Fast FPGA-Based Fault Injection Tool for Embedded Processors

Mohammad Shokrolah Shirazi, Brendan Morris, Henry Selvaraj
Department of Electrical and Computer Engineering, University of Nevada, Las Vegas
E-mail: shirazi@unlv.nevada.edu, brendan.morris@unlv.edu, henry.selvaraj@unlv.edu

Abstract

FPGA-based fault injection methods have recently become more popular since they provide high speed in fault injection experiments. During each fault injection experiment, FPGA should send data related with observation points back to host computer for fault tolerant analysis. Since there is high data volume, FPGA should spend most of its time in communication. In this paper, we solve this problem by bringing all parts of fault injection tool inside FPGA. The area overhead problem related with observation data is obviated by using simple observation circuit. As case study, we injected 6400 SEU faults into OpensRISC 1200 processor over the Cyclone II FPGA. Results show that our fault injection experiments are done more than 400 times faster than one of the traditional FPGA based fault injection methods with only 5% area overhead.

Keywords

FPGA, Fault injection methods, SEU faults

1. Introduction

In recent years, embedded systems are increasingly being used to protect large investments or human lives. Most embedded systems are the microprocessor-based systems which come with a large number of common characteristics including dependability and real time constraints [1][2]. Fault injection is one important way for evaluating microprocessors and finding dependability parameters. Within numerous fault injection methods that have been proposed, there is four major groups: 1) software implemented fault injection [3-5] 2) physical fault injection [6][7] 3) Simulation-based fault injection [8][9] 4) FPGA-based fault injection [10-18].

Simulation-based fault injection methods are more preferable than physical and software implemented fault injection methods since they provide high controllability and observability. They inject faults into Verilog or VHDL model of the circuits and they can be used in design phase of the system [8-9]. Although Simulation Based fault injection methods have several advantages, they are so time-consuming. As an alternative way, FPGA can accelerate fault injection experiments and it can provide good controllability and observability as well. There are two major groups for FPGA-based fault injection methods [11]: 1) Reconfiguration-based Techniques 2) Instrumentation-based techniques.

In reconfiguration-based techniques [14][19], faults are injected by changing the bit stream needed for configuring FPGA. So, FPGA should get reconfigured for each fault injection experiment and it suffers from time-overhead.

In instrumentation-based techniques [10-13][15-18], extra circuits are added to the original circuits and both are located inside FPGA after getting synthesized. Although these methods do not have time overhead, they suffer from area overhead. In instrumentation-based techniques, FPGA is mostly used as evaluation circuit for accelerating fault injection experiments. So, area overhead is not big deal since speed is more important. Hence, instrumentation based techniques are more preferable than reconfiguration based techniques if we need speed in our experiments.

In [17], one FPGA-based fault injection tool is presented. This tool includes software and hardware parts. The hardware part, named fault injector, is written in VHDL and it is brought inside processor. Then, processor along with the fault injector is located inside FPGA. However, software part located on host computer controls and manages fault injection experiments by sending commands to FPGA by means of host interface. Another instrumentation-based technique is presented in [15][20]. It adds extra circuits to original circuit described by VHDL. These extra circuits, which are named mask chains, are used for bit-flip fault injection and they can be read out sequentially after fault injection. Faulty vectors are sent from host computer to FPGA and observation data are come back to host computer for analyzing results. In [10], another FPGA-based fault injection tool is presented with the similar idea but it is based on Verilog language. This tool includes fault injection manager and result analyzer. Result analyzer is software part located on host computer. Fault injection manager has software and hardware parts which are located on host computer and FPGA respectively. The software part is connected to hardware part through the parallel port. Another tool which is capable of injecting faults into Memory, register file and processor core is presented in [16]. Like former methods, fault injection is performed by sending commands from host computer to FPGA. For analyzing results, it stores observation data of several experiments into internal memory of the FPGA. So, it sends observation data to host computer once instead of sending it for each fault injection experiment. This idea accelerates fault injection experiments by reducing communication between FPGA and host computer.

As we see, all of instrumentation-based techniques suffer from communication bottleneck. This communication between FPGA and host computer plays key role for determining speed for fault injection experiments. After each fault injection experiment, observation data must be sent back to host computer from FPGA for later analysis and unfortunately the data volume is usually so high. In this paper, we omit this communication overhead by bringing all fault injection parts along with processor inside FPGA. Since we bring all parts inside FPGA, we should find way to deal with the new problem. This problem is related with the high volume of observation data that should be gathered for

fault-tolerant analysis. We figure this out by using simple observation circuit for compressing data. In addition, we do not run faulty experiments for whole runtime of the workload. We hopefully assume that fault effects are manifested after several clocks for each fault injection experiment. However, we need one non-faulty experiment (golden run) after each faulty experiment for comparing and extracting fault propagation results. In the rest of our paper, we will talk about our method in more details.

The rest of our paper is organized as follows. Our fault model is presented in section 2. Different part of our fault injection tool is presented in Section 3. Section 4 describes fault injection process and section 5 has experimental results. Finally, section 6 concludes the paper.

2. Fault Model

Trends in CMOS technology, related applications as well as operating conditions cause circuits to be more sensitive to transient faults. Unfortunately, deep sub-micron system on chip and low power design techniques aggravate the reliability problem [21][22]. One major reason for having transient faults is single event up-sets (SEU faults). SEU faults come from ionized particles in atmosphere and they might hit memory elements and flip their contents.

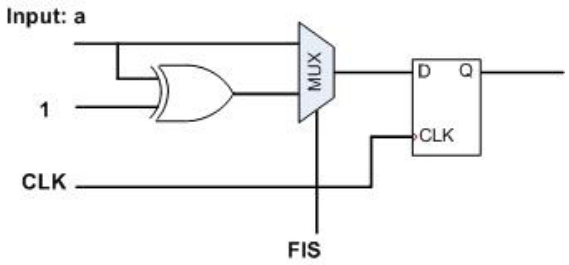

Figure 1: Instrumented circuit for SEU fault injection

In our method, we add extra circuits to original circuit for injecting SEU faults. This modified circuit, that is capable of SEU fault injection, is named instrumented circuit. Our instrumented circuit, which was formerly introduced in [10][15], is shown in figure 1. As it is shown in figure 1, inverted value of input will go into flip-flop while fault injection set (FIS) is high. This instrumented circuit is desired since it doesn't halt processor for each fault injection experiment. This helps us for having fast fault injection experiments that is important for real time applications. However, halting processor is mostly used in scan chain based SEU fault injection methods [11].

3. Fault Injection Architecture

As it is shown in figure 2, our proposed fault injection architecture includes three different modules.

1- **Fault Injection Manager Module:** This is responsible for driving FIS (setting FIS to 1) at fault injection time and triggering observation module.

2- **Observation Module:** After getting triggered by fault injection module, it starts to record data from observation points. This work is done by its observation circuit.

3- **Result Analyzer Module:** This module is triggered after each two experiments by observation module. One experiment is faulty free (golden run) and another is faulty experiment. Result analyzer compares the observed data of these two experiments and reports fault propagation results.

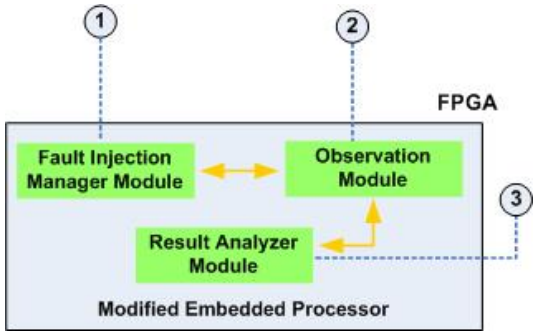

Figure 2: Fault injection architecture

3.1. Fault Injection Manager

Fault injection manager is responsible for managing and executing fault injection experiments. At first, this module sets up fault injection timers at beginning of experiments. It starts to count until it reaches to fault injection time. Secondly, it decides whether raises fault injection set (FIS) to one or not and then it triggers observation module.

In our method, we perform faulty free experiment (golden run) after each faulty experiment. So, we actually perform two experiments for each fault injection experiment. One golden run is performed to have the reference for comparison. A method for running golden run and faulty experiment alternatively is also introduced in [11] but it is done for each clock until the unequal values are manifested. In our method, we perform golden run for several clocks and then we repeat it with same values for faulty experiment. This exempts us from recording all observation data regarding to faulty and non-faulty experiments. It certainly helps us since we have implemented all parts of our fault injection tool in hardware and we always have limitation in hardware resources.

3.2. Observation

Observation module starts to record data from observation points after getting triggered by fault injection manager. The observation circuit which is used in observation module is shown in figure 3.

Since one faulty free experiment is executed for each fault injection experiment, golden flip-flop is used to distinguish them. Our simple adder circuit adds new data from observation points with previous values. It is done from two different paths regarding to golden run and faulty experiment. The reason for using this logic is related with this fact that if observed data becomes different from golden run, it will affect addition result and we will more probably have two different values at the end.

We hopefully assume that injected faults are manifested several clocks after each fault injection. This helps us to

record the data for several clocks instead of the whole runtime of each experiment. It will significantly accelerate the speed of our fault injection experiments.

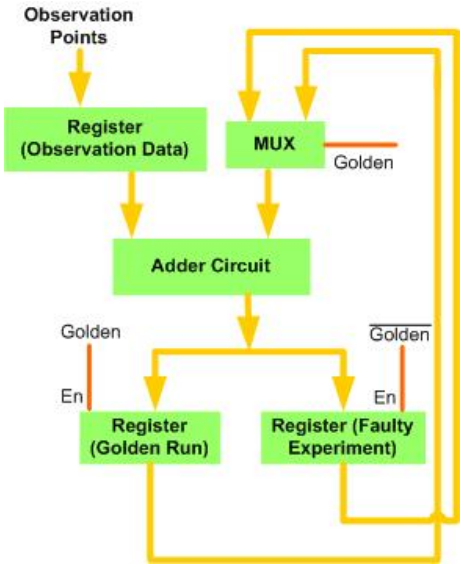

Figure 3: Observation circuit

3.3. Result Analyzer

Result analyzer is triggered by observation module while each faulty experiment in addition to its golden run is executed. It just compares the values of two registers related with golden run and faulty experiment. Then it extracts fault propagation results by means of some counters. It just increments its related counter if it finds inequality between these two values.

4. Fault Injection Process

1. Fault injection manager sets up timers and golden flip-flop. Fault injection manager first checks golden flip-flop. If it is one, it sets new values for timers regarding to fault injection time as well as experiment run time. If golden is zero, timers are loaded with same previous values used in golden run.
2. Fault injection timer starts to down count until it reaches to zero. It was formerly loaded with fault injection time by fault injection manager.
3. After fault injection timer becomes zero, fault injection manager checks the golden flip-flop. If it is zero, fault injection manager makes FIS one. Otherwise FIS will remain zero.
4. Fault Injection manager triggers observation module. Observation module sets observation timer and starts to record observed data by observation circuit which was described in section 3.2. Observation timer determines observation time for recording data from observation points by observation circuit.
5. While observation timer becomes zero, it checks whether golden flip-flop is 1 or 0. If golden is one, observation module clears golden flip-flop. The microprocessor is restarted and it goes to step 1.
6. If golden flip-flop is zero, observation module triggers result analyzer.
7. Result Analyzer compares faulty and golden run results and it increments related counters as well if they are not equal. The microprocessor is restarted and it goes to step 1.

5. Experimental Results

In our experiments, we used OpenRISC 1200 processor [23] [24] as case study for evaluating our fault injection tool. We implemented our fault injection tool by Verilog language. After implementing our fault injection tool, we connected it to OpenRISC 1200 processor. This connection is generally done in two steps. As first step, we make instrumented circuits for each fault injection location and we consider fault injection set (FIS) for each one. Then we make wiring between fault injection manager and instrumented circuits. We made instrumented circuits for different registers inside OpenRISC 1200 processor. These locations have been shown in table 1. As it's shown in table 1, we have totally considered 64 fault injection points which needs 64 fault injection sets (FIS).

Table 1: Fault injection locations & Experiments

Module	Description	#FI Points	#FI Exp
GenPC	Program counter & interface to IC	6	300
IF	Instruction fetch	13	650
Control	Control & instruction decoding unit	15	750
Op-Mux	Mux for two register file read operands	20	1000
Wb-Mux	CPU's write-back stage of the pipeline	10	500

As second step, we should determine observation points and wire them to observation circuit. Since we classify fault propagation results as control flow error, data error and failure, we considered address bus and data bus as observation points. The address bus of instruction memory is one of our observation points used for control flow error. For data error, we considered data bus of data memory as well as output register of multiplexer for register file. For failure, we only considered data bus of data memory as observation point. After doing these two steps, we just defined one input for starting experiments. Our fault injection tool is triggered with this input and starts to automatically inject SEU faults into different modules of processor. After finishing experiments number of control flow errors, data errors and failures is shown.

We developed our fault injection tool by using Altera DE2 board [25], equipped with cyclone II EP2C35F672C6 FPGA. One major aspect to consider is increasing in logical elements that are introduced with our fault injection hardware. After doing synthesis for a FPGA cyclone II EP2C35F672C6, we estimated FPGA overhead based on logic elements. The synthesis result which was carried out with Quartus II 9.1 Web Edition is shown in table 2. Results

show that our fault injection tool has 5% overhead based on logic elements.

Table 2: Available & consumed FPGA resources (EP2C35F672C6)

	Total Logic Elements	Total Registers
Cyclone II FPGA	33216	33216
OpenRISC 1200	5626/33216(17%)	2175/33216(7%)
OpenRISC 1200 + Built-in FI Tool	7407/33216(22%)	2418/33216(7%)

In our experiments, matrix multiplication and bubble sort are considered as workload programs [10][24]. These workloads are loaded in instruction memory after connecting instruction memory and data memory to microprocessor.

SEU faults are injected in different parts of CPU modules as it is described in table 1. Table 1 also shows the number of fault injection experiments for each module. As it is shown in table 1, for each fault injection location (FIS), experiments were carried out 50 times with different fault injection time. Our fault injection times were distributed equally with distance of 20 clocks. For matrix multiplication, we considered 300 clocks for observation time. Since runtime for workload execution of bubble sort is longer, we considered 1200 clocks for observation time. The fault duration was one clock and FIS signal is triggered with negative edge of clock. The OpenRISC 1200 microprocessor was run under 50 MHZ clock over the FPGA.

We compared the speed of our fast FPGA-based fault injection tool with one of the traditional FPGA-based fault injection tool described in [10].

As it is shown in table 3, we reached to speed-up more than 450 in our experiments. However, our speed is tightly related with observation time. If we decrease observation time, we will have more speed but we might not give enough time to faults for getting manifested. So, it proves that there is always trade-off between speed and obsevability.

Table 4 shows fault propagation results. Results show that for matrix multiplication GenPC and Operand Mux are most sensitive parts against SEU faults since they have more percentage of failure. For bubble sort, GenPC and control units are most sensitive parts. As we expected, injected faults into GenPC more contributed in control flow error since this module holds program counter. Because of the observation points that we considered, we see the similar results between data error and failure. AS we expected, control unit and operand mux are also sensitive against SEU faults. Control unit is responsible for making control signals for pipeline and operand mux is used so much during workload execution. At second stage of pipeline, operand mux chooses operands or immediate address based on instruction set address. So, fault injection into this module manifest errors at fourth or fifths stage of pipeline for store or load instructions. So, this module is important since it is used in each workload instruction.

Table 3: Resulted speed-ups

Workload	Traditional FI Time	Fast FI Time	Speed-up
Matrix Multiplication	81	0.18	450
Bubble Sort	318	0.52	611

6. Conclusion

In this paper, we presented fast fault injection tool which is based on FPGA. It was done by completely cutting communication time and implementing all fault injection parts in hardware. As we compared our results with pervious related work [10], we find out that our tool is fast enough as it was expected but there are some issues that should be addressed. This tool like other FPGA-based fault injection tools has limitation for controllability and observability in comparison with simulation-based fault injection methods. Since fault injection time is based on clocks, we cannot inject faults between clock edges. Another problem is related with observation time. If we make it longer, we will more probably reach to more accurate results.

Table 4: Fault Propagation Results

Workload	Module	%Error Manifest		% Failure
		% CFE	% DE	
Matrix Multiplication	WB Mux	0	11	11
	Operand Mux	2	46	46
	Ctrl Unit	8	28	27
	IF	1	15	13
	GenPC	83	83	83
Bubble Sort	WB Mux	10	14	8
	Operand Mux	11	23	19
	Ctrl Unit	21	28	25
	IF	2	5	1
	GenPC	28	27	24

7. References

[1] P. Marwedel, Embedded Systems Design, Springer, 2006.

[2] A. S. Berger, Embedded Systems Design - An Introduction to Processes, Tools, & Techniques, CMP Books Publisher, 2002.

[3] Z. Segall, and T. Lin, "FIAT: Fault Injection Based Automated Testing Environment," Proceeding of the 18th International Symposium on Fault-Tolerant Computing, Jun. 1988, pp. 102-107.

[4] G. A. Kanawati, N. A. Kanawati, and J. Abraham, "FERRARI: A Tool for the Validation of System Dependability Properties," Proceeding of the 22th International Symposium on Fault-Tolerant Computing, Jul. 1992, pp. 336-344.

[5] Carreira J. H. Madeira, and J. G. Silva, "Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers," IEEE Transaction on Software Engineering, Feb. 1998, pp. 125-136.

[6] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J. C. Fabre, J. C. Laprie, E. Martines, and D. Powell, "Fault Injection for Dependability Validation - A Methodology and Some Applications," IEEE Transaction on Software Engineering, Feb. 1990, pp. 166-182.

[7] H. Madeira, M. Z. Rela, F. Moreira, and J. G. Silva, "RIFLE: A General Purpose Pin-level Fault Injector," Proceeding of the First European Dependable Computing Conference, 1994, pp. 199-216.

[8] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson, "Fault Injection into VHDL Models: The MEFISTO Tool," Proceeding of the 24th International Symposium on Fault-Tolerant Computing, Jun. 1994, pp. 66-75.

[9] H. R. Zarandi, S. G. Miremadi, and A. R. Ejlali, "Fault Injection into Verilog Models for Dependability Evaluation of Digital Systems," Proceeding of the Second International Symposium on Parallel and Distributed Computing, Oct. 2003, pp. 281-287.

[10] M. Shokrolah-Shirazi and S. G. Miremadi, "FPGA-based Fault Injection into Synthesizable Verilog HDL Models," Proceeding of the Second International Conference on Secure System Integration and Reliability Improvement, Jul. 2008, pp. 143-149.

[11] A. R. Ejlali and S. G. Miremadi, "Error propagation analysis using FPGA-based SEU fault injection," Proceeding of the Microelectronic Reliability, Feb. 2008, 319-328

[12] K.-T. Cheng, S.-Y. Huang and W.-J. Dai, "Fault Emulation: A New Methodology for Fault Grading," IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems, Oct. 1999, pp. 1487-1495.

[13] S.-A. Hwang, J.-H. Hong, C.-W. Wu, "Sequential Circuit Fault Simulation Using Logic Emulation," IEEE Transaction on the Computer-Aided Design of Integrated Circuits and Systems, Aug. 1998, pp. 724-736.

[14] L. Antoni, R. Leveugle and B. Feher, "Using Run-Time Reconfiguration for Fault Injection in Hardware Prototypes," Proceeding of the 17th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2002, pp. 245-253.

[15] P. Civera, L. Macchiarulo, M. Rebadengo, M. S. Reorda, M. and A. Violante, "Exploiting FPGA for accelerating fault injection experiments," Proceeding of the 7th International On-Line Testing Workshop, 2001, pp. 9-13.

[16] P. Civera, L. Macchiarulo, M. Rebadengo, M. S. Reorda, M. and A. Violante, "FPGA-based Fault Injection for Microprocessor Systems," Proceeding of the 10th Asian Test Symposium, Nov. 2001, pp. 304-309.

[17] A. Steininger B. Rahbaran, and T. Handl, "Built-in fault injectors the logical continuation of bist?," Proceeding of the intelligent Solutions in Embedded Systems workshop, 2003.

[18] B. Rahbaran, A. Steininger, and T. Handl, "Built-in Fault Injection Hardware – The FIDYCO Example," Proceeding of the second IEEE International Workshop on Electronic Design, Test and applications, Jan. 2004, pp. 327-332.

[19] M. Aguirre, J. N. Tombs, F. Muñoz, V. Baena, A. Torralba, A. Fernández-León, F. Tortosa-López, "FT-UNSHADES: A new system for SEU Injection, analysis and diagnostics over post synthesis netlists," NASA Military and Aerospace Programmable Logic Devices (MAPLD 2005), Washington DC (USA), 2005.

[20] P. Civera, L. Macchiarulo, M. Rebadengo, M. S. Reorda, M. and A. Violante, "Exploiting Circuit Emulation for Fast Hardness Evaluation," IEEE Transaction on Nuclear Science, Dec 2001, pp. 2210-2216

[21] A. Maheshwari, W. Burleson, R. Tessier, "Trading off transient faults tolerance and power consumption in deep sub-micron (DSM) VLSI circuits," IEEE Transaction on Very Large Scale Integration (VLSI) Systems, Mar. 2004, pp. 299-311.

[22] N. R. Shanbhag, "Reliable and efficient system on chip design," Proceeding of the Computer & Processing, Mar. 2004, pp. 42-50.

[23] "OR1200 OpenRISC processor," available at: http://opencores.org/or1k/Main_Page, [accessed: Jul. 2012].

[24] N. Mehdizadeh, M. Shokrolah-Shirazi and S. G. Miremadi, "Analyzing fault effects in the 32-bit OpenRISC 1200 microprocessor," Proceeding of the Third International Conference on Availability, Reliability and Security, Mar 2008, pp. 648-652.

[25] "DE2 Development and Education Board ," available at:http://www.altera.com/education/univ/materials/boards/de2/unv-de2-board.html, [accessed: Jul. 2012].