

Assessing Multi-Version Systems Through Fault Injection

Paul Townend and Jie Xu

Dept. of Computer Science,

University of Durham, DH1 3LE, England

E-mails: p.m.townend@dur.ac.uk, jie.xu@durham.ac.uk

1 Introduction

Over the last few decades, a wide-range of industries have developed a growing dependence on software-based systems. Many of these systems are critical, real-time systems developed for safety-critical, business-critical or mission-critical applications, and it can be seen that the failure within such systems has the potential to be devastating. Given the need for dependability, many software systems still have an unacceptably high level of faults. *Multi-version design* (MVD) has been proposed as a method for increasing the overall dependability of software systems above that of those developed using traditional approaches. MVD works by implementing and executing several functionally equivalent systems, and comparing their outputs, with the consensus output being forwarded as the final system result, thus by-passing faults in individual systems; should no consensus be reached, human operators can be alerted to the situation [AVI77].

The multi-version approach has gained attention as a number of researchers have documented significantly increased levels of dependability within software developed using this methodology; for example, Hatton's 1997 analysis [HAT97], based on the Knight and Leveson experiment [KNI86] concludes that a three-channel version of the system, governed by majority polling would have a dependability improvement ratio of 45:1 over a single variant of the system. This is not a new finding; earlier papers, such as [AVI84] have also argued that the approach produces highly dependable software.

Such massive increases in dependability have, however, been drawn into question, and much debate has ensued; Leveson and Knight [KNI90] argue that these gains in dependability are under the assumption that there are no *correlated (common-mode) failures* within two or more channels of the system – in other words, no faults will occur in the same place and produce the same results. Numerous studies, beginning with [SCO84] have shown that this is simply not the case. Eckhardt and Lee’s study [ECK85] has shown that even small probabilities of correlated faults can reduce the overall dependability of an N-version system dramatically, and Leveson [LEV95] further argues that every experiment with the approach of using separate teams to write versions of the software has found that independently written software routines do not fail in a statistically independent way. Examples of this can also be found in [ECK91, KEL88]. The voting software used in multi-version design must also be developed correctly and free of fault, otherwise the entire system can become unstable. An example of this is the NASA study of an experimental aircraft, which found that all of the software problems that occurred during flight testing were the result of faults found in the redundancy management system, and not the control software itself [MAC88].

Therefore, it appears to be the case that such massive dependability gains can only be assumed on a theoretical level. In real-world applications, the overall cost/dependability ratio is likely to be much lower for a multi-version system than the theoretical model may suggest, although MVD systems are still potentially more dependable than traditional simplex systems (for example, previous research by the authors [TOW01a, TOW01b] indicates that even when a MVD system has poor quality channels, it may still be seen as “safer” than a simplex system as most errors will be detected by the voter).

This uncertainty toward MVD systems is accentuated by the lack of dependability metrics available in regard to such systems. Indeed, a major obstacle to the large-scale commercial rollout of MVD systems is the lack of quantitative characterizations of the approach. This is important, as in most cases resource allocation cannot be done arbitrarily or carelessly [KIM00], and without relevant metrics, sensible resource allocations cannot be achieved.

2 Fault Injection

It can therefore be seen that a concerted effort needs to be made to improve the level of empirical knowledge in regard to multi-version systems. This has been done to limited effect with traditional testing methods, but the area of fault-injection has been especially neglected.

Fault injection is a phrase covering a variety of testing techniques that can be applied to both hardware and software, all of which involve the *deliberate insertion of faults into an operational system to determine its response* [CLA95]. Once this has been performed, an examination of the system for resulting errors and failures occurs, such as analysis of interactions between system components and of the resilience of the system against known faults. Fault injection is a “late life-cycle” software analysis [VOA98a] that can simulate human operator errors and observe their impact on the software as well as the *total* system. It is a technique that *complements*, but is not a substitute for, other verification and validation procedures.

Faults can be introduced in one of two ways - either through direct alteration of code, or by the perturbation of data flows or control flows to achieve the effects of faults indirectly - and can be categorized based on when the faults are injected: either during compile-time or run-time [HSU97].

When altering program code, faults are typically created by either adding extra instructions to the code under analysis, changing the code, or deleting code. Code that is added to a program for the purpose of either simulating errors or detecting the effects of those errors is called *instrumentation code*. To perform fault injection, some instrumentation is always necessary. Typical injected faults include mis-timings, delays, missing messages, corrupted memory, faulty disk reads, logical errors, syntax errors and perturbation of variables. Faults can be injected in many ways and can address program state as well as communication and interactions. Input data can also be targeted; *faulty input data* to be passed into a system at run-time – either by the mutation of ‘real’ data or a false set of data. [VOA98a] suggests that faulty input data is the easiest form of fault to simulate correctly (i.e. in a way that reflects real errors that could occur naturally).

There are two key approaches for instrumentation – code mutation and state perturbation. *Code mutation* [DEM78] occurs at compile-time and involves direct alteration of program code, attempting to reproduce potential human errors within code; this typically involves changing the syntax of existing code statements or modifying their logic in some way. *State perturbation* [VOA97] has the intention of forcefully modifying program states created by the original code, without mutating existing code statements. This is often achieved through the use of *code insertion* whereby instrumentation code (termed *perturbation functions*) is added to a system in the form of function calls that modify internal program values, but it can also be implemented by modifying input data or by the fault injector trapping exceptions generated by the system through the use of interrupts. Perturbation functions are code instrumentation, and are typically applied to programmer-defined variables. They can either change the value of a variable to a value based

upon the current value, or can change the variable to a value picked at random, independent of the original value. They may also return a constant replacement value, if it is suspected that any fault placed at that point in the code will result in one particular value regardless of what the current value is. When non-constant replacement values are used, the perturbation functions produce random values based upon the current value and a *perturbing distribution*, with non-constant perturbing distributions including all of the continuous and discrete random distributions.

Fault injection is intended to yield three results: an understanding of the effects of real faults, feedback for system correction or enhancement, and a forecast of expected system behaviours [CAR99]. One of the major benefits of fault injection is its ability to test rare events and conditions, which have been shown to be the cause of the majority of failures within critical systems. [HEC96] states that “*The basic premise of the rare events approach is that well-tested software does not fail under routine input conditions, which means that failures must be triggered by unusual input data or computer states*”. Such unusual input data and computer states can easily be achieved with fault injection, and systems can be stress tested with large amounts of unusual conditions to garner their response. In this way, fault injection also helps to test the exception handling and redundancy management capabilities of a system, which are often overlooked by traditional testing. Fault injection is also used to measure software *sensitivity*, or tolerance. Sensitivity is measured based upon a system’s reaction to injections; high sensitivity means that injections frequently cause the system to produce undesirable outputs (“undesirable” is defined in either the system specification, requirements or defined software hazards [VOA97]). High sensitivity implies a lower tolerance for failure, and thus shows a system to have a greater risk of failure than a low sensitivity system.

One of the main arguments for the use of fault injection in MVD systems is that the very low failure rate required for many such systems makes traditional testing techniques and reliability growth models unsuitable [BUT93]. A good example of this is that it would take 10^8 to 10^{10} hours (thousands of years) of testing to demonstrate a failure rate of 10^{-7} to 10^{-9} per hour, assuming one copy of software is tested and one failure is observed. Also, most multi-version systems are highly complex, and it is often infeasible to perform the enormous amount of test cases required to test every possible input and system state; according to [VOA95], “*the number of tests required for establishing high reliability are impractical if not impossible for software of even modest complexity*”. Another weakness of traditional testing is that it often fails to exercise a systems response to rare (i.e. unlikely) events. A number of studies, such as [HEC93] and [HEC94] have shown that many failures in well-tested systems are caused by such events. The same data from these studies also shows that *multiple rare events* are almost the exclusive cause

of the most critical failures in these systems. Fault injection also has the benefit of yielding large amounts of quantitative data that can be analysed in numerous different ways to yield large numbers of metrics about a system. This therefore makes the approach highly promising when seeking to derive an increasing number of metrics for MVD systems. Currently however, most fault injection systems within the software engineering field have concentrated on the assessment of single version software; as far as fault injection for diversity evaluation is concerned, the lessons from the literature are limited and of a general nature only [CHE99].

3 FITMVS

This research is centred around the initial results of an automated fault-injection system for the analysis of multi-version systems, developed at the University of Durham in order to provide a method for easily extracting useful metrics from such systems, as well as facilitating the testing process for MVD systems by identifying areas of code with a high sensitivity to errors and common-mode failure. This tool is called Fault Injection Tool for Multi-Version Systems (**FITMVS**).

FITMVS performs *state perturbation*, whereby code modifying a particular variable's value is added to an existing system's code. The basic operation of FITMVS is to parse the code of each channel within a multi-version system, and then systematically inject faults into each scope within a source file, compile and execute the code, test the system against a user-created set of tests, log the results, revert the code back to its original state and inject a fault into the next scope within the source file. This is continued until the last scope within the source file has had an injection made to it. At the conclusion of running FITMVS, a multi-version system will therefore have had at least one injection made into each scope within its code, and been tested for each of these injections. This is explained in more detail in figure 1 below.

Messages, such as test information and the results of a test are passed between FITMVS and the MVD system using a shared memory mechanism. Because of this, it is necessary to modify the code of the target system or voter so that messages can be sent, received and acted upon by the target system. However, as most of the memory mechanisms are included in a standard header file, the effort to modify the target system should not be problematic; also, the footprint of the mechanism should have an extremely minimal effect on the systems behaviour (although it would be naïve to necessarily assume it will have no effect whatsoever).

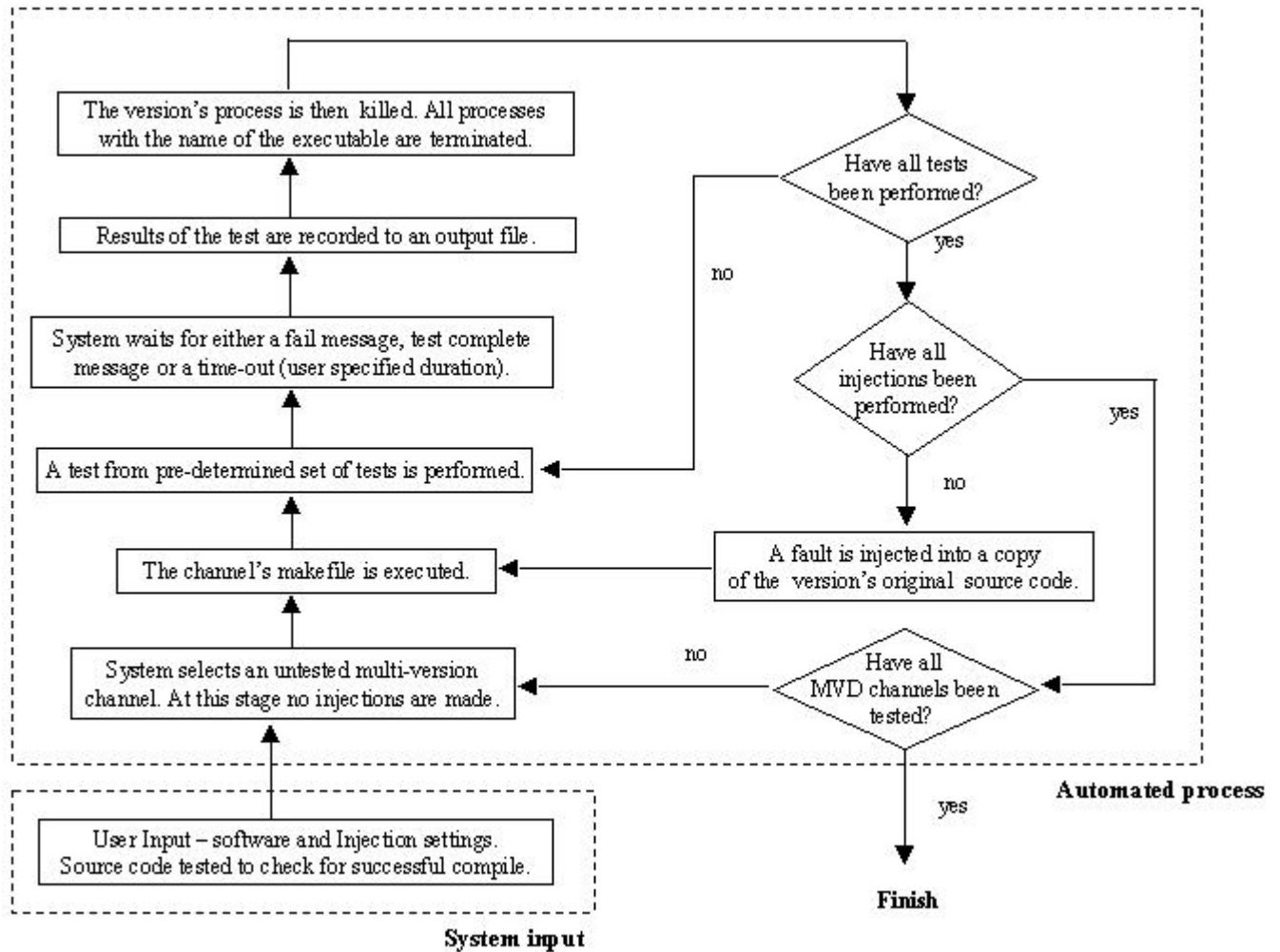


Figure 1 – The basic operation of FITMVS

When the automated testing process is in operation, FITMVS will wait for a test result message for a user specified period of time; should this time elapse, the test is deemed to have “timed out”.

By only performing data value perturbation, FITMVS neatly avoids the *equivalent mutant* problem. This occurs when an injection (in the form of code mutation) is made that does not affect the output of the code in any way (i.e. has no semantic impact on the code base) and is hence meaningless. Instead, all injections made by FITMVS will alter system state in some way – whether trivial or otherwise.

The injection settings of FITMVS can be configured quite substantially by the user. Obviously, injecting at least one fault in every scope in a source file will not scale to applications of even moderate size; therefore, the user

can specify the number of lines in size a scope must be before an injection is made. The user can also specify the number of injections that are performed per scope – the default being 1. This means that should the user specify more than 1 injection per scope, FITMVS will inject a fault, re-compile the code, test and log the result, revert the code back, and then re-inject a different fault into the same scope the user specified number of times; it is important to note that FITMVS never allows more than 1 injected fault to be present in code at any one time. The user may also specify the perturbation distribution of an injection. This refers to the maximum positive and negative value that a variable will be perturbed by; for example, the default perturbation distribution of 32768 means that when a fault is injected – and hence a variable is perturbed – the injection will neither increment nor decrement that value by more than 32768.

By default, all injections performed by FITMVS are randomly assigned, using a normal distribution. Should the user wish however, FITMVS can perturb variables by a value based upon a Gaussian probability distribution, described as follows:

$$f(x) = \frac{1}{\sigma \sqrt{2\pi}} e^{\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right)} \quad \begin{array}{l} \mu = \text{mean} \\ \sigma = \text{standard deviation} \end{array}$$

This results in a bell-shaped probability curve with a width based upon the standard deviation, with the probability of a number being selected increasing as the number becomes closer to the mean value. In FITMVS, this mean value is always 0. An example Gaussian distribution is shown in figure 2 below. 99.9% of all values generated by the Gaussian function will fall within 4 standard deviations of the mean, and so FITMVS allows the user to specify a standard deviation of up to 25% of the perturbation distribution. For example, should the perturbation distribution be set to 32768, then the maximum allowed Gaussian standard deviation is 8192. This ensures that 99.9% of possible values outputted by the function will fall between +32768 and -32768. Any that fall outside of this value are rounded to the nearest maximum (either positive or negative). This means that the probability curve will have a small up-turn on large standard deviations, but this should be negligible.

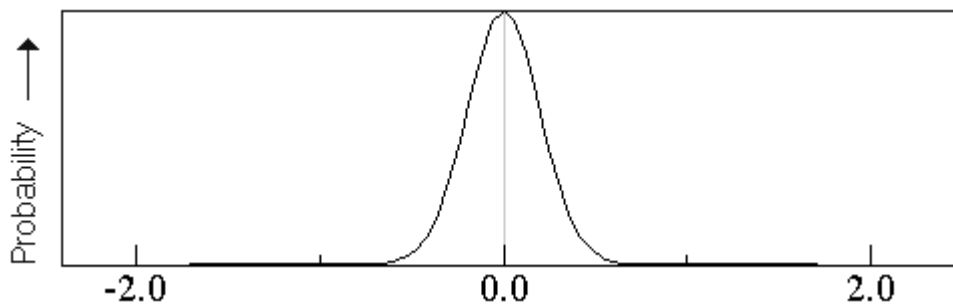
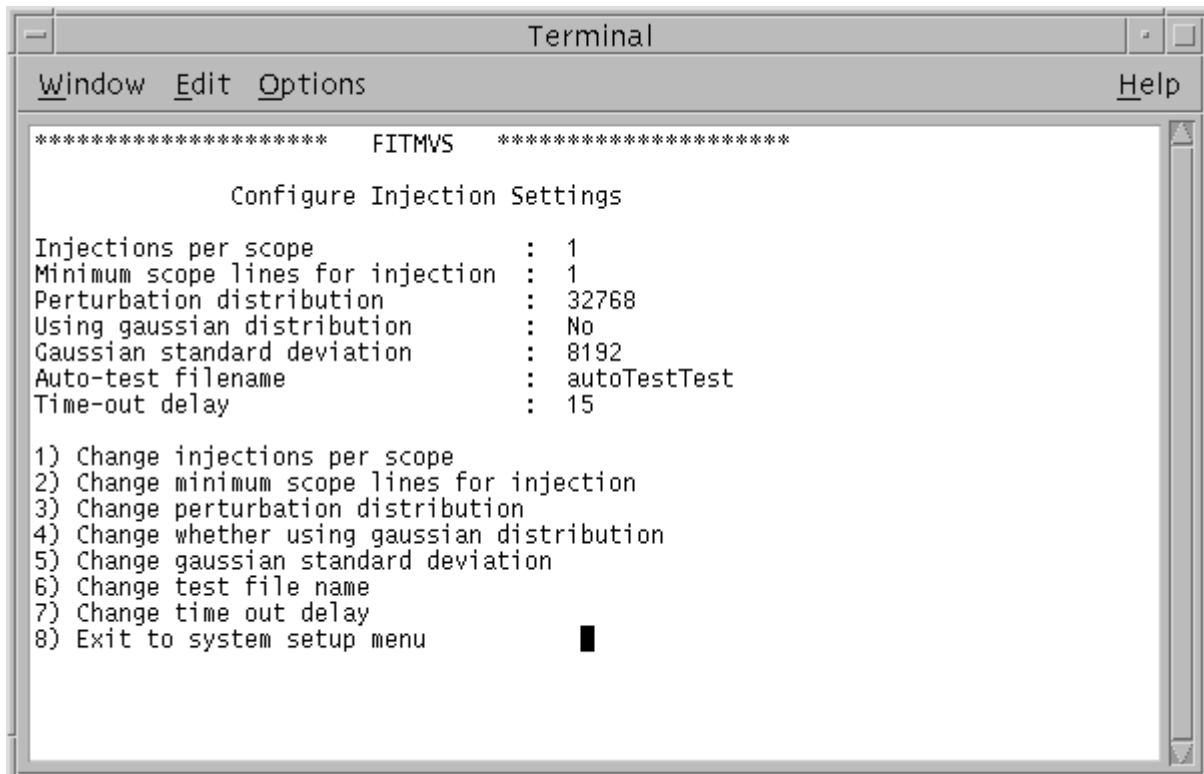


Figure 2 – A Gaussian probability distribution curve with a standard deviation of 0.2 and a mean of 0.0

The purpose of including Gaussian probability distributions in FITMVS is to further the scope for statistical analysis of data outputted by the system; varying the standard deviation will force FITMVS to perturb variables by different ranges, and so it may be of interest to see if a relationship between the size of perturbations (i.e. the standard deviation) and the sensitivity of a system exists.

The injection settings menu of FITMVS is shown in figure 3 below. FITMVS is written in using GNU C++, and has been tested and executed successfully on both UNIX (Sun Solaris) and Linux systems.



```
Terminal
Window Edit Options Help
***** FITMVS *****
          Configure Injection Settings
Injections per scope      : 1
Minimum scope lines for injection : 1
Perturbation distribution : 32768
Using gaussian distribution : No
Gaussian standard deviation : 8192
Auto-test filename       : autoTestTest
Time-out delay           : 15

1) Change injections per scope
2) Change minimum scope lines for injection
3) Change perturbation distribution
4) Change whether using gaussian distribution
5) Change gaussian standard deviation
6) Change test file name
7) Change time out delay
8) Exit to system setup menu
```

Figure 3 – FITMVS Injection settings menu

4 The Target System

In order to test the effectiveness of FITMVS, a suitable target system was needed. The application chosen was that of a simulation of a factory production cell, used in the author's previous studies of multi-version systems [TOW01a]; this is because code for a multi-version controller system for the simulation was already written and hence would require no

development. The production-cell consists of two conveyor belts, one of which delivers the raw units (blanks) into the system, and one of which moves the blanks out of the system once they have been fully processed. The unit also consists of four separate workstations, each of which has its own number. Depending on the type of a workstation, it can either be switched on and off by the controller software, or is permanently on. Two cranes are mounted on a racking which prevents them from both occupying the same horizontal position at the same time, and are used to transport blanks around the system. Each blank has its own bar-code, which will identify which workstations it needs to be placed in, and the minimum and maximum amounts of time that it could spend within each workstation. Blanks may be processed either in specific (preserved) order, or in any (non-preserved) order, depending on the instructions in the bar-code.

The major practical limitation to using the system was that the main simulation is written in the Java language, and is both relatively slow and error-prone. In order to speed the simulation up, and therefore allow more tests to be performed, the simulation program was completely re-written in C++. Only two of the three MVD channels written to control the simulation were tested using FITMVS, as the third channel was written in Java; although FITMVS parses Java correctly, additional work needs to be performed before it can accurately inject faults into Java code.

The two controller channels make great use of the system timer when waiting for blanks to process, and hence the time taken to perform tests is rather longer than we would like, and hence the number of tests that could be performed on the channels was reduced. In total, FITMVS performed 5 tests following each injection into the target systems, with each of the five tests testing a different part of the MVD system's specification.

In spite of the minimal number of tests per injection, testing still took a considerable amount of time. In order to gain as large a set of results as possible, a total of 14 SPARC workstations were used in parallel, each workstation injecting faults and testing a different copy of the MVD system.

5 Results and Analysis

In total, more than 20,000 tests were performed on the MVD system – more than 4.5 times that of last years investigation with the multi-version system. All tests were performed with the perturbation distribution set to 32768. Both normal and Gaussian probability distributions were employed, and a total of 5 injection-cycles were performed for normal distributions, and Gaussian distributions with standard deviations of 8192, 4096, 2048, and 1.

At the end of each test-cycle, the resultant log file produced by FITMVS was saved and analysed; these list every single injection and test performed, together with the results of the test. From analysis of these log files, a picture of overall sensitivity to fault is created for each channel, together. The raw figures for this analysis are shown in figure 4 below. Each row represents a complete injection-cycle; “procName” refers to the name of the channel, “PD” refers to the perturbation distribution, “G-SD” refers to the standard deviation of the Gaussian distribution (if applicable), and “Sensitivity” is the percentage chance of a test failing as a result of a fault being added. The final two columns in each table refer to the standard deviation of the sensitivity values (not to be confused with the Gaussian distribution’s standard deviation) and the average sensitivity for each set of 5 injection-cycles respectively.

procName	PD	G-SD	Sensitivity	SD	Average
Channel A	32768	8192	15.25424		
Channel A	32768	8192	24.91909		
Channel A	32768	8192	19.34426		
Channel A	32768	8192	19.6825		
Channel A	32768	8192	20	3.43229	19.840018
Channel A	32768	4096	24.29022		
Channel A	32768	4096	20.63492		
Channel A	32768	4096	25.23077		
Channel A	32768	4096	19.0476		
Channel A	32768	4096	15.9874	3.80095	21.038182
Channel A	32768	2048	21.93548		
Channel A	32768	2048	16.19048		
Channel A	32768	2048	21.29032		
Channel A	32768	2048	16.129		
Channel A	32768	2048	14.9206	3.26069	18.093176
Channel A	32768	1	20.73579		
Channel A	32768	1	19.72318		
Channel A	32768	1	25.53846		
Channel A	32768	1	17.74194		
Channel A	32768	1	23.49206	3.08736	21.446286
Channel A	32768	None	21.84615		
Channel A	32768	None	20.96774		
Channel A	32768	None	16.8254		
Channel A	32768	None	21.5873		
Channel A	32768	None	21.63009	2.1194	20.571336

procName	PD	G-SD	Sensitivity	SD	Average
Channel B	32768	8192	12.7778		
Channel B	32768	8192	14.07407		
Channel B	32768	8192	16.11111		
Channel B	32768	8192	12.5925		
Channel B	32768	8192	17.037	1.988484	14.518496
Channel B	32768	4096	14.81481		
Channel B	32768	4096	13.33333		
Channel B	32768	4096	16.48148		
Channel B	32768	4096	16.1111		
Channel B	32768	4096	13.4328	1.462952	14.834704
Channel B	32768	2048	13.40782		
Channel B	32768	2048	14.62963		
Channel B	32768	2048	12.40741		
Channel B	32768	2048	15.92593		
Channel B	32768	2048	15.9259	1.552964	14.459338
Channel B	32768	1	11.66667		
Channel B	32768	1	16.11111		
Channel B	32768	1	15.58442		
Channel B	32768	1	10.37037		
Channel B	32768	1	16.48148	2.816664	14.04281
Channel B	32768	None	13.72913		
Channel B	32768	None	16.2963		
Channel B	32768	None	11.50278		
Channel B	32768	None	15.95547		
Channel B	32768	None	16.85185	2.223737	14.867106

SD of SDs: 0.628739061
Average SD: 3.140140299
SD of Averages: 1.319274278
Average of averages: 20.1977996

SD of SDs: 0.548860549
Average SD: 2.008960344
SD of Averages: 0.33463212
Average of averages: 14.5444908

Figure 4 – FITMVS Sensitivity Analysis

As can be seen,

[BUT93] R.W. Butler, G.B. Finelli, “The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software”, IEEE Trans. on Soft. Eng., vol. SE19 no. 1, pp. 3-12, January 1993