Fault Injection Framework for System Resilience Evaluation


Oak Ridge National Laboratory
Computer Science and Mathematics Division
System Research Team

*University of Tennessee
Electrical Engineering and Computer Science Department
Context

• Large Scale HPC Systems
  – Increased number of components
  – Increased complexity (hardware/software)

• HPC Applications
  – Challenged by scale
  – Challenged by failures
Motivation

- **Resilience**
  - Keep HPC applications running in spite of failures

- **Experimentation**
  - Investigate methods to support resilience research

- **Fault Injection**
  - Provides technique for resilience experimentation
  - Repeatable process to study failures
Terminology

- **Fault, Errors & Failures (Laprie Taxonomy, DSC’04)**
  - Fault – a defect in a service, may be “active” or “dormant”
  - Error – an “active fault” in a service
  - Failure – unsuppressed error, visible outside the service

- **Fault Injection**
  - Purposeful introduction of faults (errors) into target/victim
  - Hardware or Software
    - “SWIFI” – Software Implemented Fault Injection
Fault Injection / Testing

• First purpose: testing our research
  – Inject failure at different levels: system, OS, application
  – Framework for fault injection
    • Controller: Analyzer, Detector & Injector
    • Target system & user level targets
  – Testing of failure prediction/detection mechanisms

• Mimic behavior of other systems
  – “Replay” failures sequence on another system
  – Based on system logs, we can evaluate the impact of different policies
Fault Injection

• Example faults/errors
  – Bit-flips - CPU registers/memory
  – Memory errors - mem corruptions/leaks
  – Disk faults - read/write errors
  – Network faults - packet loss, etc.

• Important characteristics
  – Representative failures (fidelity)
  – Transparency and low overhead
  – Detection/Injection are linked

• Existing Work
  – Techniques: Hardware vs. Software
  – Software FI can leverage perf./debug hardware
  – Not many publicly available tools
Related Work

- **Xception** – leveraged hardware supported debug/perf monitoring capabilities
- **FAUmachine** – simulated faults in a user-space process (similar to UML)
- **FIG** – introduce errors at library level by interposing on calls to shared library (use LD_PRELOAD)
- **NFTAPE** – component-based fault injection system for distributed environments
- **Linux-FI** – in kernel fault injector with current support for areas of the memory and IO subsystems
Existing System Level Fault Injection

• “Existing” source that is free & publicly available

• Virtual Machines
  – FAUmachine
    • Pro: focused on FI & experiments, code available
    • Con: older project, lots of dependencies, slow
  – FI-QEMU (patch)
    • Pro: works with ‘qemu’ emulator, code available
    • Con: patch for ARM arch, limited capabilities

• Operating System
  – Linux (>= 2.6.20)
    • Pro: extensible, kernel & user level targets, maintained by Linux community
    • Con: immature, focused on testing Linux
Linux Fault Injection (Linux-FI)

- Kernel supported fault injection
  - Linux >= 2.6.20
  - Send faults to user-space (PID) and system-level (module/addr)
  - Supports faults in several key kernel subsystems

- Supports injecting \( \text{(as of v2.6.25.7)} \)
  - Slab errors
  - Page allocation errors
  - Disk IO errors

- Interface via \texttt{debugfs}
  - Enable Linux FI via entries in \texttt{/debug} file-system
  - Set probability for given fault
    - Example: 0 (never) ... to ... 100 (always)
Basic Criteria for FI Framework

- **Simplicity**
  - Easy to setup, define and perform FI experiments

- **Versatility**
  - Support experiments at different levels of software stack
    - User and Kernel level

- **Reproducibility**
  - Framework should allow for reproducible experiments

- **Distributed environments**
  - Experiments on local & remote nodes; physical & virtual machines
Fault Injection Architecture

- **Driver**
  - Interface between user and framework

- **Controller**
  - Manages life-cycle of components (create, run, terminate)

- **Analyzer**
  - Responsible for collating/processing experiment info
  - Interprets events for a given detector/injector configuration

- **Injector**
  - Generates a fault (error) in a given victim/target

- **Detector**
  - Detects a failure in a given victim/target
Fault Injection Architecture

- Driver
- Controller
- Analyzer
- Injector
- Detector
- Victim/Target
Evaluation

- Initial framework implementation
  - Prototype called finject

- Preliminary evaluation
  - Memory/register based fault injection

- Two experiments
  - Experiment I: ptrace based injector
  - Experiment II: Linux-FI based injector
FInject Input File

- **Experiment file: “experiment.txt”**
  - Used to express type of failure & experiment parameters
  - One experiment per line

```
# finject experiments
# Format:
# fault_type : fault_mode : fault_args : victim_host : flags
memory : intermittent : app='/tmp/fileptr' : ubuntu-vm : finject='kern-memory',dargs='50'
register : permanent : app='/tmp/loopnest-forever' : localhost : finject='user-memory'
```

*Note, currently only minimal subset of input fields are supported*

- **Usage**

  ```
  ./finject --file experiment.txt
  ```
FInject Config File

- Framework config file: “finject.conf”
  - Used to group compatible injectors-detectors-analyzers
  - Determines backend modules used by framework for experiments

```
# finject experiment settings
[user-memory]
injection=injectors/frob-reg-injector
detector=detectors/child-watcher
analyzer=analyzers/basic-counter
```
Experiment I: Ptrace based injector

- Injects CPU register errors (bit-flips) via `ptrace()`

- Finject Components
  - **Target:** “loopnest-forever”
    - App that runs infinite loop printing PID & counter
  - **Analyzer:** “basic-counter”
    - Counts labeled events from Detector & Injector
  - **Detector:** “child-watcher”
    - Starts app & watches/reports child exit status to Analyzer
  - **Injector:** “frob-reg-injector”
    - Injects bit-flip in register value for an app (PID) & notifies Analyzer
Experiment I (cont.)

- On average the dummy application failed after sending approximately 22 faults (register bit-flips)
- As expected the application spent almost all time in a library write routine printing the output, which wasn’t esp. sensitive to the register based errors

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

Table 1: Statistics associated with Experiment-I (register bit-flip)
Experiment II: Background on Memory

• Linux memory allocation
  – Generic pages
  – Object cache (SLAB)

• SLAB
  – Cache of typed memory objects
  – Reuse freed memory objects (performance)
  – Listing of object types & statistics via /proc/slabinfo

• Example
  – Maintain cache of file pointer ("filp") objects

```c
#include <stdio.h>
FILE *tmpfile(void);
```
Experiment II: Linux-FI based injector

- Injects memory allocation errors for ‘filp’ SLAB objects via Linux-FI

- Finject Components
  - Target: “fileptr”
    - Creates temporary file(s) & handle via `tmpfile()`
  - Analyzer: “basic-counter”
    - Counts labeled events from Detector & Injector
  - Detector: “fileptr-watcher”
    - Starts app & watches child STDOUT and exit status, notifies Analyzer
  - Injector: “linux-fi-injector”
    - Just report kernel generated faults to Analyzer
    - Actual injector is the Linux-FI subsystem
Future Work

- **Finalize initial finject prototype**
  - Framework itself
  - Injector/Detectors: Linux-FI (SLAB) & ptrace

- **Fault types/methods**
  - Identify representative failures for HPC systems
  - Determine how best to perform injection/detection

- **Anomaly analysis**
  - Combine tool with current anomaly analysis prototype
  - Investigate anomaly/failure correlation
Conclusion

• Resilience research needs platforms/tools for repeatable experimentation

• Fault injection provides a useful mechanism to perform repeatable testing and development

• Proposed fault injection framework provides basis for building resilience testbeds/environments

• Prototype leveraged ptrace(2) and Linux-FI
  – CPU register bit flips
  – Linux SLAB allocation errors (type ‘filp’)
Resources

http://www.csm.ornl.gov/srt
Flow of FInject Experiment

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