Simulated Fault Injection Using Simulator Modification Technique

Jongwhoa Na and Dongwoo Lee

In the current very deep submicron technology era, fault tolerant mechanisms perform an essential function to cope with the effects of soft errors. To evaluate the effectiveness of the fault tolerant mechanism, reliability engineers use simulated fault injections using either saboteur modules or mutants in the simulation model. However, the two methods suffer from both inefficiency in the simulation mechanism and difficulties with the experimental setups. To overcome these inefficiencies, we propose the Verilog-based simulated fault injection (VFI) technique. VFI has the following advantages. First, modification of the design model is unnecessary. Second, the fault injection simulation procedure is simple and efficient. Third, various types of fault injection experiments can be performed. To evaluate the effectiveness of the proposed methodology, we developed a VFI environment using the ICARUS Verilog Simulator. From the experimental results, we were able to qualitatively evaluate the reliability of the target simulation models and to assess the effectiveness of the employed fault-tolerance mechanisms.

Keywords: Simulated fault injection, reliability, Verilog simulation, system-on-a-chip.

I. Introduction

In the very deep submicron technology era, strategies to cope with the effects of soft errors are emerging as an important design issue [1]. One solution to cope with soft errors is to design the dependable target by incorporating various types of fault-tolerant mechanisms and evaluating the effectiveness of the incorporated mechanisms. For the analysis and evaluation of the effectiveness of the fault-tolerant mechanism of modern ASICs or SoCs, fault injection techniques are the most popular [2]-[5].

We may categorize fault injection techniques into four types according to the types of the fault injection targets: i) hardware fault injection, ii) software fault injection, iii) simulated fault injection, and iv) hybrid fault injection [4]. Brief explanations of these techniques will be presented in the next section.

Among these fault injection techniques, the register transfer level (RTL) simulated fault injection techniques have been the most popular. The reasons for this popularity follow. First, they provide more accurate results than abstract simulation methodologies and cost less than physical fault injection. Second, RTL simulation models can be reused at the production stages of final products [6], [7]. Third, they lower the overall development cost by providing the failure rate or reliability data at the design stages. Using the reliability data at the design stage, we can decrease the overall expenses and increase the resilience of the SoCs or embedded systems.

Among existing RTL simulated fault injection techniques, the saboteur method and mutation method are the most popular. However, these methods suffer from both inefficiency in the simulation mechanism and difficulties with the experimental setups. In the case of fault injection using the saboteur method, we must design the saboteur components additionally and

Manuscript received Feb. 24, 2010; revised Apr. 14, 2010; accepted May 3, 2010.

This work was supported by the National Research Foundation of Korea grant funded by the Korea Government (MEST) (No. 2010-0029859).

Jongwhoa Na (phone: +82 2 300 0410, email: jwna@kau.ac.kr) and Dongwoo Lee (email: dongwoo81@kau.ac.kr) are with the Department of Electronics Engineering, Korea Aerospace University, Seoul, Rep. of Korea.

doi:10.4218/etrij.11.0110.0106

redesign the original model to integrate the saboteur components in the original design model which give rise to the numerous interconnections between the saboteur modules and the original design models. This may incur significant overhead since the complexity of the design model is enormous in this 32 nm era [5]. Furthermore, with the addition of the saboteur modules and the corresponding interconnections, although the amount might be small, we are modifying the electrical characteristics of the original target.

In the case of the mutant techniques, the preparation of a mutant simulation model for each fault injection experiment is required. Considering that the required number of fault injection experiments is huge, ranging from 10,000 times or 1,000,000 times depending upon the complexity of the target system model, the mutation method may consume a very long simulation time as well as simulation resources [5].

To overcome these inefficiencies, we are proposing a Verilog-based simulated fault injection (VFI) technique. The major difference between traditional techniques and VFI exists in the simulation kernel. VFI augments the Verilog simulation kernel such that it can perform both the general Verilog simulation and the simulated fault injection experiment, while the previous methods use the original Verilog simulator. During general simulation tasks, VFI performs the normal Verilog simulation on the design model. However, during the fault injection experiments, the user loads the fault attributes and commands for the fault injection experiments so that VFI can inject the fault model into the Verilog design model when the injection conditions are met. The fault attributes include fault injection time, fault location, fault type, and fault model. These attributes can be created by the user during the preparation phase of the fault injection experiment. The advantages of VFI are the following: i) the modification of the design model is unnecessary, ii) the fault injection simulation procedure is simple and efficient, and iii) various types of experiments can be performed.

To evaluate the effectiveness of the proposed environment, we created a VFI environment using the ICARUS Verilog simulator which can compile and synthesize the Verilog model of the IEEE 1364-2005 standard [8]. VFI is a Verilog simulator which is modified such that it can respond to both normal Verilog simulation commands and fault injection commands. After fault injection experiments, VFI performs a failure rate analysis using a diagnostic module to classify the final status of the target model.

To test the functionality of the fault injection environment and the effectiveness of employed fault-tolerance functions, we built a processor model and a fault tolerant processor model. For the processor model, we developed an ARM7 compatible processor in Verilog. It was designed to support the limited but essential instructions for the ARM compiler. For the fault tolerant processor model, we built a triple modular redundant (TMR) processor model based upon the Verilog ARM7 compatible processor. Using the single processor model and the TMR processor model, we performed random fault injection using four fault models: transient stuck-at-0/1 and permanent stuck-at-0/1. From the analysis of the experimental results, we were able to qualitatively evaluate the resilience of the TMR architecture over the single processor model.

The rest of the paper is organized as follows. In section II, the Verilog kernel-based simulated fault injection environment is explained. In section III, the ICARUS simulated fault injection environment is described. In section IV, we describe fault injection experiments using the target simulation model and explain the results. Section V concludes this paper.

II. Overview of Simulated Fault Injection

Simulated fault injection has been widely used for its simplicity, versatility, and controllability. Here, we only review the various types of simulated fault injections. For other fault injection techniques, such as hardware fault injection and software fault injection techniques, please refer to [3]. Simulated fault injection techniques can be classified according to the employed fault injection mechanisms. Depending on the method to inject faults, we may categorize a simulated fault injection as a simulator command technique, simulation model modification technique, or simulator modification injection.

1. Simulator Command Technique

In the simulator command technique, the user injects faults using the built-in commands of the simulator. The faults modify the values of the signals and variables of the target simulation model. As an illustration, the pseudo-commands such as '*simulate-until* (injection instant)' or '*modify_signal* (signal name, fault model)' are built for fault injection experiments. This method is simple because it does not require modification of the simulation model. However, it has limits in its capability to represent various fault models, such as the permanent variable. Also, it is inconvenient to perform the enormous number of fault injection simulation runs required to obtain acceptable simulation results [5].

2. Simulation Model Modification Technique

In simulation model modification, the saboteur technique

and the mutant technique are the most widely adopted methods. In the saboteur technique, we have to build saboteur modules which are used to inject various types of faults into the target model and to control the simulation. We have to connect the saboteur module with the signals and I/O ports of the target modules; thus, we need to modify the original target model. More fault models can be implemented in this technique than in the simulator command technique. However, this technique causes the addition of a large number of control signals into the original target model. This increases the complexity of the simulation model since it requires numerous interconnects between the saboteurs and the I/O ports of the original modules of the simulation model [5].

In the mutant technique, mutants are used to replace a module of the target simulation model. The mutants are the modified or corrupted components representing the fault injection. The mutant technique is versatile and expressive so that various types of fault models can be implemented. However, the problem with the mutant technique is that the mutated simulation models for the experiments have to be prepared in advance. Because any nontrivial fault injection experiment requires a large number of simulation experiments, many mutated simulation models have to be created, which occupy a large amount of storage area.

3. Simulator Modification Technique

The simulator modification technique modifies not the target simulation model but the HDL simulation kernels. In this technique, the modification of the simulation kernel must not alter the original semantics of the hardware. We can achieve this objective by implementing the fault injection services using the event-driven simulation engine without altering the HDL compiler of the simulator.

In this technique, we do not have to modify the complex simulation target models, so the fault injection experiment is simpler and more convenient than the saboteur or mutation techniques. In addition, this technique requires less simulation resources because, by making use of a simple graphic user interference, it can easily perform random and deterministic fault injection experiments in the experimental setup. Also, the simulation can be processed using distributed computing resources for faster reliability analysis.

We can apply this technique into any RTL or electronic systems level (ESL) simulators. In the ESL case, a simulator modification technique using the SystemC simulation kernel has been reported and tested with the single MIPS processor and the TMR MIPS processor models [9]. In the RTL case, we are presenting a fault injection environment using the simulator modification technique with a Verilog simulator.

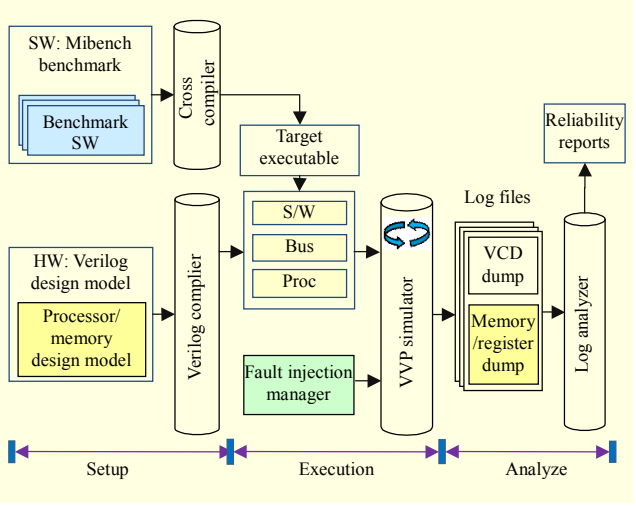

Fig. 1. Three stages of VFI environment.

III. Verilog Kernel-Based Fault Injection Environment

1. Verilog-Based Fault Injection Environment

To analyze the effects of faults and evaluate the dependability of the target system, we built a VFI as shown in Fig. 1. The VFI consists of the target co-simulation model, the Verilog simulator (which is ready for the fault injection) and the analyzer. The target co-simulation model consists of the Verilog hardware model and benchmark software from the Mibench. To evaluate the effectiveness of various kinds of the failure detection and correction modules in the fault-tolerant computing literature, we developed an ARM7 processor model and a TMR model using the base ARM7 processor. Due to limited resources, we designed the base ARM7 processor to support a limited instruction set of around 24 major instructions [10].

The VFI is the ICARUS Verilog simulator augmented with fault injection capability. We analyzed and modified the ICARUS simulation kernel called Verilog virtual processor (VVP) such that it simulates the original Verilog design model and also processes the fault injection process [11]. A detailed explanation of the supported fault model and the modification of the VVP are presented in the next subsection.

The diagnostic unit reads the simulation log results and performs the failure analysis. The simulation log files are a value change dump (VCD) file and a memory dump file. The analyzer decides whether the simulation results belong to one of the following states: i) masked state, ii) silent data corruption (SDC) state, or iii) detected unrecoverable error (DUE) state. A detailed explanation of these will be presented in next section.

Now we explain how fault injection is performed in the VFI environment using the co-simulation model, the modified

ICARUS VVP simulator, and the diagnostic unit. VFI performs the fault injection simulation through 3 steps: set up, execution, and evaluation.

A. Setup

In this stage, VFI prepares the target Verilog design models and the executable files for the target. To compare the effectiveness of the fault tolerant processor, we prepared a base ARM7 processor model and a TMR ARM7 processor model. These two models synthesize through the ICARUS Verilog simulator [11]. Currently, we are preparing fault tolerant processors using various types of failure detection and recovery mechanisms as can be seen in IBM PowerPC or Fujitsu UltraSPARC [12], [13]. For the target software, we selected the well-known embedded benchmark software from Mibench [14]. The selected files are cross-compiled using the Arm Developer Suite from ARM [15]. The compiled binary file is loaded into the program memory model, which is connected to the processor model.

B. Execution

We execute the co-simulation using the hardware simulation model, software simulation model, and the Verilog simulator designed for the fault injection experiments. A user can set the fault attributes of Table 1 and the parameters related to the fault injection experiments, such as the number of simulations and the type of application software. VFI supports the following five fault attributes.

First, the fault type is permanent, transient, or intermittent. An intermittent fault is a repeating fault with a fixed or randomly selected period. It terminates when the preset conditions are satisfied, such as the number of recurrences or the termination time.

Second, the fault model refers to the physical nature of the faults. Our simulation environment supports the fault models including stuck-at-0/1, stuck-at-multi-bit-0/1, stuck-at X/Z, bit flip, open, short, and bridge.

Third, fault time is the time the fault is injected to the hardware model. When the time event occurs and the fault injection conditions are met, it becomes the fault time, and the fault model is injected into the fault location.

Fourth, the fault location is the location of the fault injection in the hardware design model. Because the model is designed in Verilog HDL, the ports, wires, and regs are candidate objects.

Fifth, the fault interval is used with the intermittent fault type, and it controls the fault time of the subsequent fault model, which can be periodic or aperiodic.

For VFI simulation, the user can select the values of the fault attributes such as deterministic injection or random injection using the GUI of the fault injection manager. In deterministic injection, a fault is injected according to the selected injection schedule. On the other hand, in random injection, a fault is injected at a random time and a random location. After each fault injection simulation, the resulting VCD file is output. Through the analysis of each VCD file, the diagnostic unit can analyze the effect of the injected fault.

Table 1. Fault attributes for fault model of Verilog simulation fault injection.

Fault attribute	Value
Fault type	Transient, permanent, or intermittent
Fault time	Random or deterministic
Fault location	Random or deterministic
Fault model	Stuck-at-0 (1), Stuck-at-multi-bit-0 (1), Stuck-at X, Z (for logic type only), Bit flip, open, short, bridge
Fault interval	Periodic or aperiodic (intermittent fault type only)

C. Analysis

The analyzer compares the simulation results from the golden run to those of each fault injection simulation run. The simulation results are the VCD files from step 2 and the memory dump files. Using these two files, the analyzer can decide the state of the fault injection simulation. The output of the analyzer should be in one of the following states: the masked state, recovered state, or failure state. Using these results, we can calculate the fault coverage, failure rate, and reliability of the target design.

2. Verilog Simulator

As a Verilog simulator, we chose ICARUS, which is an open source Verilog simulator. It consists of two stages: the compile stage using iVerilog and the simulation stage using the VVP simulator [11], [16], [17]. In the first stage, the iVerilog compiles the Verilog design file and builds a synthetic file which is sent to the VVP simulator. In ICARUS, iVerilog transforms the Verilog design file into VVP assembly codes.

In the transformation process, the *always* block of the Verilog code is transformed into one or more threads. The sensitivity list of the *always* block is registered as the event list for each thread. The behavioral descriptions in the always block are transformed into internal micro-instructions, such as load, store, add, sub, and mov.

In the second stage, the VVP simulator evaluates the micro-instructions of the VVP assembly code and updates the changes of the corresponding variable in the internal data

structure.

The procedures of the evaluation phase are described in algorithm 1. When the VVP assembly codes are available, the VVP simulator evaluates the micro-instructions of the threads of the VVP assembly codes. For each thread, the VVP simulator executes the *schedule_simulate* function which invokes the *vthread_run* function. The *vthread_run* function executes the micro-instructions of the thread indicated by the program counter (PC) and invokes the *OPCODE_Function_Set* for each and every micro-instruction in the thread. The *OPCODE_Function_Set* executes the appropriate operations required for a given micro-instruction. This invocation repeats until there are no more threads to be executed.

After the evaluation of the micro-instructions, the VVP simulator updates the simulation process by recording the changes of the previous phase into the internal data structures and the VCD file. Note that the VVP simulator maintains the event, net, and port objects in tables, namely, the event table, net vector table, and port table, respectively. After the end of the update phase, it proceeds to the next evaluation phase.

3. Verilog Kernel-Based Fault Injection

To implement a Verilog simulation kernel-based fault injection environment, we modified the VVP simulator of ICARUS to perform kernel-based simulated fault injection without altering the original functionality of the Verilog simulator. As a result, we developed VFI. The operation of VFI is presented in algorithm 2.

In VFI, the original VVP simulation kernel is augmented with a fault injection condition checking process in the evaluation phase and an overwriting process in the update phase. During the evaluation phase, the VFI checks the fault injection time. If the current simulation time is between the *Start_Inject_Time* and the *End_Inject_Time*, it means that it is the time to inject a fault. When this checking process for the fault injection time is over, VFI continues and finishes the evaluation phase and enters into the update phase.

In the update phase, VFI calls the *signal_get_value* function to obtain the value of the target variables, which is the fault location. If the fault injection time is found, then VFI checks for the fault location. It can be determined either randomly or deterministically, depending upon the fault injection campaign profile given by the user. If VFI finds the fault location, VFI overwrites the value of the target variable with the fault model. Next, we explain how VFI operates on the different fault types.

If the fault type is transient, VFI performs fault injection only once. However, if the fault type is permanent or intermittent, VFI must prepare for the subsequent fault injection processes. If the fault type is permanent, VFI reinitializes the fault model and fault time for the next fault injection. This reinitialization process repeats until the end of the simulation. Next, if the fault is intermittent, VFI recalculates the fault time of the next fault model by adding the fault time with the fault interval. If the fault type is periodic, the preset fault interval can be used. However, if it is aperiodic, the fault interval is scaled with the proper probability distribution function or random number generator.

Algorithm 1. VVP simulation process in ICARUS

```
1:  While (! Simulation_finished){
2:    schedule_simulate ( ){
3:      ...   vthread_run (vthread*)   ...
4:      }
5:    }
6:  vthread_run (vthread* thread){
7:   read PC                          /*PC=program counter*/
8:   read vvp_assembly_code(PC)
9:    pc ← pc+1
10:   while (EoF){
11:   ...Call OPCODE_Function_Set(vvp_assembly_code) ...
12:   }
13:  OPCODE_Function_Set {
14:      Case(instruction){
15:       load (vthread*, vvp code structure)
16:           execute load micro-instructions; …
17:       store (vthread*, vvp code structure)
18:           execute store micro-instructions; …
19:       add (vthread*, vvp code structure)
20:           execute add micro-instructions; …
21:       sub (vthread*, vvp code structure)
22:           execute sub micro-instructions; …     }
23:  }
```

Algorithm 2. Fault injection mechanism in VFI

```
1:  init variable
2:   fault_location = Random or Deterministic
3:   fault_duration_time = Start_Inject_Time or End_Inject_Time
4:   fault_type = Transient, Permanent, or Intermittent
5:   fault_model = 1-bit/Multi-bit Stuck-at-0/1
6:   fault_injection_time_ready = False        // initial value
7:  schedule_simulate {                        // evaluation phase
8:   if ((simulation_time>=Start_Inject_Time)//fault injection time
        && (simulation_time <= End_Inject_Time))
9:     fault_injection_time_ready = true }
10:  …                                         // update phase
11:  find target_variable using signal_get_value ( ) //fault location
12:   if [(fault_injection_time_ready == true) and
        (target_variable is found)] {
13:  fault_data←target_variable&fault_model //inject fault model
14:   if (fault_type == Permanent)                  // fault type
15:      then force target_variable with fault_data
16:   register the fault_data in the vvp_vector_table //record
      changes
17:   }
18:   update VCD
```

IV. Fault Injection Experiments

1. Experimental Setups

A. Two Target Verilog Design Models

We developed two ARM-based target processor models to demonstrate the effectiveness of the proposed VFI environment. The base ARM processor model is the ARM7 instruction-set-compatible Verilog model which supports the limited but essential instruction set of 24 instructions.

The base ARM pipeline architecture has four stages: fetch, decode, execute, and write back. To utilize the IPs from the Opencores, we implemented a wishbone bus interface to connect the data/instruction memories as shown in Fig. 2. To compare the dependability measure of the base ARM processor with the more dependable processor, we designed a TMR ARM as shown in Fig. 3. The TMR ARM is the fault tolerant processor with the most straightforward fault protection mechanism. It is highly dependable but expensive. The architecture of the TMR ARM Verilog processor model consists of three ARM processors and a voter. The voter collects the three inputs from the three base ARM processors. If any single output is different from the others, the corresponding processor can be identified as a faulty processor so that any single failure in a processor can be overcome.

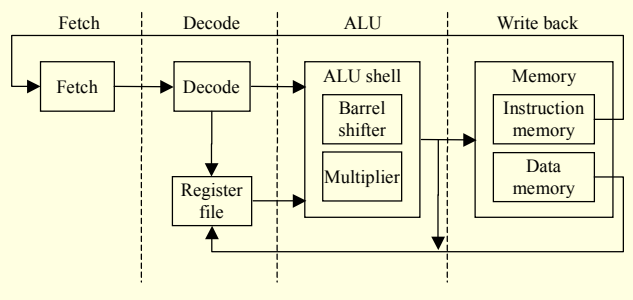

Fig. 2. Architecture of minimized ARM7-compatible processor.

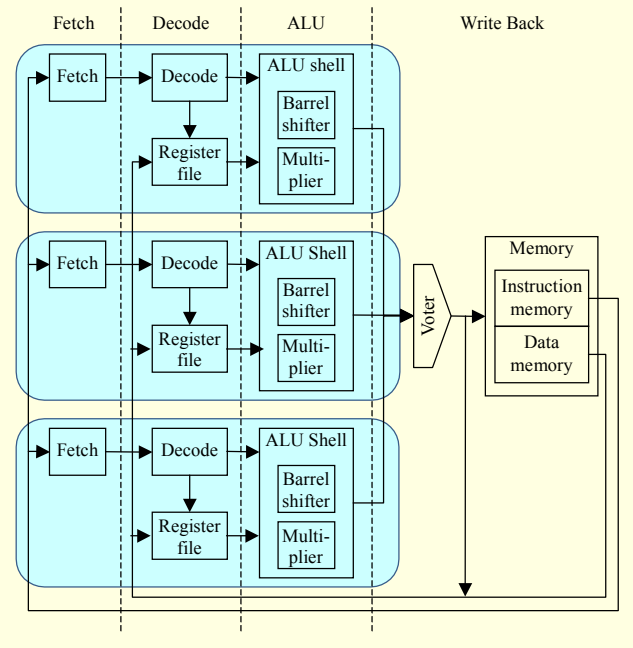

Fig. 3. Architecture of TMR ARM processor.

B. Benchmark Software Models and Fault Models

For the software model, we used the GSM encoder, CRC 32, and Bitcount codes from MiBench [14]. The benchmark codes were compiled using the SDK to obtain ARM executable files. For the fault models, we injected transient and permanent stuck-at-0/stuck-at-1 fault models, as shown in Table 2. Since we are interested in the estimation of the reliability of the two target architectures, we injected the faults at randomly selected times and locations.

Table 2. Fault injection campaign summary. Software models are from telecommunications, automotive, and industrial benchmarks in MiBench.

Hardware model	Software model	Fault model
ARM processor	GSM	Transient/permanent stuck-at-0/1
	CRC32	Transient/permanent stuck-at-0/1
	Bitcount	Transient/permanent stuck-at-0/1
TMR ARM processor	GSM	Transient/permanent stuck-at-0/1
	CRC32	Transient/permanent stuck-at-0/1
	Bitcount	Transient/permanent stuck-at-0/1

C. Number of Fault Injection Experiments Required

To measure the dependability of the target processor model, we can calculate the architectural vulnerability factor (AVF) of the target model in two ways. First, we can calculate the architecturally correct execution (ACE) bits and unACE bits of the target to estimate the AVF of the target model. Since the decision about ACE or unACE depends on the software, the number of test points is enormous. This technique is precise but the complexity of the underlying tasks makes it impractical for the qualitative dependability analysis of modern processors or system-on-a-chip.

The second technique is the dependability analysis using statistical fault injection. Here, the experimenter injects random fault models into the target co-simulation model and performs statistical analysis on the corrupted simulation results. This method is accessible for the dependability measure but the result is only valid within a specified confidence level, such as 95% or 99%. To obtain reliable simulation results, we must

calculate the required number of injection experiments for a fault model n as

$$n = \frac{4z_{\alpha/2}^2 \times \hat{p}(1-\hat{p})}{w^2}, \quad (1)$$

where $z_{\alpha/2}$ represents a value from the table of normal distribution function for a confidence level of $100 \times (1-\alpha)\%$; w is the width of the confidence interval at that particular confidence level; and $\hat{p}$ represents the AVF, which is an estimator of the probability of success in a Bernoulli trial [18].

First, we set the confidence levels as 90%, 95%, and 99%. Since the AVF is not initially known for any target model, we fixed n to a proper constant, say 500 or 1000, to find the first estimate of the AVF of the given target. After the AVF of the target model was known, we recalculated the required number of experiments, n, with the known AVF and the confidence level by using (1). Table 3 summarizes the required number of experiments for the given Verilog target processor models, fault types, fault models, architecturally vulnerability factors, and the three confidence intervals.

Table 3. Required number of experiments for given Verilog target processor models: fault types, fault models, AVF, and three confidence intervals.

Target model	Fault type	Fault model	AVF (%)	Number of experiments required (n)		
				90%	95%	99%
ARM	Permanent	Stuck-at-0	69.6	119	168	291
		Stuck-at-1	52.0	251	355	615
	Transient	Stuck-at-0	0.8	33,759	47,636	82,539
		Stuck-at-1	8.4	2,969	4,189	7,259
TMR ARM	Permanent	Stuck-at-0	22.0	965	1,362	2,359
		Stuck-at-1	15.6	1,473	2,078	3,601
	Transient	Stuck-at-0	0.6	45,103	63,643	110,274
		Stuck-at-1	1.4	19,174	27,056	46,880

2. Analysis of Fault Injection Experiments

A. Fault Injection Simulation Results

After the fault injection simulation is carried out, VFI produces the VCD files in which it records every update of all signals of the Verilog objects. Using the VCD files, we can make decisions about the failure process of the target co-simulation model.

In the simulation, we need a reference to diagnose the failure of the fault injection experiments. Therefore, we build a golden run VCD file and memory dump file, which are built by performing the co-simulation without any fault injection. Once the golden run reference files are available, we can start the fault injection experiments for each fault model to obtain the VCD file and the memory dump file. Then, by comparing these two files to the two files from the golden run, we can diagnose the result of the fault injection experiment.

B. Failure Rate Analysis

The failure state transition diagram and its interaction with the process state transition diagram are illustrated in Fig. 4. In the figure, we used the notations for the naming of the failure states presented in [18] and [19]. The failure state of the target is categorized into one of the following four categories: i) SDC, ii) DUE, iii) benign fault, and iv) SDC non-effect errors.

SDC implies that the target is in failure state. If the memory dump files from the current simulation and those of the golden run are different, we may decide that the target system is in the SDC state.

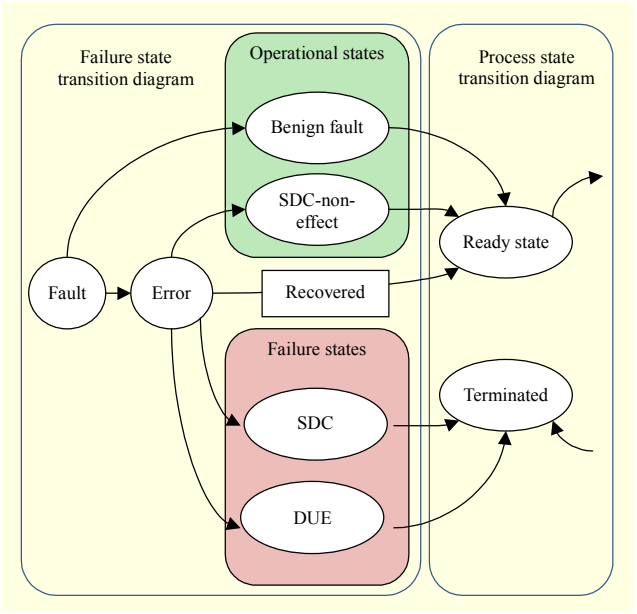

Fig. 4. Failure state transition diagram and process state transition diagram.

SDC-non-effect failure refers to a state in which errors have occurred, but the target system has completed the operation correctly. If the two memory dump files are identical but the VCD dump files are different, the target is in the SDC-non-effect error state.

Benign fault refers to a state in which the fault is masked so that the application produces the correct results. If the two memory dump files and the two VCD files are identical, then the target is in the benign fault state.

DUE refers a state in which the target system detected the errors but failed to recover the errors. DUE suggests the

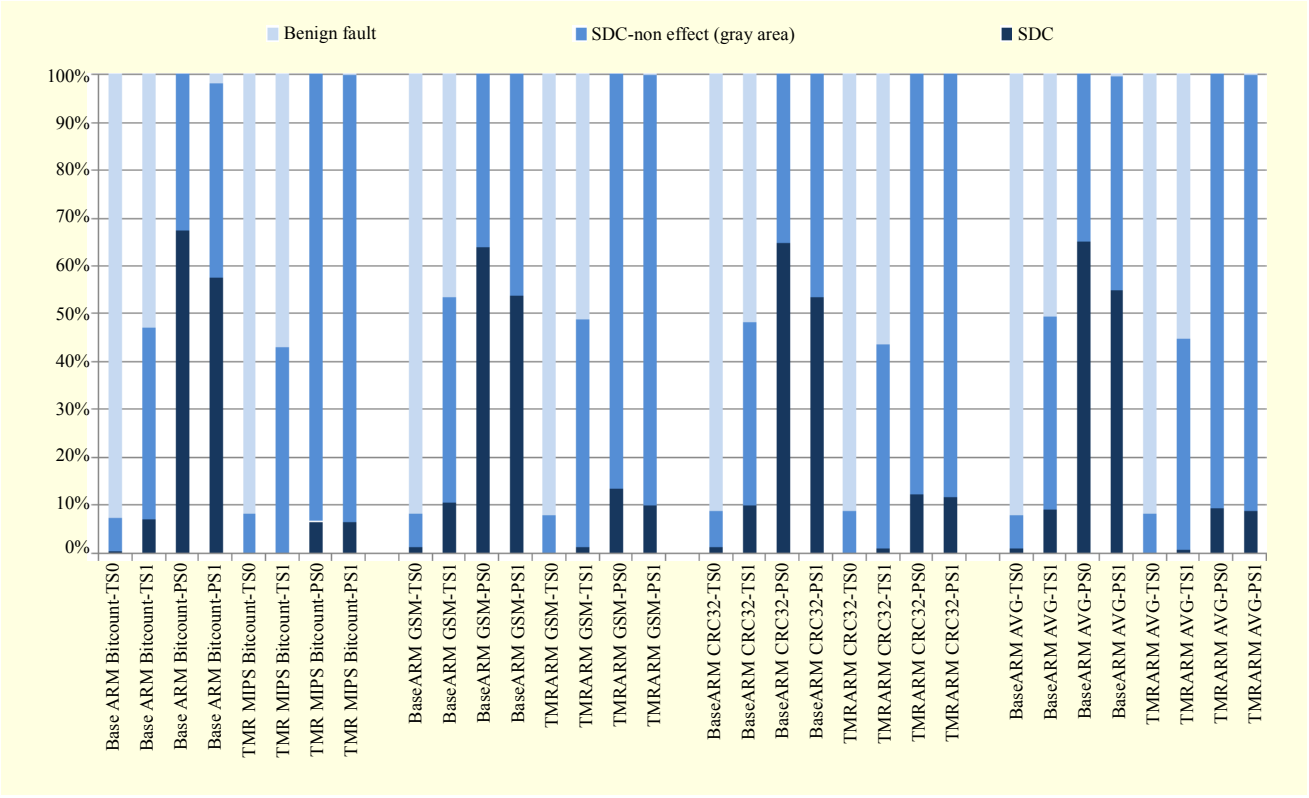

Fig. 5. Simulation results of fault injection campaign on the base ARM and TMR ARM. Note that TS0/1 and PS0/1 represents transient stuck-at-0/1 and permanent stuck-at-0/1, respectively.

presence of a fault detection and recovery circuit. Because the base ARM and TMR ARM do not support these circuits, we omit the DUE state.

Figure 5 summarizes the final fault injection simulation results using the base ARM and TMR ARM processors, the three benchmark software and eight fault models of Table 2. From these results, we may find the following.

First, the TMR ARM processor showed more resilience than the base ARM processor by two to five times. In particular, in the case of the permanent faults, the TMR ARM processor showed much higher reliability than the base ARM processor. For this reason, the mission computers of any commercial aircraft use the TMR architecture.

Second, observing the benign states of the transient and permanent fault types, the portion of the benign state of the transient fault was higher than that of the permanent fault. This demonstrates that the permanent fault type is more crucial than the transient fault type because it is more persistent, as expected.

Third, consider the stuck-at-1 and stuck-at-0 fault models of the SDC states of the transient fault type. The SDC of the stuck-at-1 fault model is higher than that of the stuck-at-0 fault model. This demonstrates that the stuck-at-1 fault model is more critical than the stuck-at-0 fault model. The reason why the stuck-at-1 fault model is more crucial is that the datapath has more 0's than 1's; therefore, the presence of the stuck-at-0 fault model may have more chance of being masked, while the presence of the stuck-at-1 fault model may have more chance of being detected.

C. Reliability Estimation

Figure 6 shows the reliability of the base ARM and the TMR ARM processor assuming the constant failure rate of λ as

$$R(t) = e^{-\lambda t}, \tag{2}$$

where the number of the failure states of the simulation results is set to be the failure rate, and t represents the operational time.

For both the transient and permanent fault models, the reliability of the TMR ARM was far better than that of the base ARM. The transient stuck-at-0/1 fault model of TMR ARMs 1 and 2 showed more resilience than that of the single core ARMs 3 and 6. Also, the permanent stuck-at-0/1 fault model of TMR ARMs 4 and 5 showed more reliability than that of the single core ARMs 7 and 8.

In the figure, we may find that the permanent stuck-at-0 fault model of TMR ARM 4 showed lower reliability than the transient stuck-at-0 fault model of single-core ARM 3. This illustrates that we must evaluate the dependability of any given

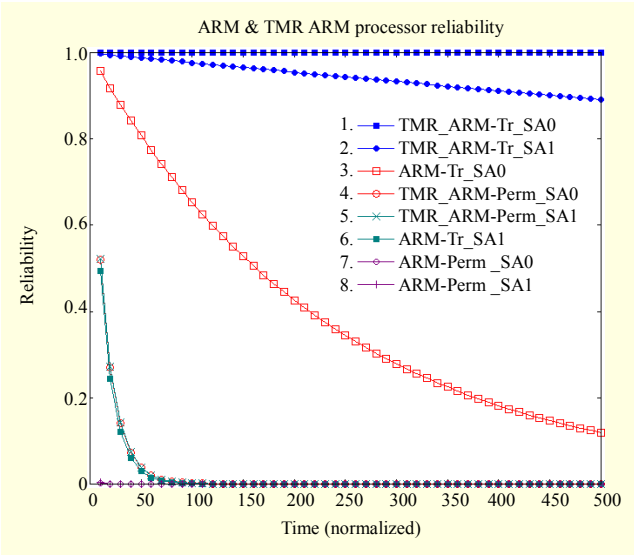

Fig. 6. Reliability of base ARM and TMR ARM for four fault models.

architecture in conjunction with environmental factors such as the fault models as well as the operational profiles. We have demonstrated that the proposed VFI can be useful to the qualitative analysis of the behavioral pattern of the fault model in the target design model.

V. Conclusion

We presented a novel simulated fault injection technique, called VFI, for the evaluation of the reliability of a Verilog hardware design model or a co-simulation target using the Verilog hardware model. VFI is unique in that the fault injection mechanism is not implemented in the Verilog simulation model; rather, it is implemented in the Verilog simulation kernel. Unlike co-simulation using ESL methodologies, RTL co-simulation can provide very detailed run-time execution results so that we may evaluate the failure rate or reliability of the target system with precision. Furthermore, unlike other simulated fault injection techniques, such as the saboteur technique or the mutant technique, the proposed VFI offers the most efficient and convenient simulation. The proposed Verilog fault injection offers the following advantages:

• Modification of the target Verilog design model for fault injection experiments is not necessary anymore. This advantage simplifies and expedites the dependability evaluation process of Verilog design models.

• The development period and cost of dependable embedded systems can be minimized. Early evaluation of the dependability of the target system is possible using the Verilog simulation model.

• The most realistic experimental results can be obtained. Using the synthesizable Verilog simulation model and co-simulation using the target application software on the Verilog model may provide experimental results close to those which could be obtained in a real fault injection experiment.

To validate VFI, we performed a dependability evaluation using a minimized version of the ARM7 RISC processor and a TMR ARM processor. The experimental results showed that the obtained failure rate of the two target processors varied significantly with respect to the different fault models injected into the system. As a result, we may use VFI to qualitatively evaluate the reliability of the target fault tolerant mechanism not at the component level but at the system level. Furthermore, this approach can be extended to qualitative evaluation of the reliability of the SoCs and embedded systems in the co-simulation environment consisting of the RTL simulation models and the application software.

References

[1] L. Anghel et al., "Multi-level Fault Effects Evaluation," *Radiation Effects on Embedded System,* Springer, 2007, pp. 69-88.

[2] D.W. Lee and J.W. Na, "A Novel Simulation Fault Injection Method for Dependability Analysis," *IEEE Design & Test Computers*, vol. 26, no. 6, Nov. 2009, pp. 50-61.

[3] Y. Yu and B.W. Johnson, "Fault Injection Techniques," in *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*, A. Benso and P. Prinetto, Eds., Kluwer Academic Publishers, 2003, pp. 7-39.

[4] M.C. Hsueh, T.K. Tsai, and R.K. Iyer, "Fault Injection Techniques and Tools," *Computer*, vol. 30, no. 4, Apr. 1997, pp. 75-82.

[5] D. Gil et al., "VHDL Simulation-Based Fault Injection Techniques," *in Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*, A. Benso and P. Prinetto, Eds., Kluwer Academic Publishers, 2003, pp. 159-176.

[6] P.S. Roop, A. Sowmya, and S. Ramesh, "Forced Simulation: A Technique for Automating Component Reuse in Embedded Systems," *ACM Trans. Design Auto. Electron. Syst.*, vol. 6 no. 4, Oct. 2001, pp. 602-628.

[7] W. Savage, J. Chilton, and R. Composano, "IP Reuse in the System on a Chip Era," *Proc. 13th Int. Symp. Syst. Synthesis*, Madrid, Spain, Sept. 2000, pp. 2-7.

[8] "IEEE Standard for Verilog Hardware Description Language," *IEEE Computer*, Apr. 2006.

[9] W.J. Koh, D.W. Lee, and J.W. Na, "Cost Efficient Fault Tolerant Dual Processor for Soft Errors," submitted for publication.

[10] B.Y. Kim, "Base ARM Technical Report," Technical Report, Korea Aviation Univ., 2009.

[11] S. Williams and M. Baxter, "Icarus Verilog: Open-Source Verilog More Than a Year Later," *Linux J.*, vol. 2002, no. 99, Jul. 2002.
[12] T.J. Slegel et al, "IBM's S/390 G5 Microprocessor Design," *IEEE Micro*., vol. 19, no. 2, Mar/Apr. 1999, pp. 12-23.
[13] H. Ando et al., "A 1.3-GHz Fifth-Generation SPARC64 Microprocessor," *IEEE J. Solid State Circuits*, vol. 38, no. 11, Nov. 2003, pp. 1896-1905.
[14] M.R. Guthaus et al., "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," *IEEE 4th Annual Workshop Workload Characterization*, Austin, TX, 2001, pp. 3-14.
[15] D. Seal, *ARM Architecture Reference Manual*, Addison-Wesley Professional, 2001, pp. 2-9.
[16] L. Li et al., "Toward Distributed Verilog Simulation," *Int. J. Simulation Syst. Sci. Technol.*, vol. 4, no. 3-4, 2003, pp. 44-54.
[17] L. Zhu et al., "Parallel Logic Simulation of Million-Gate VLSI Circuits," *Proc. 13th IEEE Int. Symp. Modeling, Anal. Simulation Computer Telecom. Syst.*, GA, USA, Sept. 2005, pp. 521-524.
[18] Shubu Mukherjee, *Architecture Design for Soft Errors*, Morgan Kaufmann Publishers, 2007, pp.149-150, 154.
[19] C.T. Weaver et al., "Reducing the Soft-Error Rate of a High-Performance Microprocessor," *IEEE Micro.*, vol. 24, no. 6, Nov.-Dec. 2004, pp. 30-37.

Jongwhoa Na received the BS in electronics engineering from Sogang University, the MS in computer engineering from Wayne State University, and the PhD in computer engineering from the University of Arizona. He is a professor of electronics engineering and avionics at Korea Aerospace University. His research interests include computer architecture, dependable embedded systems, and assistive embedded systems. He is a member of IEEE.

Dongwoo Lee received the MS in electronic engineering from Korea Aerospace University. He is a PhD candidate in electronic engineering at Korea Aerospace University. His research interests include SoC design, hardware-in-the-loop simulation, and dependable embedded system design. He is a student member of IEEE.