a prototype to aid in our fault injection experiments. Our approach is guided by the following criteria:

- *simplicity* – easy to setup, define, and perform fault injection experiments
- *versatility* – support experiments at various levels of software stack (user- / kernel-level)
- *reproducibility* – framework should allow for reproducible experiments [1]
- *distributed environments* – support for experiments on both local and remote nodes; and physical and virtual machines

The expectation is that the underlying mechanisms for performing the fault injection and detection will be abstracted by the framework. This allows users to focus on the experiments without having to get deeply involved with the low-level mechanisms. This separation also allows for different injection/detection techniques to be explored at the lower levels.

In this section, we present our framework for fault injection. Section 3.1 presents the motivation for this work and Section 3.2 presents the actual architecture of the proposed framework.

3.1 Motivation

The fault injection framework aims at enabling testing of fault tolerance and system resilience mechanisms and policies, as well as the evaluation of fault reporting/detection mechanisms.

[1]Note, some injection techniques may be non-deterministic, e.g., timeouts, and therefore non-reproducible but the framework proper should be consistent.

Therefore the notion of "experimentation" is central to our framework. This notion allows one to define the context for the execution of the framework, how to instantiate the different framework's components, and how to run those components. For that, an experiment is based on different components: "fault injectors" which actually inject the failure into the system; "fault detectors" which try to detect a given type of failure; and "analyzers" which get information from detectors and analyze it (detectors are simplistic components that do not include any policy, they for instance may just parse system logs). In addition to those mechanisms, the framework is also composed by a description of the experimentation, i.e., the policy to drive the experimentation.

Therefore, the framework should allow for the startup of experiments that couple a "fault injector" with an appropriate "fault detector". An "analyzer" component is used to encode policies about given fault scenarios, to keep the injectors and detectors more general purpose. The framework will provide the minimum functionality to support experiment startup/shutdown and common supporting services, e.g., interprocess communication (IPC) setup, logging.

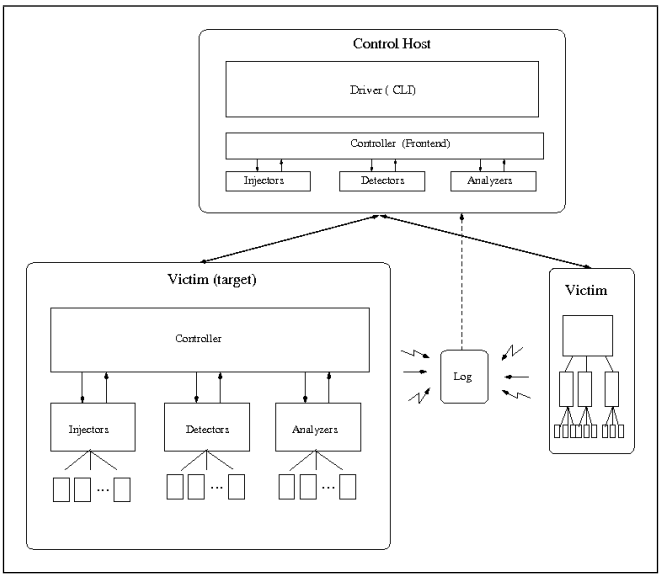

Figure 1: High-level diagram of the components and architecture.

3.2 Architecture

Injectors.

"Injectors" are the components that actually create a failure on a given system. Injectors can be local and/or remote; the remote injectors having the capability to limit the impact of failure on the injector itself.

An injector is typically composed of two parts: a backend that actually creates the fault (e.g., kill a process) and a frontend that creates the interface with the framework.

For instance, to avoid fault impacts on the injector, one may decide to use `ssh` and remotely trigger faults. In that case, the backend is actually the `ssh` daemon on the node where the fault has to be injected (and calls a local command such as `killall my_app`) and the frontend remotely

executes the `ssh` command.

Detectors.

"Detectors" are generally coupled with injectors: while an injector actually creates a fault, the detector attempts to discover the existence of the fault and report it. Detectors are typically an abstraction of the technical details regarding the detection of a specific fault.

Analyzers.

"Analyzers" are responsible for collating the information and processing into a useful form. The analyzer can be used to interpret the events associated with a given detector/injector configuration. The coupling of analyzers with detectors/injectors is managed by a framework configuration file, "finject.conf". This file specifies the acceptable component tuples supported by the framework, which are selected based on information given in the experiment input records.

Controller.

The controller manages the life-cycle of all components instantiated for a given experiment (e.g., creation, termination). It makes sure that all components are correctly created and setup so an experiment can actually run.

Fault Type.

It has been indicated in the literature that the "fault type" will influence the detection mechanism. Therefore, it is expected that detectors and injectors will be coupled and an appropriate analyzer will be used to evaluate the results.

Examples of fault types are:

```
[processor|memory|communication|device] fault
```

Experiments also include various "arguments" and "flags" that are passed to the components related to the given fault type. For example, the following arguments are used to specify a target (victim) application and memory address for an experiment.

```
app='loopnest-forever',addr='0xbfef4000'
finject='user-memory'
```

Fault flags are typically optional, which could be passed to the analyzer for "hints" as to the expected behavior for a given experiment. Examples of flags are

```
victim-status=[offline|online]
finject=[user-memory|kern-memory]
sudo='/usr/bin/sudo'
```

Fault Mode.

We identified 3 fault modes: (i) permanent – will persist throughout remainder of FI session; (ii) transient – singular (one time) faults; (iii) intermittent – repeating faults (e.g., occurrence of 50%).

Examples of fault modes are:

```
[permanent|transient|intermittent] fault
```

Monitors.

A passive service that can be used by detectors.

Victim.

The "victim" is the target of the fault to inject. The value of this field is used to contact the target. Note, the target can be the local machine or the hostname of a remote physical machine (or virtual machine). Examples of victims are:

```
[localhost|node13|ubuntu-vm]
```

Driver.

The "driver" is the interface between the user and the framework. The driver actually interprets the experiment description (i.e., the policy for the experimentation) and interacts with the controller to drive the experiment.

Currently, the driver is designed to use an input file which describes the experiment. The format for the experiments file is currently just a flat file with line oriented records. The record fields are separated by colons (':'). In some cases, the field may contain multiple options that are expressed in a key / value pair (key=value). Lines beginning with a pound symbol ('#') are treated as comments and ignored, as are blank lines. We are currently experimenting with different injection mechanisms and expect the format to evolve but the current structure is simple and easy to extend. The following shows a sample experiments file[2],

```
#
# finject experiments
#
# Format:
#   fault_type:fault_mode:fault_args:victim:flags
#

memory:intermittent:app='/tmp/fileptr': \
   ubuntu-vm:finject='kern-memory',dargs='50'

register:permanent:app='/tmp/loopnest-forever': \
   localhost:finject='user-memory'
```

Additionally, there is a framework configuration file, *finject.conf*, that is used to group compatible injectors-detectors-analyzers. This determines what backend modules are to be used for the experiments. The following shows an example entry from this file.

```
[user-memory]
    injector=injectors/frob-reg-injector
    detector=detectors/child-watcher
    analyzer=analyzers/basic-counter
```

4. LINUX – MEMORY FAULT INJECTION

Our initial investigation, in preparation for the proposed framework, used the Linux kernel-level fault injection facility [8] to introduce memory errors into the system. Here we describe the relevant aspects and components. The individual components used to drive this initial work, will be used as the the injectors and detectors in our current prototype.

4.1 Memory in Linux

In the Linux kernel, memory can be allocated in two main ways. Either the kernel can request memory in pages, which is the main way the Linux kernel handles memory, or by making more fine-grained requests via the slab layer. When requesting pages, the kernel can only get memory in increments of the page size for that particular architecture. In order to pick a different sized chunk of memory, it needs

[2]Note, due to column widths restrictions in the paper, newlines have been inserted into the example.

to call some sort of kernel space malloc call (`kmalloc()` or `vmalloc()`). These requests eventually result in either a page being requested or memory being pulled off of the slab. The slab layer is used to cache freed memory from the kernel to make requesting the same types of objects later much faster[3]. This keeps the kernel from having to allocate and set up memory every time it is requested and facilitates faster memory allocations. Both of these (slab and pages) have injection points for the Linux fault injection framework that are discussed in Section 4.2.

4.2 Linux Fault Injection

Starting with version 2.6.20, the Linux kernel includes a fault injection (FI) framework that can be used to inject faults into memory and block devices. As of version 2.6.25.7, the framework could inject three different types of errors: slab errors, page allocation errors, and disk IO errors. For our purposes, we focussed on the memory errors and specifically the slab errors.

We used this Linux based framework to begin our studies into system-level fault injectors. We chose this as a starting point because of Linux's wide-spread use and its ties to virtualization solutions like Xen. The Linux FI framework is extensible and supports runtime configuration. Also, since it is at the operating system level, the faults can be directed at both user-space (application-level) and system-level targets. To enable the Linux FI framework, there are a set of files in the `/debug` file-system to direct the kernel to the location where faults should be inserted. This can be automated within an application or through a script. Since Linux is open-source the FI mechanisms can be modified and enhanced for specific points of interest. Therefore, future work can extend the kernel to support new fault injection capabilities such as network IO errors, which can then be leveraged as part of our experiment based framework.

4.3 Mechanisms for Detecting Failures

Once the faults are injected, there must be methods to reliably detect these faults using methods that can be seen at the user level. Currently, these methods typically include using output from standard error and checking the return codes of the applications. These are simple, easy to implement methods of checking for problems that might be occurring in the application which can be extended later to include more complex detection mechanisms, e.g., analyzing system logs [9]. The current detection mechanisms are purposely simplistic to help with the development and testing of the injection framework.

4.4 Implementation

The fault injection and failure detection experiments, that are based on the Linux in-kernel framework, are currently implemented as a small application called "watcher". The "watcher" performs all the tasks necessary to create and detect faults. First, it sets up the fault injection environment by setting the appropriate values in the `/debug` file-system used by the Linux kernel fault injection mechanisms. Next it launches, the application that it will be monitoring and duplicates its file descriptors for `stderr` and `stdout` so it can monitor the output from the program. It also begins to monitor the system logs in `/var/log/messages` so it can see when faults are injected. Then the watcher turns on the Linux kernel fault injection and waits for output from the program. At some point, the child process will complete and return a value which the watcher will capture and report on. Throughout this process the watcher is tracking how many faults are injected and how many of those faults became application detectable failures. Note, this single application approach was somewhat cumbersome and is broken apart to fit into our new experiment based framework, 'finject', which is discussed in Section 5.

[3]A list of slab object types and associated statistics can be obtained from `/proc/slabinfo`.

5. EVALUATION

Based on the architecture discussed in Section 3, an initial framework prototype has been developed called *finject*. While the implementation is ongoing, a preliminary evaluation was performed using memory/register based fault injection experiments. One that performs fault injection via an entirely user-level interface, i.e., `ptrace(2)`. The second is an adaptation of our prior experiment using the Linux in-kernel fault injection facility. In both cases the target (victim) is a user-space process.

The various components of the framework (Section 3) are implemented as Perl modules. The actual backend executable associated with a given component, e.g., *Injector Component*, is referred to as a "module". For instance, the *Analyzer Component* will select an *Analyzer Module*. In the below discussion, a general description is provided for the various component/modules used in the experiments. Note, to simplify the discussion this distinction between component and module is largely ignored.

The general flow of an experiment is as follows:

1. Driver: reads and processes list of experiments
2. Driver: invokes Controller with an experiment
3. Controller: reads framework configuration (policy) settings
4. Controller: redirects STDERR for children
5. Controller: starts Analyzer
6. Analyzer: routes Detector/Injector STDOUT to Analyzer STDIN
7. Analyzer: starts Detector
8. Detector: starts victim App, watches/reports to Analyzer
9. Analyzer: starts Injector
10. Injector: victimizes App, reports to Analyzer
11. Analyzer: waits on Detector/Injector
12. Analyzer: sends results to Controller
13. Controller: prints results and returns to Driver

Field	Value	Description
Count (victims)	100	Number of victim application instances
Total (injections)	2197	Number of injected failures for all runs
Minimum	1	Number of injections to victim failure
Maximum	98	Number of injections to victim failure
Mean	21.97	Number of injections to victim failure
Median	17	Number of injections to victim failure
Mode	4	Number of injections to victim failure
Std.Dev.	21.419	Number of injections to victim failure

Table 1: Statistics associated with Experiment-I (register bit-flip)

5.1 Experiment I

This experiment introduced faults into a user-space application via the `ptrace(2)` system call. The errors were in the form of register bit-flips and resulted in application failure at a varied frequency rate. In the tests, an arbitrary max of 100 injected failures was set, with application failures occurring at varied points (see Table 1). The experiment used a basic analyzer to count the number of faults injected and failures detected and reported the results to the controller for display.

Note, in earlier experiments the target application did a bit of work and then called a sleep function. As we began injecting faults we noticed that there were never errors. After further investigation, we realized that the majority of the time was spent in these sleep routines, which were not effected by our fault injection method (register bit-flip). Therefore, in later tests all sleep statements were removed from the target application.

The three components and target (victim) application used for Experiment I are as follows:

target – 'loopnest-forever' simply generates work forever, using a shallow hierarchy of calling sub-functions in order to have items pushed on/off the stack and return values that are used in the output data.

analyzer – 'basic-counter' which takes input from STDIN and counts occurrences of input from the injector and detector. Upon completion it returns the results to the controller which displays them to the user.

detector – 'child-watcher' this is a basic program that starts a given application and watches the return status to see if the application exited with a non-zero status and/or signal. Upon detection of such an event, it prints to STDOUT, which is routed to the analyzer.

injector – 'frob-reg-injector' this is a simplistic program that flips a bit in a register for a given application via the `ptrace(2)` interface. Upon injection of a fault, it prints to STDOUT, which is routed to the analyzer. The program stops when either the target (PID) no longer exists, or a max number of tries is reached.

5.2 Experiment II

The second experiment used the Linux fault injection feature [8] as described in Section 4.2. Here the earlier single "watcher" program was split into separate pieces and re-run using the FInject framework. In this scenario, we focus on exercising slab based memory failures. The victim application opens a temporary file using the `tmpfile()` function, which generates a unique filename and opens a file handle of type `FILE*`, e.g., `fopen()`. As described in Section 4.1, the slab layer maintains a cache of memory objects. In this experiment we focused on failures to the 'filp' objects, which are associated with `FILE*` memory allocations. Therefore, when the victim application was run with a probability of slab failures set to 100%, all calls to `tmpfile()` returned an error (NULL). Note, we also confirmed the case of a 50% failure rate by adjusting the Linux-FI parameters, i.e., with probability=50% and times=25, for $N = 50$, 25 failures occurred.

The three components and target (victim) application used for Experiment II are as follows:

target – 'fileptr' is a program that tries to create N temporary file, with an associated file pointer object (`FILE*`) using the `tmpfile()` function. Upon file create/open failure, a formatted error message is printed to STDOUT.

analyzer – 'basic-counter' which takes input from STDIN and counts occurrences of input from the injector and detector. Upon completion it returns the results to the controller which displays them to the user.

detector – 'fileptr-watcher' starts the child process, watching the STDOUT and return code. The child's STDOUT is scanned for a sentinel that is used to signal failure events. The parent detects these events, and reports them to the analyzer.

injector – 'linux-fi-injector' scan for occurrences of Linux-FI occurrences and prints formatted messages to analyzer. In current configuration, this module just reports on actual kernel generated fault injections, i.e., kernel is the actual injector.

6. RELATED WORK

There has been a great deal of work into fault injection with varied approaches. A review of fault injection techniques and significant tools is available in [5]. The Xception [7] project leveraged hardware supported debugging and performance monitoring capabilities to implement efficient fault injectors and detectors. e.g., breakpoint registers. In [12] a general purpose tool was developed to support dependability studies to include distributed environments.

The FAUmachine [10] project includes a full hardware simulator (virtual machine) and supports the injection of faults into the disk (read/write errors), network (packet loss) and bit-flips to CPU registers and memory. The FAUmachine project's virtual machine is similar to the more widely

known user-mode Linux (UML) virtual machine, where the Linux kernel runs as a user-level process.

In [2], modifications were made to the QEMU CPU emulator to support injection of faults for the purpose of evaluating research on self-healing operating system mechanisms. The fault injection utility was primarily targeted at the ARM architecture.

In [6], the effects of injecting faults into Linux system calls were studied. This included sending invalid or corrupted (bit-flipped) arguments to system calls, and corruption of the arguments for the internal kernel functions. They indicated that the bit-flip approach resulted in more failures than the invalid parameter approach.

Starting in Linux v2.6.20, support was added to allow for fault-injection at instrumented points of key kernel subsystems [8]. The current set of support injection points target API level failures for memory (slab errors and page allocation errors) and disk IO errors.

FIG [4] is a fault-injection tool for introducing errors at the library level. They use the `LD_PRELOAD` mechanism associated with the loading of shared libraries to interpose on calls and inserting errors.

Note, while there has been a great deal published on fault injection, and the techniques found to be quite useful, many of the well cited works in the area are not freely available for download. At least two are available as part of commercial offerings, e.g., Xception, NFTAPE.

The area of dependability benchmarking leverages fault injection in order to assess the integrity of systems. Many of the tools mentioned here have been used in these studies [3, 11, 12].

7. CONCLUSION

We presented a general framework for fault injection in distributed systems. Because of its modularity, the framework enables experimentation with many different fault tolerance/system resilience mechanisms and policies. It also enables experimentation with various fault reporting/detection mechanisms. For instance, it is possible to test fault reports via system logs injecting faults directly into the system.

To illustrate the benefit of the proposed framework, we presented the implementation of a system for the injection of faults into memory, using two different fault injection techniques. One based on the `ptrace(2)` system call, and a second based on Linux (in-kernel) fault injection capabilities.

The current prototype is under active development. In the future, we plan to extend the capabilities of the prototype and increase our set of experiments. For example, use the Linux FI facility to test cases where the target (victim) is at the system level (kernel modules). Additionally, we plan to use the framework to investigate failure detection mechanisms and to evaluate tools for system resilience analysis.

8. REFERENCES

[1] Algirdas Avižienis, Jean-Claude Laprie, and Brian Randell. Fundamental concepts of computer system dependability. In *IARP/IEEE-RAS Workshop on Robot Dependability: Technological Challenge of Dependable Robots in Human Environments*, May 21-22, 2001.

[2] Francis M. David and Roy H. Campbell. Building a Self-Healing Operating System. In *Proceedings of the 3rd IEEE International Symposium on Dependable, Autonomic and Secure Computing*, pages 3–17, Columbia, MD, Sep 2007.

[3] Dbench: Dependability benchmark project. Available at: http://www.laas.fr/DBench/. (Last accessed March 2009).

[4] Fig: Library-level error injection for shared libraries in unix/linux. Available at: http://roc.cs.berkeley.edu/projects/fig/index.shtml (Last accessed: March 2009).

[5] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, Apr 1997.

[6] T. Jarboui, J. Arlat, Y. Crouzet, and K. Kanoun. Experimental Analysis of the Errors Induced into Linux by Three Fault Injection Techniques. *Proceedings of International Conference on Dependable Systems and Networks (DSN 2002)*, pages 331–336, 2002.

[7] João Carreira and Henrique Madeira and João Gabriel Silva. Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers. *IEEE Transactions on Software Engineering*, 24(2), February 1998.

[8] Linux fault injection capabilities infrastructure. Documentation available at: http://lxr.linux.no/linux/Documentation/fault-injection/.

[9] Adam Oliner and Jon Stearley. What supercomputers say: A study of five system logs. In *Proceedings of the 37th International Conference on Dependable Systems and Networks (DSN)*. IEEE Computer Society, June 25-28, 2007. Edinburgh International Conference Centre, Edinburgh, UK.

[10] S. Potyra, V. Sieh, and M. Dal Cin. Evaluating fault-tolerant system designs using FAUmachine. In *Proceedings of the 2007 Workshop on Engineering Fault Tolerant Systems (EFTS'07)*, page 9, New York, NY, USA, 2007. ACM.

[11] Roc: Recovery-oriented computing. Available at: http://roc.cs.berkeley.edu/ (Last accessed: March 2009).

[12] David T. Stott, Benjamin Floering, Daniel Burke, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. NFTAPE: A framework for assessing dependability in distributed systems with lightweight fault inectors. In *Proceedings of the 4th IEEE International Computer Performance and Dependability Symposium (IPDS)*, pages 91–100. IEEE, March 2000.