STATISTICAL FAULT INJECTION AND ANALYSIS AT THE REGISTER TRANSFER LEVEL USING THE VERILOG PROCEDURAL INTERFACE

By

Corey Toomey

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Electrical Engineering

May, 2011

Nashville, Tennessee

Approved:

Professor Bharat L. Bhuva

Professor William H. Robinson

ACKNOWLEDGEMENTS

First, I would like to thank my advisor, Dr. Bharat Bhuva for his patience and assistance with this project and thesis. Dr. Bhuva has guided me every step of the way and without his assistance, I am not sure if this project would have been as successful as it has been. I would also like to thank Dr. William H. Robinson for his guidance and assistance with this project as well. I would also like to acknowledge Brian Sierawski for the use of his fault injection code and Andrew Sternberg for his assistance and patience with this project. I've also had many friends who have helped with this project including Daniel Limbrick and Cody Dinkins and would like to think they for their advice over the years. The Radiation Effects and Reliability group at Vanderbilt have also been a great help and source of guidance throughout my graduate career.

Finally, I would like to thank my mother, Judy Toomey, and my late father Ronald Toomey. Their endless support and the mindset that they instilled in me have made me the person that I am today. Without the appreciation for knowledge that they gave me, I would not have reached the goals that I have.

TABLE OF CONTENTS

Page

Chapter

Appendix

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Single-event effects have become a major challenge in the design and verification of microprocessors and application-specific integrated circuits (ASIC). These effects stem from particle strikes on sensitive regions of a microelectronic circuit. These highly energetic particles are typically protons and neutrons from cosmic rays, alpha particles from package decay, or other heavy ions [1]. These particles can strike the drain or source of a transistor, which causes the diffusion of electron-hole pairs creating current and interfering with the operation of the transistor. This transistor interference could invert the state of flip-flops and introduce faults into a circuit's operation [2]. These faults are called soft errors since they do not cause a permanent error in the flip-flop.

While this effect was typically thought to be only a concern for space-based electronics, it is now becoming a major concern in the terrestrial market. Studies show that soft errors are an increasing trend in relation to technology scaling [3-4]. A number of companies, including Sun Microsystems and Fujitsu, have done studies and taken measures to try and harden their designs against soft errors [5-6]. There are a variety of techniques that are available to deal with soft errors. These range from radiation hardened by design (RHBD) techniques at the circuit level to redundancy at the architectural level [7-8]. No matter what technique is used for radiation hardening, a penalty must be paid in one or more areas including die size, power, performance, and design time. Therefore it is not economical to harden every transistor in a circuit or add redundancy to every architectural structure in a design. A tool or methodology must be used by the designer to decide what structures, or modules, of a design should be hardened given the drawbacks mentioned previously. The tool or methodology should determine the possibility that a given fault will produce an error at the output pins, also known as the design's architectural vulnerability factor (AVF).

This thesis presents a methodology to determine the architectural vulnerability factor of a design and its sub-modules along with other metrics including error latency and error persistence. The designs used as test beds for this thesis were an ASIC design with 1.1 million register elements and an eight-bit microprocessor. These designs are written in verilog at the register transfer level. The tools used for fault injection and error detection were written in C/C++ and used the verilog procedural interface. It was found that by using the methodology laid out in this thesis, that a designer could use the data found during simulation to rank the sub-modules of a design by a vulnerability factor. A designer could then use this ranking to selectively harden the design.

The organization of this thesis is as follows. Chapter II gives a detailed background into metrics pertaining to this study as well as the reasoning behind simulating at the register transfer level. Chapter III is an overview of previous work on the topic of fault injection and computing AVF. Chapter IV gives a brief description of the goals of this project as well as a description of the simulation methodology and software modules used for this thesis. Chapter V presents a brief description of the devices used for testing as well as results and discussion of the simulations carried out using this tool. Chapter VI summarizes the work and presents possible future applications of the tool.

CHAPTER II

SINGLE EVENT AND DESIGN LEVEL BACKGROUND

Single-Event Metrics

The error budget of a design is typically expressed in terms of Mean Time Between Failures (MTBF) [9]. This is a metric that is used to represent the mean time between expect failures for a given type of error. The error metric that we will be using in this thesis is the failure-in-time (FIT) rate. This is inversely related to MTBF, and one FIT equals one failure in a billion hours. With respect to this study, the FIT rate equation that is used is represented by equation 1.

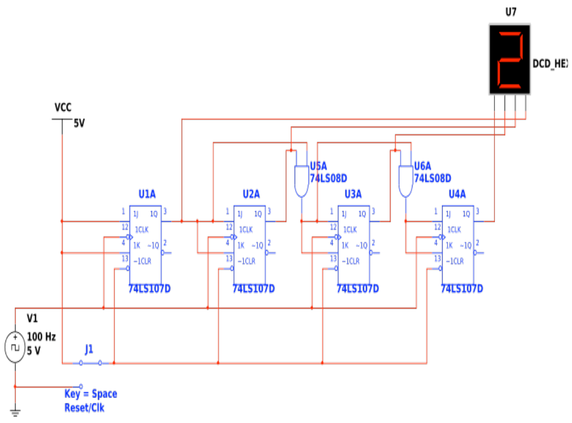
$$\mathbf{FIT \propto \phi \cdot \sigma_{bit} \cdot n_{bits} \cdot AVF} \quad \mathbf{(1)}$$

The term ϕ in Equation 1 represents the particle flux in the environment with which the device under test will be placed. The term σ_{bit} is the cross section per flip-fop and n_{bits} represents the number of flip-flops in the design or module. These three parameters are either set by the environment or set by the design and technology. The last parameter, which is the architectural vulnerability factor(AVF), is a metric that the designer has some control over and is the one which the methodology in this thesis will help to determine. The AVF represents the probability that a visible error will occur at the output given a bit flip in a register or storage cell. In our case, AVF is represented as the number of errors detected divided by the number of faults injected into the design.

Given the size and function of today's ASIC and microprocessor designs, not every fault results in a detected error. This stems from both logical and temporal masking. Logical masking represents the scenario where certain input stimulus will put the design in such a state that the fault is benign or not latched at the time of injection. Temporal masking represents the scenario where a latch is not accepting data the time of fault injection and thus the fault is not latched. This is also referred to as the timing vulnerability factor of a latch.

Register-Transfer Level

The design used for this study is written at the register-transfer level (RTL) as are many used in previous fault injection studies (eg. [10-11]). A register-transfer level description of a circuit uses the flow of signals between registers and the operations performed on those signals to represent the operation of the design. This level of abstraction is used in hardware description languages such as verilog and VHDL. Figure 1 shows the conversion of going from a gate/block level description to an RTL based one.



```
module 4_bit_counter(sys_clk, count_out);
input sys_clk;
output [3:0] count_out;
reg count_out;

always @ (posedge sys_clk) begin
        if (count_out == 4'b1111) begin
                count_out <= 4'b0000;
        end
        else begin
                count_out <= count_out + 1'b1;
        end
end
endmodule
```

Figure 1: Gate level description compared to RTL description

The benefits of using this level of abstraction for fault injection studies are faster simulation time, smaller simulation space, and the fact that an RTL model is independent of technology, cell libraries, and physical circuit descriptions. The speed-up in simulation time comes from the lack of having to simulate every gate of a design. The registers are the only true structures in an RTL description. Also, many verilog and VHDL simulators are optimized for RTL descriptions. The simulation space is smaller because you simply have less possible fault injection points as compared to the nodes in a gate level description. The only existing fault injection points are register bits instead of every single input and output node of a gate description. Also the RTL description is available earlier in the design cycle when compared to a gate level description. Problems related to AVF are much easier to fix when detected earlier in the design cycle.

CHAPTER III

AVF METHODOLOGY BACKGROUND

Computing AVF

There are three main techniques for computing the AVF of a design: analytical models, performance models and statistical fault injection [12]. The latter will be presented first as statistical fault injection was used in this study and can be broken down much further. All of the techniques essentially do the same thing; they try to determine the effect of faults on a design. However they all use different methodologies to calculate this effect.

Analytical Model

Using an analytical model requires the use of architecturally correct execution (ACE) bits. These bits will alter the final outcome of the program if they are changed [9][12]. This model uses a formula known as Little's Law which can be translated as $N = B \times L$ [13]. *N* represents the average number of bits in the structure, *B* is the average bandwidth per cycle into the structure, and *L* is the average latency of an individual bit through the structure. Using ACE bits, you can then compute the AVF by the formula $(B_{ace} \times L_{ace})/N_{ace}$. This technique has an advantage in that it can be used in the early stages of a design when an RTL description may not be available. The downsides to using this technique are the need for a significant amount of knowledge about the design as well as the fact that all of the bits used in this technique must flow unmodified and without duplication through the design under test [12]. Given these drawbacks, this technique can only be used in select cases.

Performance Model

The use of a performance model to compute AVF also uses the ACE bits mentioned in the analytical model section. This model is described in detail in the Mukherjee paper [9]. The key to this analysis is the determination of the fraction of time a bit is an ACE bit. The analysis assumes a bit is ACE unless it can be proven to be un-ACE. They determine if a bit is un-ACE by using a set of rules described in [9]. The performance model's main positive point is its speed. AVFs for many structures can be computed in parallel as well as the fact that a performance model can realistically be run for a larger number of simulations than other models in a given amount of time. The main drawback for the performance model technique is the amount of knowledge that must be known about the design. To determine whether a bit is ACE or un-ACE, the person running the experiment must be an expert in both the design's architecture and microarchitecture.

Statistical Fault Injection

Statistical fault injection (SFI) is the most widely used and accepted method for measuring AVF. This technique uses a randomized bit flip to introduce a fault typically in an RTL or gate level design description. The simulation is then run either to completion or to a determined point. An error check is done by either checking the architectural state of the model with the state of an error-free model or checking the output pins against the output pins of an error-free model. This check is done either in real time or at the end of simulation. SFI's main positive point is that it requires very little, if any, knowledge of the design before the simulations. The largest drawback to SFI is the simulation time. Since SFI is typically applied to RTL or gate-level descriptions, the simulation time is typically much longer than a performance based model. Also considering that SFI is just flipping a bit and checking against a golden copy, the simulation space is also very large. Thus a large number of simulations are needed.

Since this thesis proposes a tool that uses SFI, it can be broken down further into three separate versions of SFI. These are physical fault injection, software fault injection and simulated fault injection[14].

Initially physical fault injection was the method of choice for SFI. This method injects faults directly into the hardware either through modifying the value of pins or disturbing the working environment. This technique has one positive in that once the test setup is prepared, the simulation time is very fast. Hardware based simulations are significantly faster than software. While physical fault injection is very fast, it has many drawbacks. The initial time for set up of the test devices and cost are very prohibitive. While the use of FPGA's for this technique has lowered both time and cost, it is still very time prohibitive to synthesize the design on the FPGA and modify the design for fault injection. A few examples of hardware fault injection are MESSALINE[15], RIFLE[16], and FIST[17].

Software fault injection or software implemented fault injection (SWIFI) emulates a corruption in the software running on the device. This is done through memory or register manipulation. This technique allows control of the injection place and the simulations are also easily reproducible. Because this method corrupts the software on the device, there are limited injection locations due to the software only accessing particular parts of the design. Timing constraints are also a drawback due to command execution times. The implementation of SWIFI can be seen in papers on FERRARI[18], DOCTOR[19], and Xception[20].

Simulated fault injection models the entire system behavior using simulation. This technique makes primary use of hardware description languages such as Verilog or VHDL. This technique is widely used because it allows access to all the component models. A typical simulation-based technique can be categorized either as a code-modification technique or a simulator-command technique. Code-modification techniques actually modify the design source code for fault injection. This can be done through using either saboteurs or mutants. A saboteur technique is based on adding components to the design to allow fault injection, while the mutant technique is based on replacing components with structures that are created to allow access to nodes for fault injection. Since both the saboteur and mutant technique modify the source code they have a major drawback. Modification of the source requires recompilation of the design. Given the large number of simulations needed for SFI, this recompilation time can be detrimental. Code modification techniques can be seen in papers on MEFISTO[21], VERIFY[22], and SINJECT[23].

Simulator-command based techniques rely on built in simulator commands to modify the register values of a design. This technique is typically VHDL based, though some verilog simulators do allow this technique. Since simulator commands are used the source code is not modified and a recompilation is not necessary. However, accessibility of injection space is limited and the lack of portability between simulators is a major drawback. This technique can be seen as part of the VFIT based approach [24].

Another technique, which is part of the simulation based fault injection family, is the use of the verilog procedural interface for use in fault injection and error detection. However this technique does not fit in either the code-modification or simulator-command based approaches. Since this is the technique used in this thesis, it will be described in more detail in the next section. This technique is also used by D. Kammler for fault injection and error analysis[14].

CHAPTER IV

SIMULATION METHODOLOGY

Main Goals

Goals were set for the project to determine the path of development of the analysis tool. The main goals were:

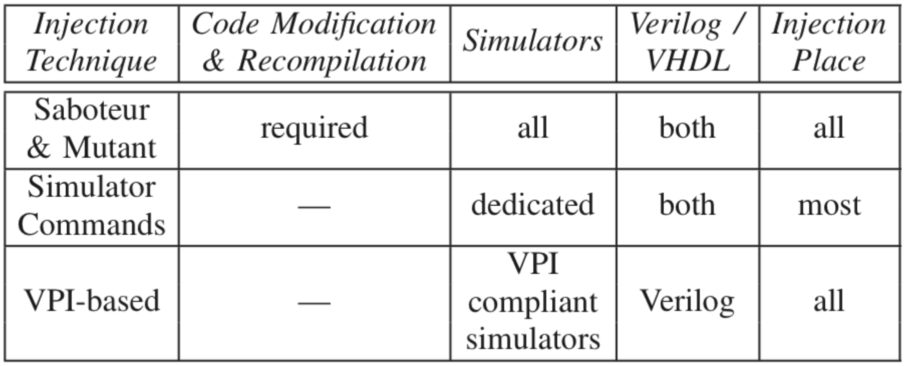
1. Non-intrusive to the design code
2. Portable to any simulation or verification environment
3. Requires no knowledge of design function to implement the fault injection and error detection code
4. Uses existing verification test benches

These goals were decided upon and were used to guide the design of the fault injection and analysis tool. Goals 1 and 2 led to the use of the verilog procedural interface.

Verilog Procedural Interface (VPI)

VPI is an interface for the verilog HDL that uses the C or C++ programming language. VPI consists of access and utility routines that you call from standard C programming functions. One can then use these routines to interact with instantiated verilog design objects. Table 1 from the Kammler paper shows why it would be advantageous to use VPI over the other methods described for simulation based fault injection[14].

Table 1: Comparison of simulation based fault injection techniques



Injection Technique	*Code Modification & Recompilation*	*Simulators*	*Verilog / VHDL*	*Injection Place*
Saboteur & Mutant	required	all	both	all
Simulator Commands	—	dedicated	both	most
VPI-based	—	VPI compliant simulators	Verilog	all

The use of VPI allows this technique to get around the major downfalls of code-modification and simulator-command techniques. VPI requires no recompilation and is not simulator specific as long as the simulator supports VPI.

Simulation Approach and Software Modules

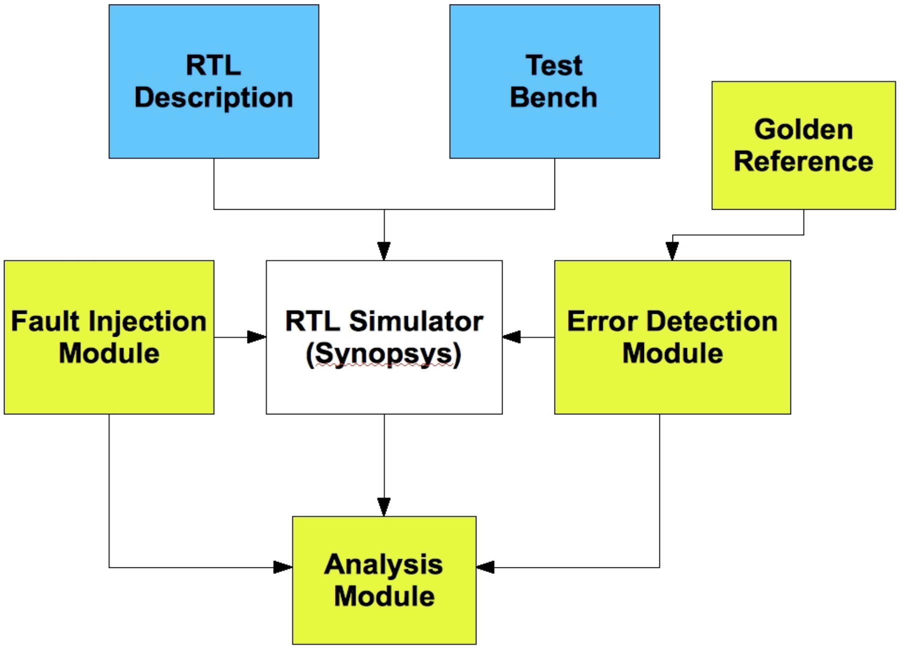


Figure 2: Simulation Approach Flow Chart

Figure 2 represents the simulation approach for this thesis. The blue boxes represent what is supplied by the design. This includes both the source RTL description and one or more test benches. The white box represents the verilog simulator. In this thesis, the Synopsys VCS tool was used. However this methodology is known to be compatible with Icarus verilog simulator as well as Cadence's NCVerilog simulator. The yellow boxes represent what this methodology brings to this simulation approach. These are the fault injection module, the error detection module, the golden reference file, and the analysis module. A verilog implementation example is shown in Appendix C.

Brian Sierawski of the Institute for Space and Defense Electronics wrote the fault injection module. It is designed to inject faults into storage elements randomly and uniformly across all sub-circuits and length of the test bench. It is implemented using C++ with VPI and is compiled using gcc. The source code is not openly available for this software module, thus it will not be a part of the appendix. This module consists of four verilog calls: *singleEventMaxMemorySize("value")*, *singleEventInit()*, *pseudoRandom("value")*, and *singleEventUpset(DUT)*. The *singleEventMaxMemorySize("value")* call allows the user to set the max size of the registers that will be targets for fault injection. It was assumed with our simulations that any registers over the size of 2048 bits were memory arrays and had some type of error correcting codes. The *singleEventInit()* call initializes the single event software module as well as sets the seed for any randomization needed for the fault injection module. It also outputs design related data that is needed for analysis such as the number of register bits contained in each sub-module of the design. The *pseudoRandom*("value") call is used to create a random number between 0 and the number passed to the call. This is used to obtain a random time for fault injection. Lastly and most important, the *singleEventUpset*(DUT) call will randomly choose a register bit in the design and flip its value which simulates a soft error. This fault is then logged to standard out with fault location, fault time, and how the fault was manifested (e.g., 0→1, 1→0). These are all called within an initial block in the test bench or verification wrapper.

The error detection module is designed to compare outputs of the faulty circuit with a golden reference file. This methodology allows the user to have real time detection of errors as well as real time detection of error corrections. This software module also produces the golden reference copy by logging the output stream during a simulation with no fault injection. The error detection module was written in C with VPI and sections of code were used from Sutherland[25]. The source code is available as Appendix A. There is only one verilog call associated with the error detection module, which is *create_output_log*(DUT). This call is made within a verilog always block that is triggered by the rising and falling edge of a given clock. This call has two possible uses. The first is used with no fault injected and set up in such a way that it produces a golden reference copy of all the output pins of the given DUT on the rising and falling edge of every clock cycle. After that golden reference file is created the error detection simulations can begin. The fault injection and error detection modules are now both working during each simulation. The error detection module will log the outputs at the falling and rising edge of every clock cycle and then compare those against the golden copy. If a mismatch is detected, it will log the error pin location and error time. The simulation will also keep running whether an error is detected or not. At this point the error detection module can also log whether an error is corrected or, if an error is not corrected, it will detect if it persists past a certain number of clock cycles. The error detection module logs all of this to standard out.

The analysis module is a script written to compile information from the simulation runs. This module is used to log: fault locations, fault times, error locations, error times, error corrections, error correction times, and test bench failures. Error latency between fault injection and error detection is also logged. Test bench failures are logged to determine internal state corruption. This raw data can then be compiled and analyzed by the user to determine vulnerability and error severity.

Both the fault injection and error detection modules are used at simulation time and the analysis module is used at the finish of simulation to compile the raw data into log files. Each simulation represents a single fault injection. Multiple fault injection is possible with this tool; however it was not done for this thesis.

CHAPTER V

RESULTS AND DISCUSSION

Designs Used for Test

The designs used for test in this thesis are a large, complex, and sequential design comprised of over 1.1 million register nodes and a small eight-bit microprocessor design comprised of 447 bits. To comply with the goals in chapter IV, not much is known about the function of these designs.

One full simulation of the large ASIC design, which represents one fault injection, takes approximately one hour and represents approximately 32000 clock cycles of simulation time. A total of 1400 simulations were completed in a reasonable amount of time and the data was compiled. In an ideal situation, one would run more simulations. However, with limited time and limited computing resources, there was a limit to how many simulations could reasonably be finished. Three levels of the design were analyzed. Level one is the core sub-modules e.g. Module1, and two more levels below that representing the sub-modules of level one modules.

One full simulation of the eight-bit microprocessor design takes approximately 3 seconds. A total of 50,000 simulations were completed to provide data for an over sampled test case. Only level one of this design was analyzed as it is used to provide data to illustrate the results provided by the methodology presented in this thesis.

Results

The following charts and analysis represent the compilation and analysis of the data compiled from the simulations described above. These figures and charts will represent all of the level-one analysis as well as one of the level-two and level-three analyses for the large ASIC. The data presented for the large ASIC is illustrative of the results obtainable using this methodology. Due to time and computing constraints, the data is not sufficient for adequate coverage. Figure 3 below represents a graph that proves the fault injection module can randomly injected faults among the range of the given test bench which in this case was 32000 clock cycles for the large ASIC design.

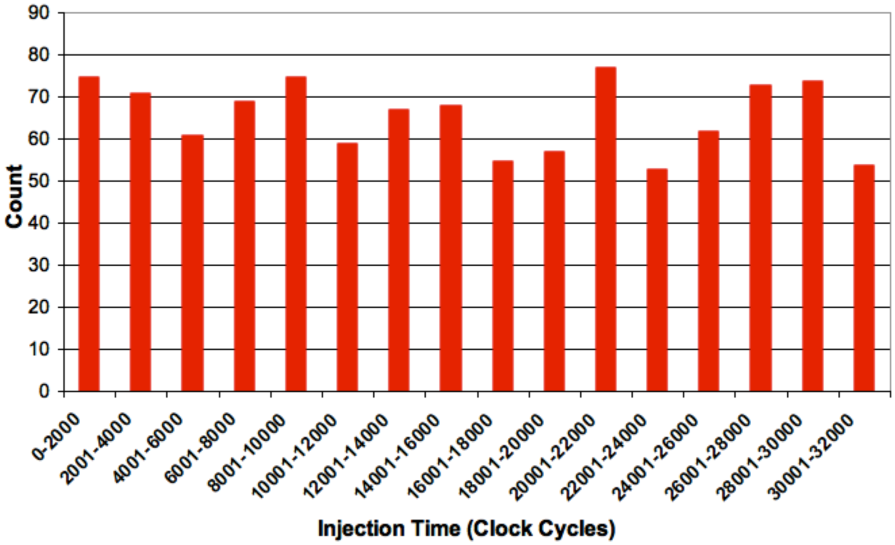


Figure 3: Distribution of faults along test bench time

The next figures are a compilation of a three data sets. These are number of bits per module, number of faults injected per module, and number of errors detected per module. Figure 4 represents the data for level one of the large ASIC and figure 5 represents this data for the microprocessor.

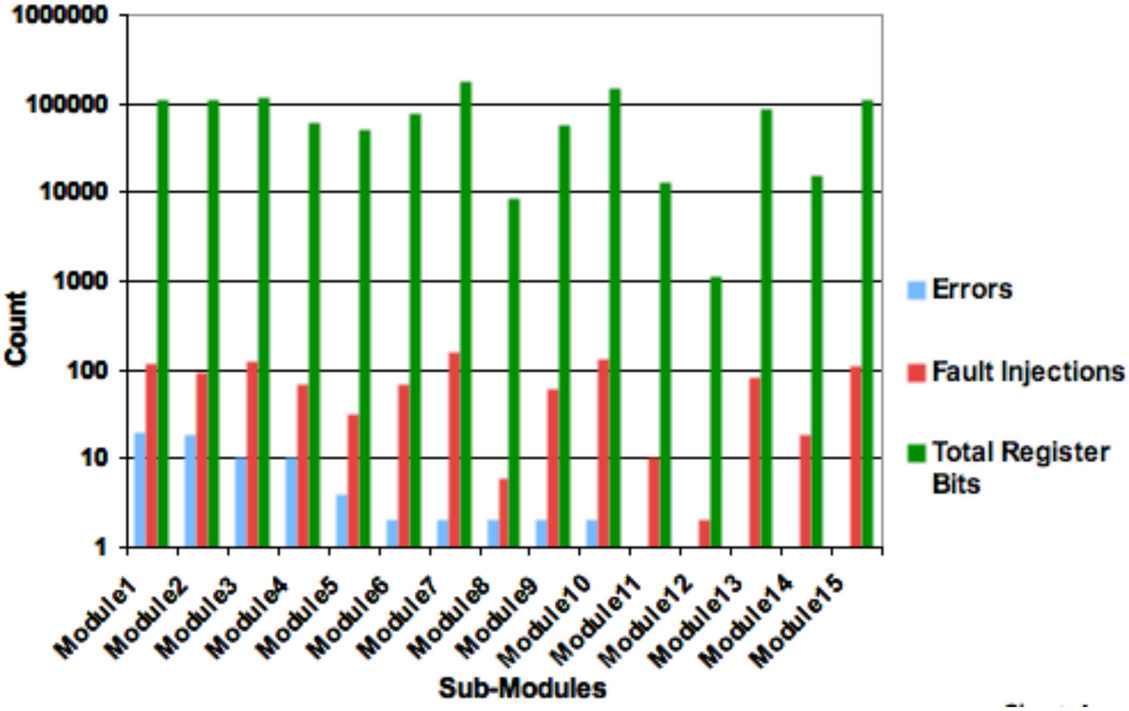


Figure 4: Level one sub-module data for large ASIC

The data in Figure 4 shows a number of trends. Firstly it shows that the randomization of fault injection is proportional to the size of the module. This is expected and is correct operation of the fault injection module. It also shows the number of errors that were detected given a fault injected in that module. From the graph one can see that both Module1 and Module2 have the largest number of detected errors. Also of note is that both Module13 and Module15 received a large number of fault injections, but no errors. This could either be because the faults were logically or temporally masked, or the given test bench simply did not exercise these modules very often.

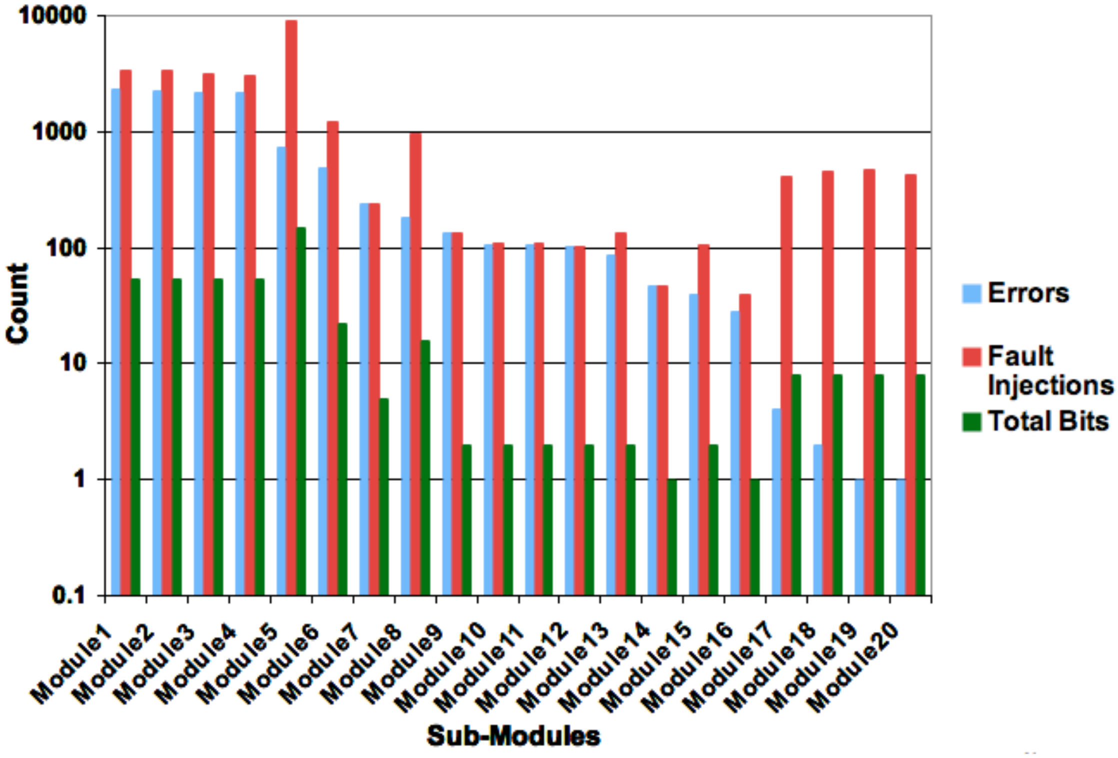


Figure 5: Level one sub-module data for eight-bit microprocessor

The data in Figure 5 shows the same trends as figure 4. However, this design had no time or computing constraints. Therefore, it has been oversampled, as evident from the number of fault injections for each module being much larger than the number of registers in that module. From the graph one can see that both Module1 and Module2 have the largest number of detected errors. Also of note is that modules 7, 9, 10, 11, and 12 show an almost 100 percent AVF.

A designer can use figures 4 and 5 to determine which modules to harden. However, it is much easier to see the most critical modules by plotting them as their number of errors versus number of injections as seen in figure 6 and figure 7.

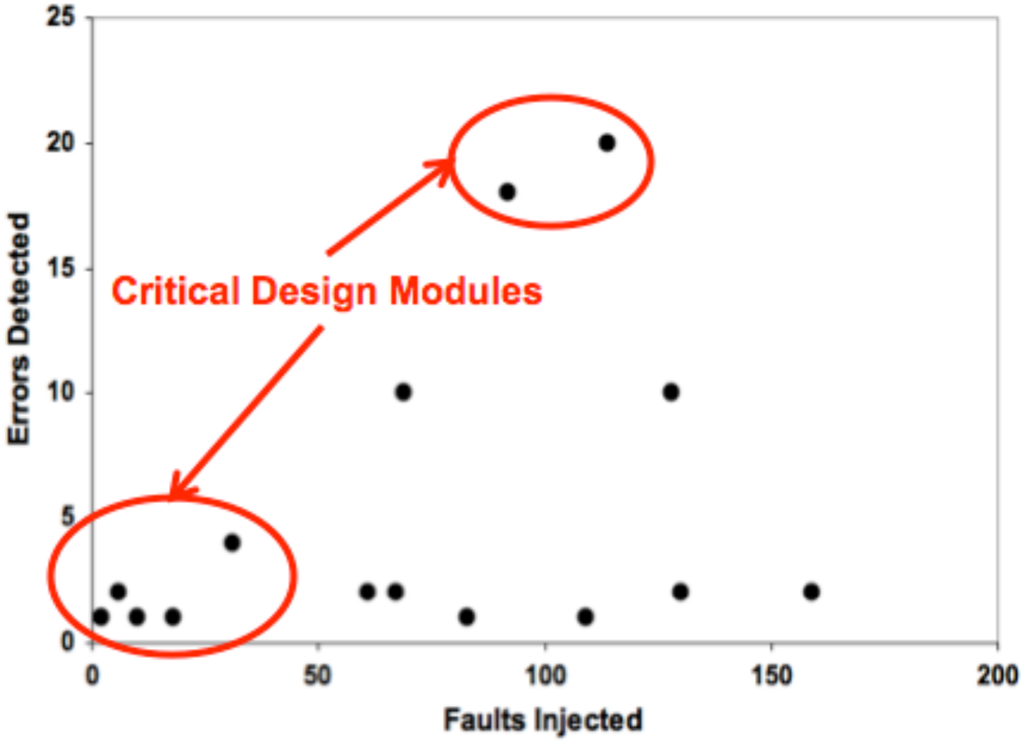


Figure 6: Error data for large ASIC modules

The data in Figure 6 is a representation of the AVF of each module in the large ASIC design. Each point on the figure represents a single module of the design. Given errors on the y-axis and faults on the x-axis, those points towards the upper left of the figure represent the most critical modules. In this instance they are being illustrated as an example of the most critical modules. A designer can decide on a given AVF budget and then determine which modules meet that budget with this chart. The same type of analysis is done for the microprocessor and is represented in figure 7 below.

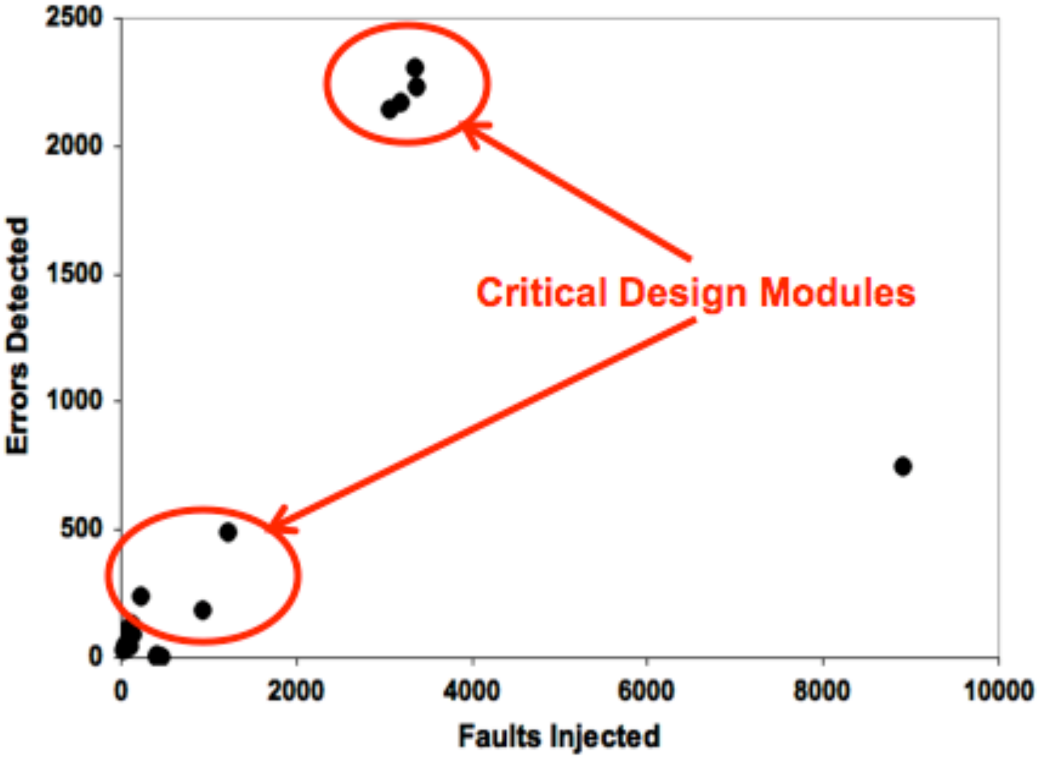


Figure 7: Error data for small microprocessor modules

One of the main goals of this methodology is to provide data to the designer so that they can selectively harden modules in the design. If a designer choose to harden Module1 of the large ASIC based on figures 4 and 6, but could not efficiently harden the entire thing based on size, they would then have to analyze the sub-modules of Module1. The figures below represent an analysis of levels below the core sub-modules. Figure 8 represents an analysis of one level down from Module1.

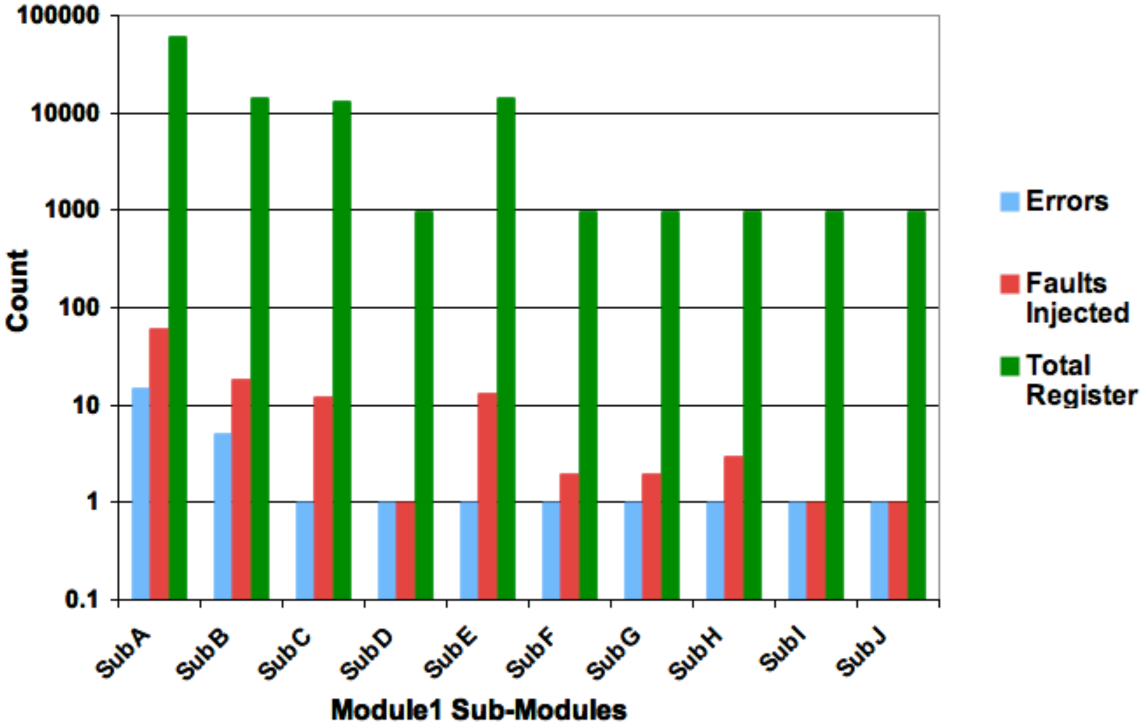


Figure 8: Module1 sub-module analysis

A designer can see from this chart that they should focus on hardening SubA, SubB, or both. However, let us assume that these modules are also too large to harden efficiently. So the designer decides to focus on SubA and selectively harden sub-modules of that module. Figure 9 represents the data analysis for the sub-modules at that level.

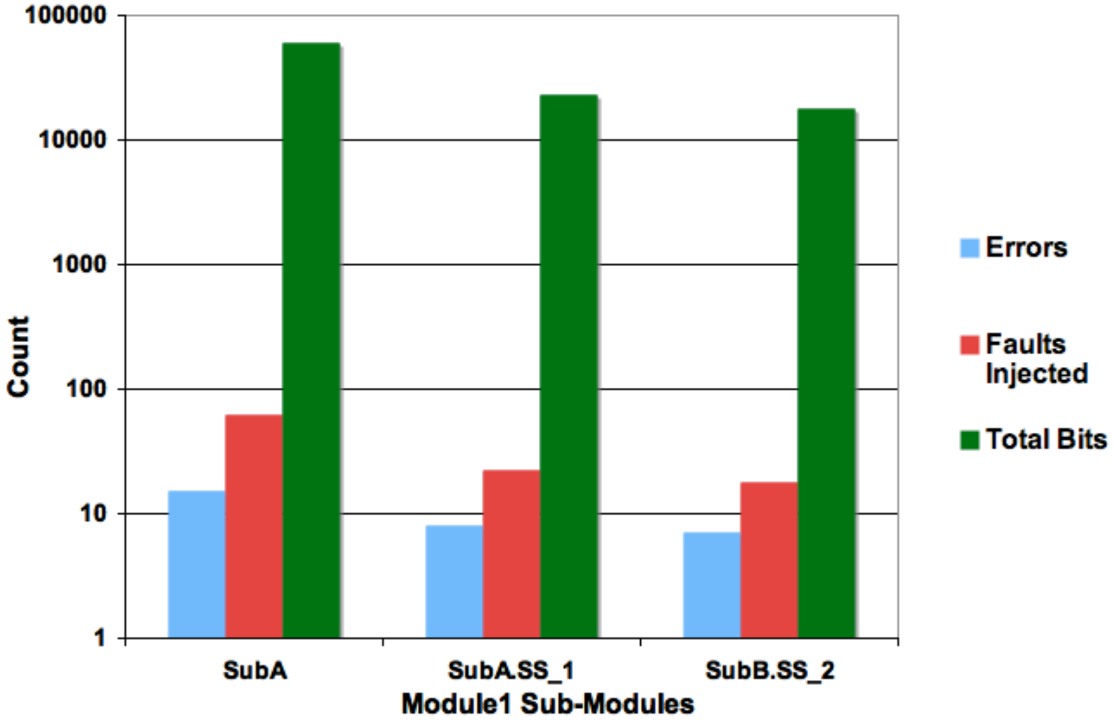


Figure 9: Level three module1 analysis

Now, with the analysis of level three, the designer can decide which sub-module to harden assuming they are now within reasonable size to harden. The designer can analyze the data as many levels down as they want all the way to the single register bit. However, the further down the data is analyzed, the more simulations are needed for statistically accurate results.

Figure 10 represents the analysis of the data for error latency for the large ASIC design. This chart does not represent all of the errors found. A timing bug was found in the fault injection software early on during the simulations and fixed. The computing resources for re-running the previous simulations were not available once the data set was finished.

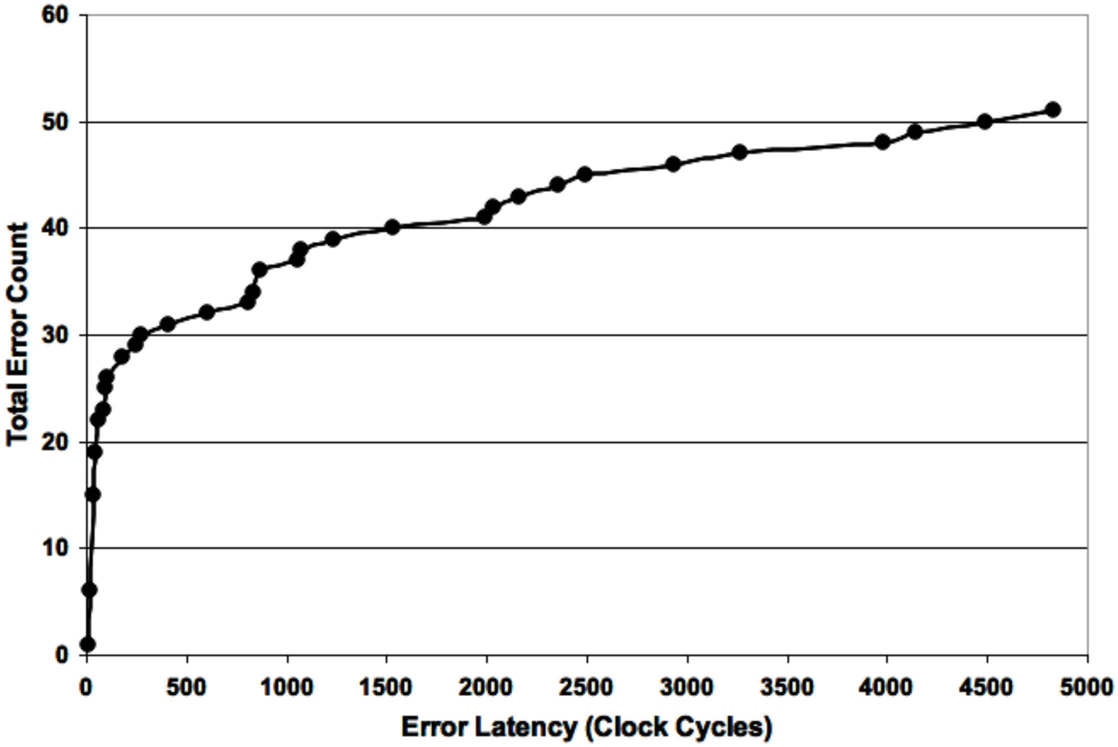


Figure 10: CDF of error latency data

This figure is not only useful for the designer in analyzing the error latency of the design and test bench but it was also useful for modifications to the simulation methodology. A proposed speed-up technique was to wait for 5000 clock cycles after the fault was injected and end the simulation if an error was not detected. It was proposed that if an error did not propagate to the output within 5000 clock cycles, it was a latent error and probably would not manifest itself at the output. However, one can see from Figure 10 that assumption was false. While the curve does certainly level out to a certain extent past 1000 clock cycles, it is still rising. Thus it was realistically possible to detect faults past 5000 clock cycles.

The next figures represent the AVF of each design as a factor of fault injection. Given the technique of statistical fault injection and the large sample size of complicated designs, it is not efficient to simulate every possible fault. Therefore a subset of the simulation space must be used to provide accurate results within an efficient amount of time. The size of that subset should be determined by figure 11 and 12 below.

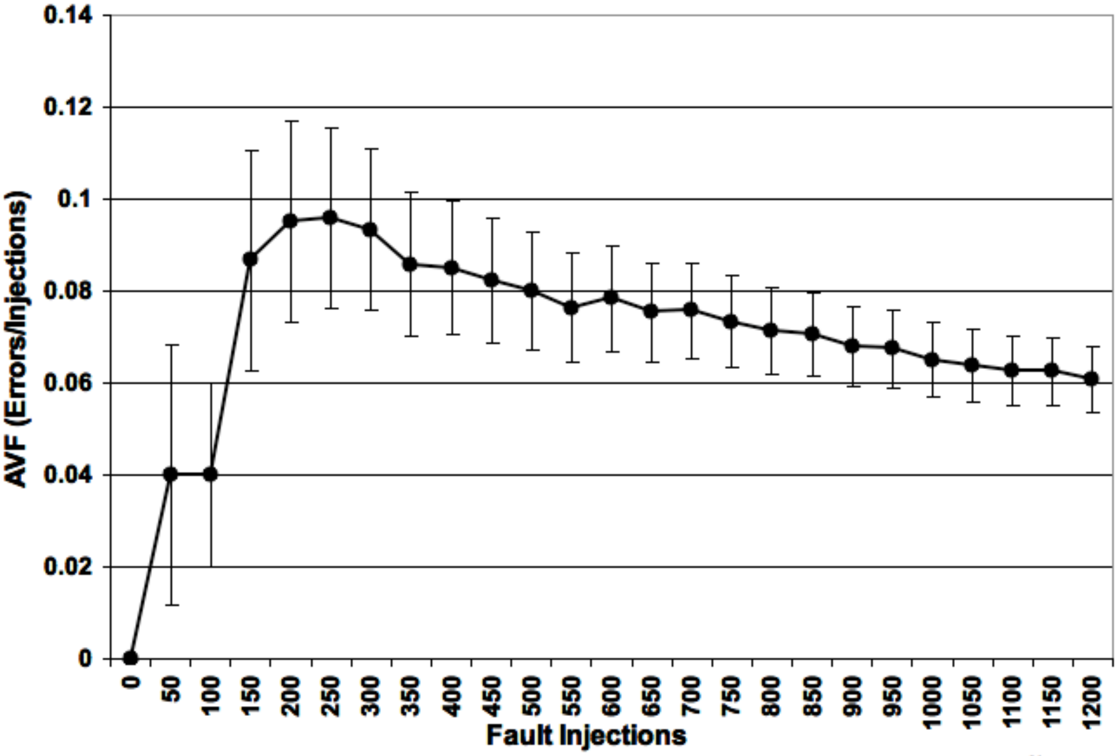


Figure 11: AVF vs. injections for large ASIC

Figure 11 represents the AVF data for the large ASIC. Since the line in figure 11 has not saturated and the error bars are still not within an acceptable range, more simulations would be needed for this design. From this graph, a designer could conclude that there are not a sufficient number of simulations to make accurate decisions for the large ASIC. However, to prove that AVF does saturate given enough simulations, the same plot was done for the microprocessor and is plotted in figure 12.

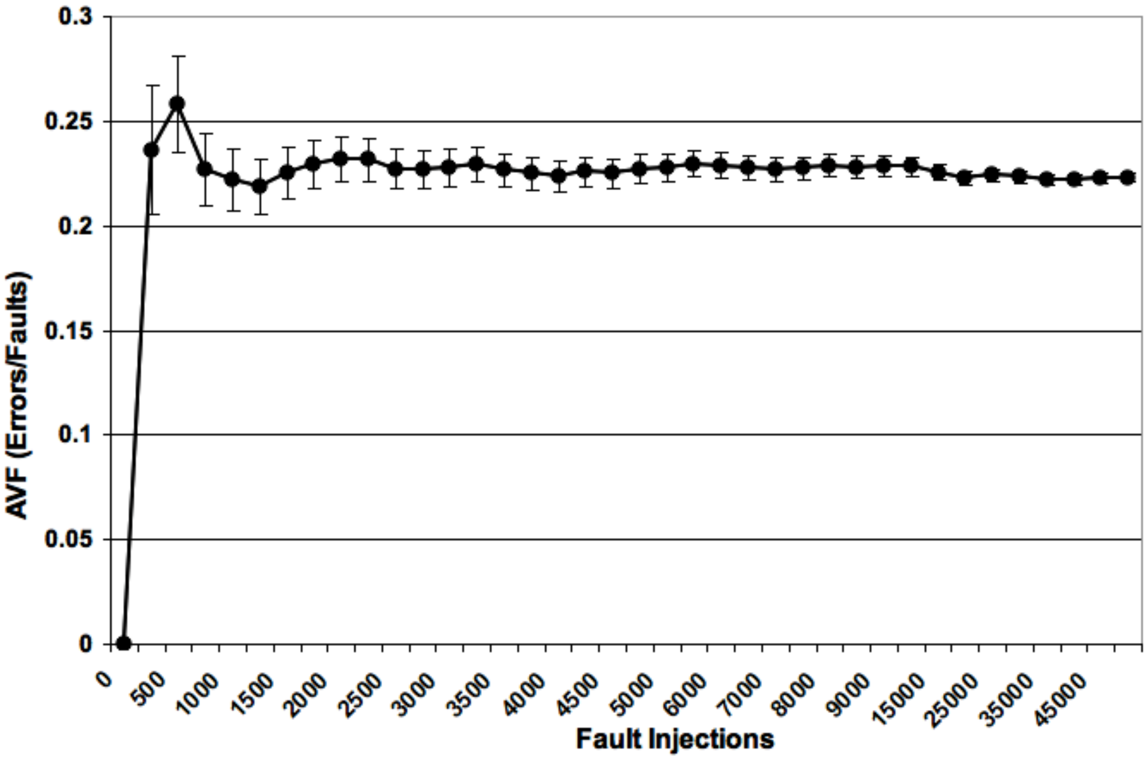


Figure 12: AVF vs. injections for eight-bit microprocessor

The AVF for the microprocessor in figure 12 has saturated and the error bars are within an acceptable range. A designer could then take the data obtained from the simulation set used to produce this graph and make accurate decisions on the susceptibility of this design. It can also be observed in this figure that a subset of the total data set for this design would have been sufficient. Had there only been 10,000 simulations, the AVF would have been saturated and the errors bars within an acceptable range. Thus, one-fifth of the data set provided was actually needed to make conclusions about the eight-bit microprocessors.

There are a few notes to take away from these results. First is that these are completely dependent upon the test bench. Given another test bench for either design, these results could have been different. It would be advised to use a test bench that exercises the design in such a way that it mimics real world operation. The second note is that these results are for a limited number of simulation runs for the large ASIC. With proper computing resources, one could obtain more data. The more data the designer has, the more accurate the results, and increasing number of simulations will give a better picture of sub-module ranking at lower levels and would have provided AVF saturation for figure 11.

CHAPTER VI

CONCLUSIONS AND POSSIBLE FUTURE WORK

This thesis presented a set of software modules and a simulation methodology for determining the AVF of a design and its sub-modules using the verilog procedural interface. The methodology is non-invasive to design code, portable to many RTL simulators, and requires little knowledge of the design. This work is intended to be used by designers to selectively harden modules of a design to reduce overall design AVF.

An expansion of the project could be done to include simulations at the gate level including stuck-at-faults and transients. Also doing a gate level and register transfer level analysis of the same design in parallel and comparing compute time and results. Also this tool could be used to compare AVF data from a design with AVF data of that same design using other fault injection methods, like hardware fault injection or analytical analysis.

APPENDIX A

ERROR DETECTION MODULE SOURCE CODE

```
/********************************************
Print Outputs VPI Module $create_output_log

Usage: $create_output_log(<module_instance>);

Code sections taken from "The Verilog PLI Handbook" by Stuart Sutherland

Version 10b: Fixed vpiPort problem with NCVerilog

Version 9: Added registration function for use with NC-Verilog

Version 8f: Added fix for unrecoverable errors

Version 8e: Added fix to stop SEU ERROR from printing and filling up hard drive space

Version 8d: Fixed output problem and able to check busses

Version 8a: Quick Fix for memory problem

Version 7: Live Error checking.

11/12/09: Added a vpiHandle array to keep store the handles
        to the outputs & inouts so that we only iterate through
           all nets and ports in col_compiletf

02/03/10: Set an environment variable COL_FIRST=TRUE to run the first
           time to avoid recompilation and changing comments

********************************************/

#include <stdlib.h>    /* ANSI C standard library */
#include <stdio.h>     /* ANSI C standard input/output library */
#include <string.h>
#include <stdarg.h>    /* ANSI C standard arguments library */
#include "vpi_user.h"  /* IEEE 1364 PLI VPI routine library  */

extern  PLI_INT32  col_compiletf(PLI_BYTE8 *user_data);
extern  PLI_INT32  col_calltf(PLI_BYTE8 *user_data);
```

```
void     PrintSignalValues(vpiHandle port_handle);

int ARRAY_SIZE = 200000;
FILE* columnFilePtr = NULL;
char* columnLocation = "./singleColumn.seq";
int i = 0;
int j = 0;
int valueArray[200000];
int HANDLE_ARRAY_SIZE = 5000;
vpiHandle handleArray[5000];
int k = 0;
int a = 0;
int build_golden=0;
int logsize=0;
int ErrFlags[5000];
int ErrIndex=0;
int ErrCount[5000];
int UnErrorCap = 1000;

/**********************************************************************
 * Registration Function
 *********************************************************************/
/*
void registration() {
        s_vpi_systf_data task_data_s;
        p_vpi_systf_data task_data_p = &task_data_s;
        task_data_p->type = vpiSysTask;
        task_data_p->tfname = "$create_output_log";
        task_data_p->calltf = col_calltf;
        task_data_p->compiletf = col_compiletf;

        vpi_register_systf(task_data_p);
}

void (*vlog_startup_routines[ ] ) () = {
        registration,
        0
};
*/
/**********************************************************************
 * Compiletf application
 *********************************************************************/
extern  PLI_INT32 col_compiletf(PLI_BYTE8 *user_data)
{
 vpiHandle systf_h, tfarg_itr, tfarg_h, port_itr, port_chk, port_h,
        net_itr, net_h, reg_itr, reg_h;
```

```
    if (!(getenv("GCPATH")==NULL))
    {
            columnLocation=(getenv("GCPATH"));
    }

    if (!(getenv("COL_FIRST")==NULL))
    {
            if ((strcmp(getenv("COL_FIRST"),"true")==0) ||
(strcmp(getenv("COL_FIRST"),"TRUE")==0))
            {
                vpi_printf ("Building golden copy in %s\n",columnLocation);
                build_golden=1;
            }
    }
i=0;
for(;;){
  systf_h = vpi_handle(vpiSysTfCall, NULL);
  if (systf_h == NULL)
  {
    i++;
    vpi_printf("Could not open output_log, retrying....\n");
    usleep(1e6);
    if (i==3)
    {
            vpi_printf("SEU ERROR: create_output_log could not obtain handle to systf
call\n");
        vpi_control(vpiFinish, 1);  /* abort simulation */
        return(1);
    }
  }
  else{
    break;
  }
}
  tfarg_itr = vpi_iterate(vpiArgument, systf_h);
  if (systf_h == NULL)
  {
    vpi_printf("SEU ERROR: create_output_log could not obtain iterator to systf args\n");
    vpi_control(vpiFinish, 1);  /* abort simulation */
    return(0);
  }
  tfarg_h = vpi_scan(tfarg_itr);
  if (vpi_get(vpiType, tfarg_h) != vpiModule)
  {
    vpi_printf("SEU ERROR: $create_output_log arg must be module instance\n");
    vpi_control(vpiFinish, 1);  /* abort simulation */
    return(0);
  }
  if (vpi_scan(tfarg_itr) != NULL)
  {
    vpi_printf("SEU ERROR: $create_output_log requires 1 argument\n");
    vpi_free_object(tfarg_itr); /* because not scanning until null */
    vpi_control(vpiFinish, 1);  /* abort simulation */
    return(0);
  }

  net_itr = vpi_iterate(vpiNet, tfarg_h);
  reg_itr = vpi_iterate(vpiReg, tfarg_h);
  port_chk = vpi_iterate(vpiPort, tfarg_h);

  if (!net_itr && !reg_itr)
  {
    vpi_printf("   No nets or registers found\n");
    return(0);
  }

  if (!port_chk)
   {
     vpi_printf("  No Ports found\n");
     return(0);
   }

  vpi_free_object(port_chk);
/*
if(net_itr){
  while(net_h=vpi_scan(net_itr))
  { port_itr = vpi_iterate(vpiPort, net_h);
    while(port_h=vpi_scan(port_itr))
    {
       vpi_printf("PORT NAME 1: %s \n", vpi_get_str(vpiName,port_h));
    }
  }
  vpi_printf("Finished with NET ITR");
}
else
 vpi_printf("NO NETS");
if(reg_itr){
  while(reg_h=vpi_scan(reg_itr))
  {
    port_itr = vpi_iterate(vpiPort, reg_h);
    while(port_h=vpi_scan(port_itr))
    {
```

```
    vpi_printf("PORT NAME 2: %s \n", vpi_get_str(vpiName,port_h));
  }
 }
}
else
 vpi_printf("NO REGS");
*/
 //net_itr = vpi_iterate(vpiNet, tfarg_h);
// reg_itr = vpi_iterate(vpiReg, tfarg_h);
 while (net_h = vpi_scan(net_itr))
 {
  port_itr = vpi_iterate(vpiPort, net_h);

  // fp = fopen("test.log","a+b");
  if (port_itr == NULL)
    continue;

  while ((port_h = vpi_scan(port_itr)) && (k < HANDLE_ARRAY_SIZE))
  {
   vpi_printf("Port name is %s\n", vpi_get_str(vpiName, port_h));
   vpi_printf("Size is %d\n", vpi_get(vpiSize, port_h));
   switch (vpi_get(vpiDirection, port_h))
   {
    case vpiInput:
      vpi_printf("    Direction is input\n");
      break;
    case vpiOutput:
     vpi_printf("    Direction is output\n");
     logsize += vpi_get(vpiSize, port_h);
       handleArray[k] = net_h;
     ++k;
       break;
    case vpiInout:
     vpi_printf("    Direction is inout\n");
       logsize += vpi_get(vpiSize, port_h);
       handleArray[k] = net_h;
       ++k;
       break;
   }
  }
  if (k == HANDLE_ARRAY_SIZE)
  {
    vpi_printf("SEU ERROR: Handle array is not large enough");
    vpi_control(vpiFinish, 1);
    return (0);
  }
```

```
   if (port_h)
   {
     vpi_free_object(port_itr);
   }
 }
 if (net_h)
 {
   vpi_free_object(net_itr);
 }

while (reg_h = vpi_scan(reg_itr))
 {
   port_itr = vpi_iterate(vpiPorts, reg_h);

  if (port_itr == NULL)
      continue;

   while ((port_h = vpi_scan(port_itr)) && (k < HANDLE_ARRAY_SIZE))
   {
     vpi_printf("Port name is %s\n", vpi_get_str(vpiName, port_h));
     vpi_printf("Size is %d\n", vpi_get(vpiSize, port_h));
     switch (vpi_get(vpiDirection, port_h))
     {
       case vpiInput:
         vpi_printf("    Direction is input\n");
         break;
       case vpiOutput:
         vpi_printf("    Direction is output\n");
             logsize += vpi_get(vpiSize, port_h);
         handleArray[k] = reg_h;
         ++k;
         break;
       case vpiInout:
         vpi_printf("    Direction is inout\n");
             logsize += vpi_get(vpiSize, port_h);
         handleArray[k] = reg_h;
         ++k;
             break;
     }
   }
   if (k == HANDLE_ARRAY_SIZE)
   {
     vpi_printf("SEU ERROR: Handle array is not large enough");
     vpi_control(vpiFinish, 1);
     return (0);
```

```
    }

    if (port_h)
    {
      vpi_free_object(port_itr);
    }
  }
  if (reg_h)
  {
    vpi_free_object(reg_itr);
  }
vpi_printf("Number of outputs and in/outs: %i\n",k);
vpi_printf("Number of bits per log: %i\n",logsize);

ARRAY_SIZE = (ARRAY_SIZE/logsize) * logsize;

vpi_printf("ARRAY_SIZE: %i\n",ARRAY_SIZE);

for(a=0; a < 5000; a++)
{
        ErrFlags[a] = 0;
        ErrCount[a] = 0;
}

  return(0);
}

/**********************************************************************
 * calltf routine
 *********************************************************************/
extern  PLI_INT32 col_calltf(PLI_BYTE8 *user_data)
{

  vpiHandle systf_h, arg_itr;

  /* get module handle from first system task argument.  Assume the  */
  /* compiletf routine has already verified correct argument type.   */
  systf_h = vpi_handle(vpiSysTfCall, NULL);
  if (systf_h == NULL)
  {
    vpi_printf("SEU ERROR: create_output_log could not obtain handle to systf call\n");
    return(0);
  }

  arg_itr = vpi_iterate(vpiArgument, systf_h);
 if (systf_h == NULL)
 {
  vpi_printf("SEU ERROR: create_output_log could not obtain iterator to systf args\n");
  return(0);
 }

 vpi_free_object(arg_itr);

// Uncomment the section below when creating original file
//********************************************************
  if (build_golden==1)
  {
 if (columnFilePtr == NULL)
   columnFilePtr = fopen(columnLocation, "w");

 if (columnFilePtr == NULL)
 {
   vpi_printf("Could not open column file for writing");
   vpi_control(vpiFinish, 1);
 }

 }
/********************************************************/
// Uncomment the section above when creating original file

 // Comment the section below when creating original file
 // No errors found, it is now safe to open the column file
/*********************************************************/
 if (build_golden==0)
 {
 if (columnFilePtr == NULL){
  vpi_printf ("Reading golden copy from %s\n",columnLocation);
      vpi_printf ("Using COL Version 9\n");
  columnFilePtr = fopen(columnLocation, "r");
 }
 if (columnFilePtr == NULL)
 {
  vpi_printf("SEU ERROR: Could not open column file for reading");
  vpi_control(vpiFinish, 1);
  return (0);
 }
 }
/*****************************************************/
// Comment the section above when creating original file

 k = 0;
```

```
  int temp;
  while (handleArray[k] != NULL)
  {
        //Comment the section below when creating original file
   /***************************************************/
   if (build_golden==0)
   {
          if (j == ARRAY_SIZE)
                j = 0;

          if (j == 0)
          {
//              if (j == ARRAY_SIZE)
//                     vpi_printf("TEMP is: %c", temp);
                temp = fgetc(columnFilePtr);

                while ((temp != EOF) && (i < ARRAY_SIZE))
                {
                 if (temp != 10)
                 {
              valueArray[i] = temp;
//               vpi_printf("Index: %i Value: %c Actual Value:
%i\n",i,valueArray[i],valueArray[i]);
              ++i;
             }
                 temp = fgetc(columnFilePtr);
                }

                i = 0;

                if (ferror(columnFilePtr) != 0)
                {
                vpi_printf("SEU ERROR: Error occurred while reading column file");
                fclose(columnFilePtr);
                vpi_control(vpiFinish, 1);
                return (0);
                }
          }
   }
        /*******************************************************/
        //Comment the section above when creating original file

//  vpi_printf(" %s \n", vpi_get_str(vpiName,handleArray[k]));
   PrintSignalValues(handleArray[k]);
   ++k;
  }
  return(0);

}

void PrintSignalValues(vpiHandle port_handle)
{
  PLI_INT32   signal_type;
  s_vpi_value current_value;
  int size;
  int l = 0;

  s_vpi_time now;

  unsigned long long nowtime;

  now.type = vpiSimTime;

  vpi_get_time(0, &now);

  int timeUnit = vpi_get(vpiTimeUnit,NULL);

  nowtime = now.high;
  nowtime = nowtime << 32;
  nowtime = nowtime | now.low;

//  vpi_printf("  Port name is %s\n", vpi_get_str(vpiName, port_handle));
//  vpi_printf("    Size is %d\n", vpi_get(vpiSize, port_handle));
  size = vpi_get(vpiSize, port_handle);
  current_value.format = vpiBinStrVal;
  vpi_get_value(port_handle, &current_value);

  char* cur_val = current_value.value.str;

// Comment the section below when creating original file
/************************************************************/
  if (build_golden==0)
  {
  for(l=0; l < size; l++){
  if (valueArray[j] != cur_val[l])
  {
  // An error has been detected- compute the simulation
  //  time, print information and abort simulation with '$finish'

   if (ErrFlags[ErrIndex] == 0 && ErrCount[ErrIndex] <= UnErrorCap){
```

```
  vpi_printf("\nSEU ERROR: Output difference at time: %llu e %i\n", nowtime,
timeUnit);

  vpi_printf("SEU ERROR: Original value:   %c", valueArray[j]);
  vpi_printf("\nSEU ERROR: Error line:  ");
  vpi_printf("%s",
            current_value.value.str);
  vpi_printf(" %s \n", vpi_get_str(vpiName,port_handle));
  vpi_printf("SEU ERROR: In Bit Location: %i \n",l);
  ErrFlags[ErrIndex] = 1;
  }
  else if (ErrCount[ErrIndex] == UnErrorCap + 1){
        vpi_printf("SEU ERROR: %s in bit location %i has an error past the recovery
cap!\n", vpi_get_str(vpiName,port_handle), l);
  }
  ErrCount[ErrIndex] += 1;

//   vpi_printf("Error Count is: %i", ErrCount[ErrIndex]);

//   vpi_control(vpiFinish, 1);
  }
  else if (ErrFlags[ErrIndex] == 1) {
        ErrFlags[ErrIndex] = 0;
        vpi_printf("\nSEU CORRECTION: Output difference is corrected at time: %llu e
%i\n", nowtime, timeUnit);
        vpi_printf("SEU CORRECTION: At port: %s in bit location: %i
\n",vpi_get_str(vpiName,port_handle),l);
  }
 ++j;
 ++ErrIndex;
 if (ErrIndex == logsize){
        ErrIndex = 0;
 }
 }
 }
/*****************************************************/
// Comment the section above when creating original file


// Uncomment the section below when creating original file
//****************************************************
if (build_golden==1)
{
 for(l=0; l < size; l++){
 if (fputc(cur_val[l], columnFilePtr) == EOF)
 {
  printf("Unable to write to output file");
  return;
 }
 if (fputc(10, columnFilePtr) == EOF)
 {
  printf("Unable to write to output file");
  return;
 }
 }
}
/*********************************************************/
// Uncomment the section above when creating original file
```

APPENDIX B

SIMULATION ANALYSIS BASH SCRIPTS

Large ASIC Script

```
#!/bin/bash
UPSETS=0
OUTPUT_PATH=/auto/dcbu_hw/DFR-nbidokht/vanderbilt/vanderbiltvcs
SIMS=$1
for ((i=1;i<1+$SIMS;i+=1)); do
        runhwk -build -tn test_fc_basic -vo "+define+HWK_BEH_MEM" -vo "-P" -vo
"/auto/dcbu_hw/DFR-nbidokht/vanderbilt/test/cfiles/vpiSingleEvent/vcs_pli.tab" -vo
"/auto/dcbu_hw/DFR-nbidokht/vanderbilt/test/libsingleEvent.so" -vo
"/auto/dcbu_hw/DFR-nbidokht/vanderbilt/COLBuild/create_output_log4.c" -dd
/auto/dcbu_hw/DFR-nbidokht/vanderbilt > $OUTPUT_PATH/run$i.seq;
        grep -e "Corey" -e "Time is:" $OUTPUT_PATH/run$i.seq >
$OUTPUT_PATH/grun$i.seq
        if [ `grep 'SEU ERROR' $OUTPUT_PATH/run$i.seq | wc -l` -gt 0 ]; then
                echo run$i >> $OUTPUT_PATH/errors.log
                grep 'Generated' $OUTPUT_PATH/run$i.seq >>
$OUTPUT_PATH/errors.log
                grep 'Generated' $OUTPUT_PATH/run$i.seq >>
$OUTPUT_PATH/upset.log
                grep 'SEU ERROR' $OUTPUT_PATH/run$i.seq >>
$OUTPUT_PATH/errors.log
                let UPSETS=$UPSETS+1
        else
                grep 'Generated' $OUTPUT_PATH/run$i.seq >>
$OUTPUT_PATH/noupset.log
        fi
        gzip $OUTPUT_PATH/run$i.seq;
        gzip $OUTPUT_PATH/grun$i.seq;
done

echo $UPSETS " upsets" >> $OUTPUT_PATH/upset.log
echo $UPSETS " upsets" >> $OUTPUT_PATH/errors.log
```

Eight-Bit Microprocessor Script

```
#!/bin/bash
SIMS=$1
UPSETS=0
ERRORS=0
CORRECTIONS=0
TOTERRORS=0
TOTCORRECTIONS=0
UNERRORS=0
INJTIME=0
INJECTIONS=1

for((i=1; i<$SIMS; i+=1)); do
 ./simv > out/run$i.seq;
 if [ `grep 'SEU ERROR' out/run$i.seq | wc -l` -gt 0 ]; then
        echo run$i >> out/errors.seq
        grep 'Generated' out/run$i.seq >> out/errors.seq
        grep 'SEU ERROR\|SEU CORRECTION' out/run$i.seq >> out/errors.seq
        grep 'Generated' out/run$i.seq >> out/upset.seq
        echo run$i >> out/latency.seq
        grep 'Generated' out/run$i.seq >> out/latency.seq
        grep 'SEU ERROR' out/run$i.seq | head -n 4 >> out/latency.seq
        grep 'SEU CORRECTION' out/run$i.seq | tail -2 >> out/latency.seq
        let ERRORS=$(echo `grep 'SEU ERROR' out/run$i.seq | wc -l` / 4 | bc)
        let CORRECTIONS=$(echo `grep 'SEU CORRECTION' out/run$i.seq | wc -l` / 2
| bc)
        echo $ERRORS " errors" >> out/errors.seq
        echo $CORRECTIONS " corrections" >> out/errors.seq
        let UNERRORS=$ERRORS-$CORRECTIONS
        echo $UNERRORS " errors not corrected!!!" >> out/errors.seq
        let TOTERRORS=$TOTERRORS+$ERRORS
        let TOTCORRECTIONS=$TOTCORRECTIONS+$CORRECTIONS
        let UPSETS=$UPSETS+1
        if [ `grep 'corrected' out/run$i.seq | wc -l` -gt 0 ]; then
                echo run$i >> out/corrections.seq
                grep 'SEU CORRECTION' out/run$i.seq >> out/corrections.seq
        fi
 else
        grep 'Generated' out/run$i.seq >> out/noupset.seq
fi

echo $INJECTIONS " injections" " " $UPSETS " upsets" >> out/avf.log
let INJECTIONS=$INJECTIONS+1

rm out/run$i.seq
```

```
done

echo "  " >> out/errors.seq
echo $SIMS " total faults injected" >> out/errors.seq
echo $UPSETS " total faults causing errors" >> out/errors.seq
echo $TOTERRORS " total errors" >> out/errors.seq
echo $TOTCORRECTIONS " total corrections" >> out/errors.seq
```

APPENDIX C

IMPLEMENTATION EXAMPLE

```
        …
initial begin
    $singleEventMaxMemorySize(2024);
    $singleEventInit();
    #($pseudoRandom(2400000))
    $singleEventUpset(hwk_core);
end

always @ (posedge refClk or negedge refClk) begin
        $create_output_log(hwk_core);
end
        …
```

REFERENCES

[1] P.E. Dodd, L. W. Massengill, "Basic mechanisms and modeling of single-event upset in digital microelectronics", *IEEE Transactions on Nuclear Science, vol.50, no.3, pp. 583-602,* June 2003.

[2] J.F. Ziegler, et al., "IBM experiments in soft fails in computer electronics (1978-1994)," *IBM Journal of Research and Development, vol. 40, no. 1, pp. 3-18,* January 1996.

[3] R.C. Baumann, "The impact of technology scaling on soft error rate performance and limits to the efficacy of error correction" , *2002 International Electron Devices Meeting, pp. 329-332,* 2002.

[4] M. Santarini, "Cosmic radiation comes to ASIC and SOC design" , *EDN,* 2005.

[5] R.C. Baumann, "Soft errors in commercial semiconductor technology: overview and scaling trends," *IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals, pp. 121_01.1-121_01.14,* April 7, 2002

[6] H. Ando, et al., "A 1.3 GHz fifth generation SPARC64 microprocessor," *International Solid-State Circuits Conference,* 2003.

[7] T. Calin, M.Nicolaidis, and R. Velazco, "Upset hardened memory design for submicron CMOS technology," *IEEE Transactions on Nuclear Science, vol. 43, no. 6,* December 1996.

[8] E. Rotenberg, "AR-SMT: A microarchitectural approach to fault tolerance in Microprocessors," *Proceedings of Fault-Tolerant Computing Systems (FTCS),* 1999

[9] S.S. Mukherjee, et al., "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," *36th Annual International Symposium on Microarchitecture (MICRO),* December 2003.

[10] P.A. Thaker, V.D. Agrawal, and M.E. Zaghloul, "Register-transfer level fault modeling and test evaluation techniques for VLSI circuits," *International Test Conference, pp. 940-949*, 2000.

[11] S.Kim and A.K. Somani, "Soft error sensitivity characterization for microprocessor dependability enhancement strategy," *Proceedings of the International Conference on Dependable Systems and Networks (DSN),* 2002.

[12] S.S. Mukherjee, et al., "The soft error problem: an architectural perspective," *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA-11 2005), pp. 243-247,* Feb. 2005.

[13] E.D. Lazowkska, et al., "Quantitative System Performance," *Prentice-Hall, Inc., Englewood Cliffs, NJ,* 1984.

[14] David Kammler, et al., “A fast and flexible platform for fault injection and evaluation in verilog-based simulations,” *Third IEEE International Conference on Secure Software Integration and Reliability Improvement (SSIRI), pp. 309-314,* 2009.

[15] J. Arlat, et al., “Fault injection for dependability validation – a methodology and some applications,” *IEEE Transactions on Software Engineering, 16 (2), pp. 166-182,* February 1990.

[16] H. Madeira, et al., “RIFLE: A general purpose pin-level fault injector,” *Proc. EDCC-1, Springer LNCS, Vol. 852, pp. 199-216,* 1994.

[17] U. Gunneflo, et al., “Evaluation of error detection schemes using fault injection by heavy-ion radiation,” *Proc. 19th Ann. Int’l Symp. Fault-Tolerant Computing (FTCS 89), pp. 340-347,* 1989.

[18] G.A. Kanawati, et al., “FERRARI: A tool for the validation of system dependability properties,” *Proc. 22nd Ann. Int’l Symp. Fault-Tolerant Computing (FTCS 92), pp. 336 – 344,* 1992.

[19] S. Han, et al., “DOCTOR: An integrated software fault injection environment for distributed real-time systems,” *Proc. Int’l Computer Performance and Dependability Symp., pp. 204-213,* 1995.

[20] J. Carreira, et al., “Xception: A technique for the experimental evaluation of dependability in modern computers,” *IEEE Trans. On Software Engineering, vol. 24, pp. 125-136,* February 1998.

[21] E. Jenn, et al., “Fault injection into VHDL models: the MEFISTO tool,” *Proc. 24th Int’l Symp. Fault Tolerant-Computing (FTCS 94), pp. 66-75,* 1994.

[22] V. Sieh, et al., “VERIFY: Evaluation of reliability using VHDL-models with embedded fault descriptions,” *Proc. 27th Int’l Symp. Fault-Tolerant Computing (FTCS 97), pp. 32-36,* 1996.

[23] H.R. Zarandi, et al., “Dependability analysis using fault injection tool based on synthesizability of HDL models.” *Proc. 18th IEEE Int’l Symp. Defect and Fault Tolerance in VLSI Systems (DFT 03), pp. 485 – 492,* 2003.

[24] J.C. Baraza, et al., “A prototype of a VHDL-based fault injection tool: description and application,” *J. Systems Architecture, vol. 47, no. 10, pp. 847-867,* 2002.

[25] S. Sutherland, *The Verilog PLI Handbook, Second Edition.* Kluwer Academic Publishers, 2002.