A Software-Implemented Fault Injection Methodology for Design and Validation of System Fault Tolerance

Raphael R. Some*, Won S. Kim, Garen Khanoyan, Leslie Callum, Anil Agrawal, and John J. Beahan
Jet Propulsion Laboratory, California Institute of Technology, 4800 Oak Grove Drive, Pasadena, CA 91109
Raphael.R.Some@jpl.nasa.gov

Abstract

In this paper, we present our experience in developing a methodology and tool at the Jet Propulsion Laboratory (JPL) for Software-Implemented Fault Injection (SWIFI) into a parallel processing supercomputer, which is being designed for use in next generation space exploration missions. The fault injector uses software-based strategies to emulate the effects of radiation-induced transients occurring in the system hardware components. The JPL's SWIFI tool set called JIFI (JPL's Implementation of a Fault Injector) is being used, in conjunction with an appropriate system fault model, to evaluate candidate hardware and software fault tolerance architectures, determine the sensitivity of applications to faults and measure the effectiveness of fault detection, isolation, and recovery strategies. JIFI has been validated to inject faults into user-specified CPU registers and memory regions with a uniform random distribution in location and time. Together with verifiers, classifiers, and run scripts, JIFI enables massive fault injection campaigns and statistical data analysis.

1. Introduction

The objective of the Remote Exploration and Experimentation (REE) Project is to bring supercomputing technology into space [1], [2]. It has twin goals of 1) demonstrating a process for rapidly transferring Commercial-Off-The-Shelf (COTS) high-performance computing technology into ultra-low power, fault tolerant architectures for space and 2) demonstrating that high-performance onboard processing capability enables a new class of science investigation and highly autonomous remote operation. It is important to note that REE's use will be as a scientific data processor, not a spacecraft control computer. This is critical in assessing the suitability of the REE system to the application and to evaluation of its performance. It is also key to defining a development and testing approach including the appropriate use of fault injection based experimentation.

The REE project is employing mainly COTS hardware and software components due to the wide gap (10-100x) in performance/power between commercial processors and those available in a radiation hardened version as well as in cost (projected to be 100x or greater). The REE architecture [1] is a parallel processing "cluster computer" consisting of a pool of interconnected stand-alone computers working together as a single integrated computing resource. The MPI (Message Passing Interface) standard is used for parallel program coordination and communication.

Because REE hardware is not radiation hardened, Galactic Cosmic Ray (GCR) and Solar Proton induced transients will cause Single Event Effects (SEE's), i.e., spurious bit flips, signal line noise and clock pulses, in the system hardware components, thus affecting system reliability and availability. Expected fault types and rates are being determined by a separate modeling effort in which the results of ground-based radiation testing are extrapolated to a range of mission environments. The fault model thus generated provides the expected fault conditions for the mission environments of interest.

A suite of fault injection tools used in conjunction with the fault model [2] is needed to analyze and evaluate fault behavior of the REE system. Before a fault-tolerant system is deployed, it must be tested and validated. How reliable is it in detecting and recovering from faults? Are there critical hardware/software components or techniques in fault-tolerant system design? What are the sensitivities of different application software modules and the system software to various types of faults? What are the fault detection and mitigation coverage? These questions become even more relevant in the REE system , where COTS components instead of radiation-hardened chips are used.

There are basically four approaches to implementing fault injection: 1) hardware-implemented fault injection, 2) fault injection by simulation, 3) software-implemented fault injection (SWIFI), and 4) hybrid fault injection which is a combination of the above three techniques. Hardware-implemented fault injection techniques need special hardware, and simulation techniques need a good simulation model of the target system. SWIFI techniques alter the hardware/software state of the system using special software in order to cause the system to behave as if a hardware fault had occurred. SWIFI techniques have several potential advantages compared to the other

0-7695-1101-5/01 $10.00 © 2001 IEEE

techniques: 1) lower complexity detailed models, 2) reduced development effort, 3) lower cost, and 4) increased portability. There are quite a few SWIFI based fault injectors such as FIAT, FERRARI, NFTAPE, DOCTOR, ORCHESTRA, SPI, LOKI, and Xception. Most of these have been well documented elsewhere. Two which are of interest to the REE project, and which we are investigating for future use to augment JIFI, are Xception and NFTAPE.

Xception [3] is a commercially available fault injector that operates at the exception handler level, independent of the application code, to minimize intrusion into the application. NFTAPE [4] permits the injection of CPU, memory, I/O and communication related faults in a networked system. In NFTAPE, the fault injection component is implemented in a Lightweight Fault Injector (LWFI). This fault injector uses a component-based architecture that allows porting of NFTAPE to a variety of platforms.

JIFI has been developed to support the REE Project. This in-house development effort has provided JPL with experience and an understanding of fault injection. This paper describes the JIFI fault injector and recent enhancements to achieve uniform fault distribution over the user-specified fault location range and time interval. The original version was implemented by John Beahan [5] on PowerPC boards under the Linux operating system and was later ported to the real-time Lynx operating system. In this paper, we present an overview of JIFI including the theory of operation, uniform fault distribution, statistical error test, and experimental methodology.

2. Theory of Operation

JIFI is an application-level software-implemented fault injector, providing an easy-to-use environment for fault injection experiments including massive fault injection campaigns to get statistical data on fault performance. The user needs to add a few JIFI function calls inside the application source codes to allow fault injections. When used with a configuration file and execution scripts, a campaign of a thousand fault injection experiments may be automatically run, results collected and evaluated and statistical results obtained in a single day.

The basic approach is to have a JIFI fault injector process forked from the application program for each node in such a way as to be transparent to MPI and other facilities. By utilizing the system ptrace() routines, JIFI can inject bit-flip faults from outside of the application into the registers and memory of an executing process. It has facilities for both 1) time-triggered random faults and 2) location-triggered targeted faults. The time-triggered faults are injected into the application at the specified rates, but are not repeatable due to randomized trigger. The location-triggered targeted faults are injected into variables, arrays, data structures, or subroutines specified by the user in the application, and the application can control the injection process in a repeatable manner.

Control of fault rates, single/multiple-fault models, multiple-bit-flip faults, watchdog timeouts, modeled memory size, logging to files, verbosity of output, and injection control is done via setting parameters in a centrally-located set of hierarchical configuration files. The configuration file can be easily customized for testing single or multiple MPI tasks across multiple machines, with settings customizable on a per-task basis.

Two kinds of fault injection locations are supported by JIFI: CPU registers and memory. The Power PC 604 or 750 CPU contains more than 120 registers. The current JIFI tool identifies 75 of the registers by the number 0 to 74, including 32 general purpose registers and 32 floating-point registers. Among the 75 registers, ptrace() does not support alteration of the content of register 32, 33, 34, and 39, and thus faults cannot be injected into these four registers.

Memory fault locations can be grouped into user area and Operating System (OS) kernel area. At present, only fault injections into user areas are available. Fault injections into the following four user memory areas are supported: user text, user data, user heap, and user stack. The text area contains machine codes and read-only constants. The data region contains global variables. The heap region contains dynamically allocated memories which are created/allocated by calling malloc() or calloc(). The stack region contains local variables. Both heap and stack memory sizes vary dynamically.

JIFI tools support uniform fault distribution over time for time-triggered random faults at specified rates. The fault rate for each fault injection location group is specified by its inverse parameter, average fault injection interval or mean time between faults (MTBF). Each injection region can be assigned different MTBF's. JIFI tools also support multiple-bit-flip fault injections. When the user sets the JIFI parameter multiBitMem to 1, each memory fault injection causes a single bit-flip.

A functional block diagram of JIFI is shown in Figure 1. When swifiInitMPI() is called at the beginning of the application main(), it forks JIFI as a parent process and the application as a child process for each node. The JIFI parent process attaches ptrace to the application child process, so that the application process responds to JIFI ptrace calls, while executing the application program. The JIFI process computes the time to the next fault injection based on uniform random distribution over time using fault rates (specified by MTBF's) of all activated fault injection regions, and sets the timer to the computed time to invoke the SIGALARM signal upon timer expiration. Until SIGALARM is detected, the JIFI process continues baby-sitting by tending all the signals received. Since all

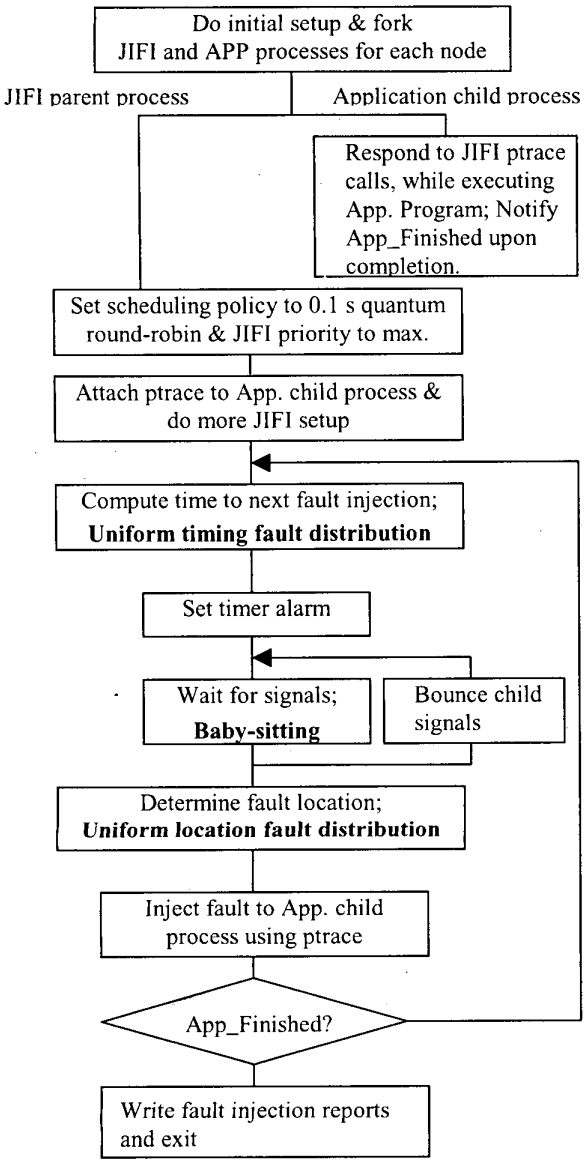

Figure 1. JIFI functional block diagram illustrating how faults are injected into the application program with uniform distribution in time and space. Since all signals bound for the ptraced child process are intercepted, the baby-sitting loop handles these signals while waiting for the JIFI timer alarm signal.

the signals bound for the ptraced child process are intercepted, the baby-sitting loop bounces these signals to the child process properly. Uniform timing fault distribution is maintained independently for each of the different fault-location types of register, code, data, stack, heap, and user-defined regions.

When the baby-sitting loop receives the SIGALARM signal indicating timer expiration, the JIFI process stops the application process immediately. It then determines the fault location based on a uniform random distribution over the specified fault region, and injects the fault into the application process using ptrace(). Upon completion of the fault injection, the JIFI process saves the fault injection data and goes back to the beginning of the fault injection loop to prepare for the next fault injection. If the total number of faults has reached the user specified number, no more faults are injected. When the application finishes the program execution, it notifies JIFI of its completion via the App_Finished flag and swifiEnd() function call. JIFI then writes fault injection reports for each node of the multi-processor system and exits.

3. Uniform Fault Distribution

It is also desirable to be able to perform a wide range of both targeted and random fault injection experiments. For example, instead of injecting faults randomly over the entire range of the memory locations and over the entire duration of the program's execution, the fault injector should be able to inject faults with a uniform random distribution over a specified location range and execution time window. Thus, injection is targeted towards those areas and times representing the resources used by a particular routine or representing a specific program execution point or a mission event. This controlled fault injection enables us to investigate fault tolerance behavior (such as sensitivities, error latency, propagation, detection, and recovery) for each software segment.

3.1 Uniform fault distribution over time

An initial version of JIFI failed to yield uniform random fault distribution over time (Figure 2). A simple "iterative floating-point addition" program was run 200 times with JIFI configuration parameters of: initDelay = 1 s (no injection for the initial 1 s), regMTBF = 1 s (register mean time between fault injections is 1 s), regInjActive = 1 (register fault injection ON), and multiFault = 1 (1 fault per run). The fault injection time distribution from the 200 runs clearly shows that fault injections occur discretely at 0.25 s intervals (Figure 2). The ideal result would have been a step pulse of a 1 s interval from 1 to 2 s. This problem occurred because the JIFI and application programs were running at the same priority level and the default LynxOS round-robin scheduling policy had the quantum size of 0.25 s. Figure 2 also shows that the fault injection started after 2 s, much later than the specified starting time of 1s.

This quantum/priority problem has been fixed in the recent enhanced version of JIFI. As described earlier in

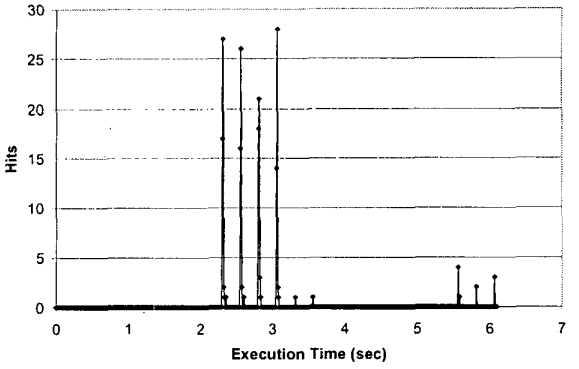

Figure 2. Time distribution for 200 runs of a register fault injection per run with initDelay = 1 s and regMTBF = 1 s, using an earlier version of JIFI before enhancement. Fault injections occur at 0.25 s discrete intervals due to the 0.25 s quantum of the default round-robin scheduling policy.

Figure 1, the JIFI parent process now changes the scheduling policy from the default LynxOS round-robin with a 0.25 s quantum to the default POSIX round-robin with a 0.1 s quantum. Further, the JIFI process is now set to the maximum priority so that when the SIGALARM is up at the calculated fault injection time, the JIFI process can pre-empt the application process immediately to inject the fault without waiting for the application to run for the entire quantum. One thousand runs of the enhanced JIFI version using the same "iterative floating-point addition" program with the same JIFI configuration parameters for register fault injections show a fairly uniform distribution over the 1 s interval from 1 to 2 s (Figure 3). When JIFI was not able to inject a fault in the first try, JIFI tried in the next 1 s interval.

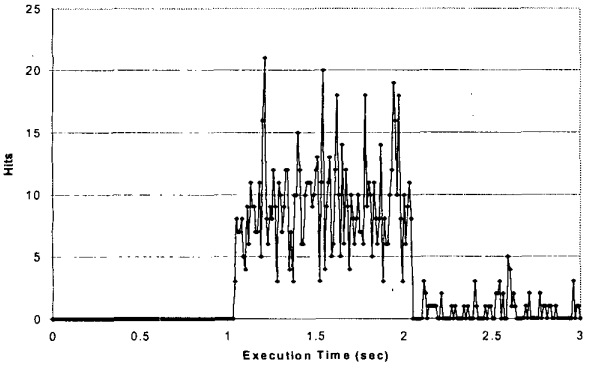

Figure 3. Fault time distribution for 1000 runs of register fault injections (1 fault per run), showing a fairly uniform step pulse distribution from 1 to 2 s with initDelay = 1 s and regMTBF = 1 s. Some register fault injections failed in the first attempt, and new attempts were made in the next 2 to 3 s interval.

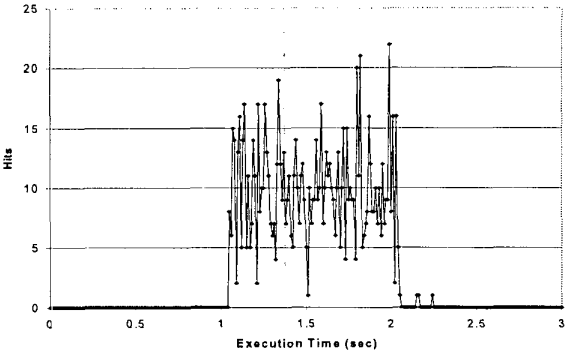

Figure 4. Fault time distribution of 1000 runs of memory fault injections (1 fault per run), showing a fairly uniform step pulse distribution from 1 to 2 s with initDelay = 1 s and memMTBF = 1 s. Very few memory fault injections failed in the first attempt, compared to the above register fault injections.

The test result of the enhanced JIFI version from the same 1000 runs for memory fault injections also demonstrates a fairly uniform distribution (Figure 4). As can be seen in the figure, JIFI was able to inject almost all memory faults on the first attempt.

3.2 Uniform fault distribution over location

As described earlier, JIFI is designed to yield uniform random fault distributions over both time and location. The register fault distribution for the same 1000 register fault injection runs corresponding to Figure 3 is shown in Figure 5, demonstrating a fairly uniform register location distribution. Registers 32, 33, 34, and 39 are not injectable. The memory fault distribution for the same 1000 memory fault injection runs corresponding to Figure 4 is shown in Figure 6, again demonstrating a fairly uniform memory location distribution. For memory fault injections, faults are injected uniformly over all four regions of text, data, heap, and stack. In this specific example of the "iterative floating-point add" MPI application program, the sizes of text, data, heap, and stack regions were 296 KB, 659 KB, 185 KB, and 5 KB, respectively. Although the heap and stack sizes are generally changing dynamically, they are basically constant within the floating-point add loop. Since the stack size is small in this example, only one stack hit occurred out of 1000 runs. The gdb debugger symbol table output indicated that the starting addresses of the text, data, and stack regions are fixed at 0x10001000, 0x20000000, and 0x80000000, respectively, while the heap region starts at 0x200A0024 right after the end of the data region. These address values agree with the virtual memory map described in LynxOS manuals.

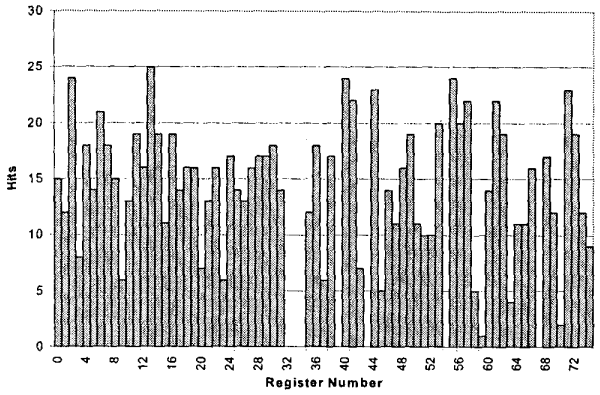

Figure 5. Register fault distribution for 1000 runs of register fault injections (1 fault per run), showing a fairly uniform distribution across the entire range of all 75 registers defined by JIFI. Register 32, 33, 34, and 39 are not injectable.

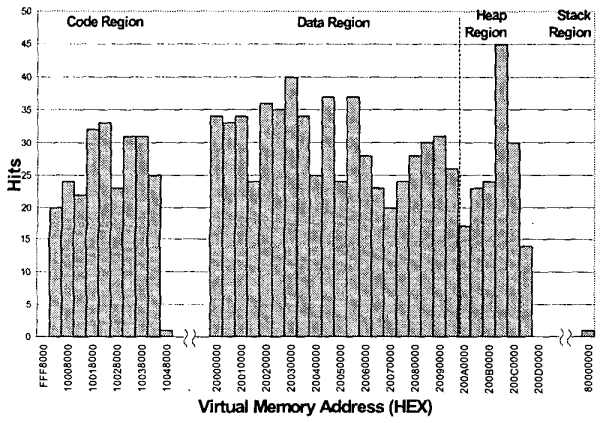

Figure 6. Memory fault distribution for 1000 runs of memory fault injections (1 fault per run), showing a fairly uniform distribution over all four regions of user code (296 KB), data (659 KB), heap (185 KB), and stack (5 KB). Each bin is 32 KB except for the end bin of each region. Only one stack hit occurred in this example due to a small stack size.

4. Fault Injection Data Analysis Tools

Fault injection experiments require data logging and analysis tools to analyze fault behavior and evaluate the system. Data logging and analysis tools include 1) JIFI output, 2) debugger, 3) verifier, 4) classifier, and 5) data plot tool.

JIFI generates the JIFI output file for each run. It prints out the application end status, total number of faults, run time, as well as detailed data for each fault such as fault injection time, fault location, values before and after injection, program counter at the time of injection, and others. It generates a comma-delimited format so that relevant data can be easily extracted.

A debugger such as gdb can be used to get the symbol table or linker map, which provides the address information of all functions and global data in the application code and data regions. It enables the user to relate each fault location to the software component such as various application modules, MPI, C libraries, or JIFI.

A verifier compares the current output file with the gold-run file and categorizes the outcome into three groups: 1) CorrectOutputFile, 2) IncorrectOutputFile, and 3) NoOutputFile. The correct output file produced with no faults injected is called the gold-run file. In the scientific parallel programs which are REE's applications, two issues arise with the typical "gold run" approach: 1) slightly different execution environments such as the number of nodes performing the application, may result in correspondingly different output files, both of which may be "correct" and 2) due to the noise environment of the application such as may occur in the instrument or simply noise in the measured variable, there is an acceptable range of "correct" output values, i.e., there is not a single, unique "correct" answer. It was therefore necessary to work with application developers (research scientists) to devise "verifiers" which grade the application output and provide indication of acceptable variance from the "gold" output. Each application, and beyond this, potentially each phase of each mission where the application may be used, requires a unique verifier to ascertain the sensitivities and reliability of the application under simulated fault conditions.

A classifier uses the information obtained from the above tools to classify the outcome of each fault injection run into different groups. In the application-run level, the outcome can be categorized into five groups: 1) Correct --- application output file was created and correct, 2) Incorrect --- application output file was created but incorrect, 3) Crash --- application output file was not created and JIFI output reported application crash status, 4) Hang --- the application was timed out by JIFI after the maximum time limit, and 5) Invalid --- no faults were injected.

5. Statistical Error Tests

A texture segmentation program for Mars rover autonomy applications was initially used for statistical error tests. For each run, a fault was injected into the entire code region (application code, MPI code, and C libraries code) during the FFT module execution. Each run was classified as CORRECT, INCORRECT, CRASH,

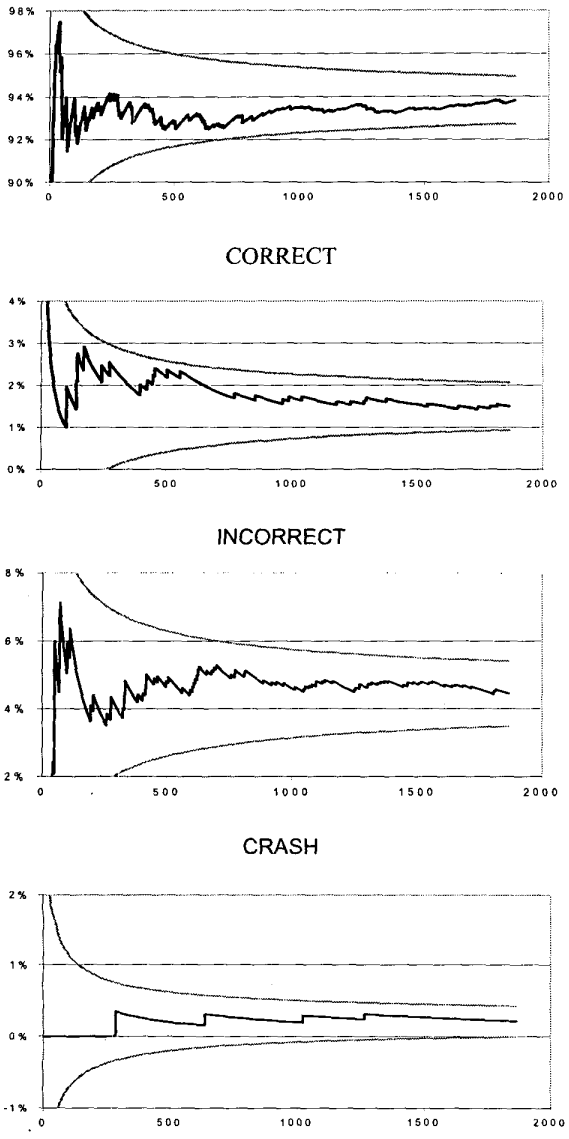

Figure 7. Statistical error tests with $\pm 2\sigma$ curves.

HANG, or INVALID based on JIFI and verifier outputs as described above. Since the outcome yields either TRUE or FALSE for each class, each class outcome is a binomial distribution with

$$\mu_i = c_i / N, \quad (1)$$

$$\sigma_i = \sqrt{\mu_i * (1-\mu_i) / N}, \quad (2)$$

where μ_i is the mean, σ_i is the standard deviation, c_i is the number of runs that belong to the class *i*, and N is the total number of runs. Figure 7 shows the "moving" average vs. the number of runs for each class, obtained from 2 sets of 1000 fault injection runs. The $\mu_i \pm 2\sigma_i$ curves are overlaid to show the 95% confidence level. If the data were far off from between these curves, the system would have been time-varying and the time-varying factors would have to be isolated as controlled variables. A different input image, for example, could produce a somewhat different outcome. The plots and equation (2) indicate that 500 to 1000 runs are sufficient to achieve acceptable first-order statistically meaningful results for controlled experiments. In the case of 500 runs, 2σ=0.9% for μ=1% or 99%, and 2σ=1.9% for μ=5% or 95%. However, if one wants to achieve 2σ=0.1% for μ=1%, 40,000 runs are required!

6. Conclusion

A fault injection tool set for use in experimentation on a fault tolerant parallel processing cluster has been developed. Initial experience indicates that the tool will be useful in both developing and validating the computer design and its fault coverage as well as characterizing system behavior under fault conditions. In the course of the development, several problems including achieving uniform random distribution of fault injections were encountered and their solutions were described.

7. References

[1] R. R. Some and D. C. Ngo, "REE: A COTS-Based Fault Tolerant Parallel Processing Supercomputer for Spacecraft Onboard Scientific Data Analysis," Proc. of the Digital Avionics System Conference, vol. 2, pp. B3-1-7 - B3-1-12, 1999.

[2] J. J. Beahan, L. Edmonds, R. D. Ferraro, A. Johnston, D. Katz, R. R. Some, "Detailed Radiation Fault Modeling of the Remote Exploration and Experimentation (REE) First Generation testbed Architecture," Aerospace Conf. Proc., vol. 5, pp. 279-291, 2000.

[3] D. T. Stott, B. Floering, D. Burke, Z. Kalbarcyzk and R. K. Iyer, "NFTAPE: a Framework for Assessing Dependability in Distributed Systems with lightweight fault injectors" , Proceeding Computer Performance and Dependability Symposium, pp. 91-100, 2000.

[4] J. Carreira, H. Madeira and J. G. Silva, "Xception: A technique for the Experimental Evaluation of Dependability in Modern Computers," IEEE Transactions On Software Engineering, Vol 24, pp. 125-135, Feb. 1998.

[5] J. J. Beahan, "SWIFI: A Software-Implemented Fault Injection Tool," JPL Internal Document, June 2000.

Acknowledgment

This work was performed at the Jet Propulsion Laboratory, California Institute of Technology under a contract with the National Aeronautics and Space Administration. This project is part of NASA's High Performance Computing and Communications Program, and is funded through the NASA Office of Space Sciences.