Vulnerability Analysis and Risk Assessment for SoCs Used in Safety-Critical Embedded Systems

Yung-Yuan Chen and Tong-Ying Juang
National Taipei University
Taiwan

1. Introduction

Intelligent systems, such as intelligent automotive systems or intelligent robots, require a rigorous reliability/safety while the systems are in operation. As system-on-chip (*SoC*) becomes more and more complicated, the *SoC* could encounter the reliability problem due to the increased likelihood of faults or radiation-induced soft errors especially when the chip fabrication enters the very deep submicron technology [Baumann, 2005; Constantinescu, 2002; Karnik et al., 2004; Zorian et al., 2005]. *SoC* becomes prevalent in the intelligent safety-related applications, and therefore, fault-robust design with the safety validation is required to guarantee that the developed *SoC* is able to comply with the safety requirements defined by the international norms, such as IEC 61508 [Brown, 2000; International Electrotechnical Commission [IEC], 1998-2000]. Therefore, safety attribute plays a key metric in the design of *SoC* systems. It is essential to perform the safety validation and risk reduction process to guarantee the safety metric of *SoC* before it is being put to use.

If the system safety level is not adequate, the risk reduction process, which consists of the vulnerability analysis and fault-robust design, is activated to raise the safety to the required level. For the complicated IP-based *SoCs* or embedded systems, it is unpractical and not cost-effective to protect the entire *SoC* or system. Analyzing the vulnerability of microprocessors or *SoCs* can help designers not only invest limited resources on the most crucial regions but also understand the gain derived from the investments [Hosseinabady et al., 2007; Kim & Somani, 2002; Mariani et al., 2007; Mukherjee et al., 2003; Ruiz et al., 2004; Tony et al., 2007; Wang et al., 2004].

The previous literature in estimating the vulnerability and failure rate of systems is based on either the analytical methodology or the fault injection approach at various system modeling levels. The fault injection approach was used to assess the vulnerability of high-performance microprocessors described in Verilog hardware description language at RTL design level [Kim & Somani, 2002; Wang et al., 2004]. The authors of [Mukherjee et al., 2003] proposed a systematic methodology based on the concept of architecturally correct execution to compute the architectural vulnerability factor. [Hosseinabady et al., 2007] and [Tony et al., 2007] proposed the analytical methods, which adopted the concept of timing vulnerability factor and architectural vulnerability factor [Mukherjee et al., 2003] respectively to estimate the vulnerability and failure rate of *SoCs*, where a UML-based real time description was employed to model the systems.

The authors of [Mariani et al., 2007] presented an innovative failure mode and effects analysis (FMEA) method at *SoC*-level design in RTL description to design in compliance with IEC61508. The methodology presented in [Mariani et al., 2007] was based on the concept of sensible zone to analyze the vulnerability and to validate the robustness of the target system. A memory sub-system embedded in fault-robust microcontrollers for automotive applications was used to demonstrate the feasibility of their FMEA method. However, the design level in the scheme presented in [Mariani et al., 2007] is RTL level, which may still require considerable time and efforts to implement a *SoC* using RTL description due to the complexity of oncoming *SoC* increasing rapidly. A dependability benchmark for automotive engine control applications was proposed in paper [Ruiz et al., 2004]. The work showed the feasibility of the proposed dependability benchmark using a prototype of diesel electronic control unit (ECU) control engine system. The fault injection campaigns were conducted to measure the dependability of benchmark prototype. The domain of application for dependability benchmark specification presented in paper [Ruiz et al., 2004] confines to the automotive engine control systems which were built by commercial off-the-shelf (COTS) components. While dependability evaluation is performed after physical systems have been built, the difficulty of performing fault injection campaign is high and the costs of re-designing systems due to inadequate dependability can be prohibitively expensive.

It is well known that FMEA [Mikulak et al., 2008] and fault tree analysis (FTA) [Stamatelatos et al., 2002] are two effective approaches for the vulnerability analysis of the *SoC*. However, due to the high complexity of the *SoC*, the incorporation of the FMEA/FTA and fault-tolerant demand into the *SoC* will further raise the design complexity. Therefore, we need to adopt the behavioral level or higher level of abstraction to describe/model the *SoC*, such as using SystemC, to tackle the complexity of the *SoC* design and verification. An important issue in the design of *SoC* is how to validate the system dependability as early in the development phase to reduce the re-design cost and time-to-market. As a result, a *SoC*-level safety process is required to facilitate the designers in assessing and enhancing the safety/robustness of a *SoC* with an efficient manner.

Previously, the issue of *SoC*-level vulnerability analysis and risk assessment is seldom addressed especially in SystemC transaction-level modeling (TLM) design level [Thorsten et al., 2002; Open SystemC Initiative [OSCI], 2003]. At TLM design level, we can more effectively deal with the issues of design complexity, simulation performance, development cost, fault injection, and dependability for safety-critical *SoC* applications. In this study, we investigate the effect of soft errors on the *SoCs* for safety-critical systems. An IP-based *SoC*-level safety validation and risk reduction (SVRR) process combining FMEA with fault injection scheme is proposed to identify the potential failure modes in a *SoC* modeled at SystemC TLM design level, to measure the risk scales of consequences resulting from various failure modes, and to locate the vulnerability of the system. A *SoC* system safety verification platform was built on the SystemC *CoWare Platform Architect* design environment to demonstrate the core idea of SVRR process. The verification platform comprises a system-level fault injection tool and a vulnerability analysis and risk assessment tool, which were created to assist us in understanding the effect of faults on system behavior, in measuring the robustness of the system, and in identifying the critical parts of the system during the *SoC* design process under the environment of *CoWare Platform Architect*.

Since the modeling of *SoCs* is raised to the level of TLM abstraction, the safety-oriented analysis can be carried out efficiently in early design phase to validate the safety/robustness of the *SoC* and identify the critical components and failure modes to be protected if necessary. The proposed SVRR process and verification platform is valuable in that it provides the capability to quickly assess the *SoC* safety, and if the measured safety cannot meet the system requirement, the results of vulnerability analysis and risk assessment will be used to help us develop a feasible and cost-effective risk reduction process. We use an ARM-based *SoC* to demonstrate the robustness/safety validation process, where the soft errors were injected into the register file of ARM CPU, memory system, and AMBA AHB.

The remaining paper is organized as follows. In Section 2, the SVRR process is presented. A risk model for vulnerability analysis and risk assessment is proposed in the following section. In Section 4, based on the SVRR process, we develop a *SoC*-level system safety verification platform under the environment of *CoWare Platform Architect*. A case study with the experimental results and a thorough vulnerability and risk analysis are given in Section 5. The conclusion appears in Section 6.

2. Safety validation and risk reduction process

We propose a SVRR process as shown in Fig. 1 to develop the safety-critical electronic systems. The process consists of three phases described as follows:

Phase 1 (fault hypothesis): this phase is to identify the potential interferences and develop the fault injection strategy to emulate the interference-induced errors that could possibly occur during the system operation.

Phase 2 (vulnerability analysis and risk assessment): this phase is to perform the fault injection campaigns based on the Phase 1 fault hypothesis. Throughout the fault injection campaigns, we can identify the failure modes of the system, which are caused by the faults/errors injected into the system while the system is in operation. The probability distribution of failure modes can be derived from the fault injection campaigns. The risk-priority number (RPN) [Mollah, 2005] is then calculated for the components inside the electronic system. A component's *RPN* aims to rate the risk of the consequence caused by component's failure. RPN can be used to locate the critical components to be protected. The robustness of the system is computed based on the adopted robustness criterion, such as safety integrity level (SIL) defined in the IEC 61508 [IEC, 1998-2000]. If the robustness of the system meets the safety requirement, the system passes the validation; else the robustness/safety is not adequate, so Phase 3 is activated to enhance the system robustness/safety.

Phase 3 (fault-tolerant design and risk reduction): This phase is to develop a feasible risk-reduction approach by fault-tolerant design, such as the schemes presented in [Austin, 1999; Mitra et al., 2005; Rotenberg, 1999; Slegel et al., 1999;], to improve the robustness of the critical components identified in Phase 2. The enhanced version then goes to Phase 2 to recheck whether the adopted risk-reduction approach can satisfy the safety/robustness requirement or not.

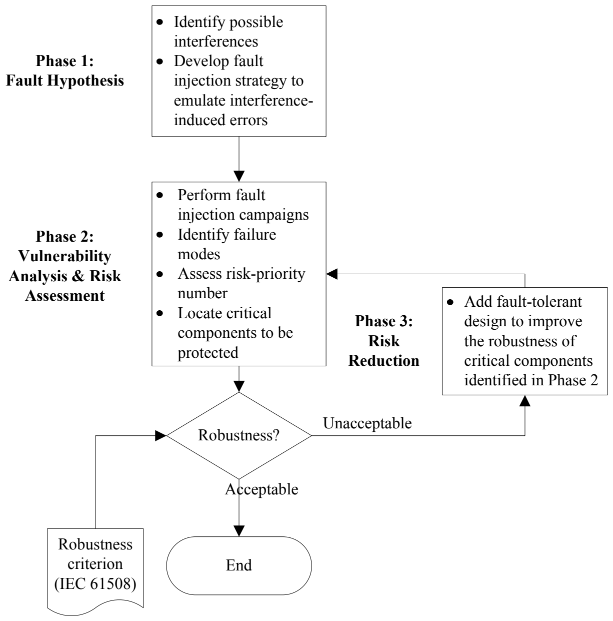

Phase 1: Fault Hypothesis
• Identify possible interferences
• Develop fault injection strategy to emulate interference-induced errors
Phase 2: Vulnerability Analysis & Risk Assessment
• Perform fault injection campaigns
• Identify failure modes
• Assess risk-priority number
• Locate critical components to be protected
Phase 3: Risk Reduction
• Add fault-tolerant design to improve the robustness of critical components identified in Phase 2
Robustness?
Unacceptable
Acceptable
Robustness criterion (IEC 61508)
End

Fig. 1. Safety validation and risk reduction process.

3. Vulnerability analysis and risk assessment

Analyzing the vulnerability of *SoCs* or systems can help designers not only invest limited resources on the most crucial region but also understand the gain derived from the investment. In this section, we propose a *SoC*-level risk model to quickly assess the *SoC's* vulnerability at SystemC TLM level. Conceptually, our risk model is based on the FMEA method with the fault injection approach to measure the robustness of *SoCs*. From the assessment results, the rank of component vulnerability related to the risk scale of causing the system failure can be acquired. The notations used in the risk model are developed below.

- *n*: number of components to be investigated in the *SoC*;
- *z*: number of possible failure modes of the *SoC*;
- *C(i)*: the i^{th} component, where $1 \le i \le n$;
- *ER_C(i)*: raw error rate of the i^{th} component;
- *SFR_C(i)*: the part of *SoC* failure rate contributed from the error rate of the i^{th} component;
- *SFR*: *SoC* failure rate;
- *FM(k)*: the k^{th} failure mode of the *SoC*, where $1 \le k \le z$;
- *NE*: no effect which means that a fault/error happening in a component has no impact on the *SoC* operation at all;
- *P (i, FM(K))*: probability of *FM(K)* if an error occurs in the i^{th} component;
- *P (i, NE)*: probability of no effect for an error occurring in the i^{th} component;
- *P(i, SF)*: probability of *SoC* failure for an error occurring in the i^{th} component;

- *SR_FM(k)*: severity rate of the effect of *k*th failure mode, where $1 \leq k \leq z$;
- *RPN_C(i)*: risk priority number of the *i*th component;
- *RPN_FM(k)*: risk priority number of the *k*th failure mode.

3.1 Fault hypothesis

It is well known that the rate of soft errors caused by single event upset (SEU) increases rapidly while the chip fabrication enters the very deep submicron technology [Baumann, 2005; Constantinescu, 2002; Karnik et al., 2004; Zorian et al., 2005]. Radiation-induced soft errors could cause a serious dependability problem for *SoCs*, electronic control units, and nodes used in the safety-critical applications. The soft errors may happen in the flip-flop, register file, memory system, system bus and combinational logic. In this work, single soft error is considered in the derivation of risk model.

3.2 Risk model

The potential effects of faults on *SoC* can be identified from the fault injection campaigns. We can inject the faults into a specific component, and then investigate the effect of component's errors on the *SoC* behaviors. Throughout the injection campaigns for each component, we can identify the failure modes of the *SoC*, which are caused by the errors of components in the *SoC*. The parameter *P(i, FM(k))* defined before can be derived from the fault injection campaigns.

In general, the following failure behaviors: fatal failure (FF), such as system crash or process hang, silent data corruption (SDC), correct data/incorrect time (CD/IT), and infinite loop (IL) (note that we declare the failure as IL if the execution of benchmark exceeds the 1.5 times of normal execution time), which were observed from our previous work, represent the possible *SoC* failure modes caused by the faults occurring in the components. Therefore, we adopt those four *SoC* failure modes in this study to demonstrate our risk assessment approach. We note that a fault may not cause any trouble at all, and this phenomenon is called no effect of the fault.

One thing should be pointed out that to obtain the highly reliable experimental results to analyze the robustness/safety and vulnerability of the target system we need to perform the adequate number of fault injection campaigns to guarantee the validity of the statistical data obtained. In addition, the features of benchmarks could also affect the system response to the faults. Therefore, several representative benchmarks are required in the injection campaigns to enhance the confidence level of the statistical data.

In the derivation of *P(i, FM(K))*, we need to perform the fault injection campaigns to collect the fault simulation data. Each fault injection campaign represents an experiment by injecting a fault into the *i*th component, and records the fault simulation data, which will be used in the failure mode classification procedure to identify which failure mode or no effect the *SoC* encountered in this fault injection campaign. The failure mode classification procedure inputs the fault-free simulation data, and fault simulation data derived from the fault injection campaigns to analyze the effect of faults occurring in the *i*th component on the *SoC* behavior based on the classification rules for potential failure modes.

The derivation process of *P(i, FM(K))* by fault injection process is described below. Several notations are developed first:

- *SoC_FM*: a set of *SoC* failure modes used to record the possible *SoC* failure modes happened in the fault injection campaigns.
- *counter(i, k)*: an array which is used to count the number of the k^{th} *SoC* failure mode occurring in the fault injection experiments for the i^{th} component, where $1 \leq i \leq n$, and $1 \leq k \leq z$. *counter(i, z+1)* is used to count the number of no effect in the fault injection campaigns.
- *no_fi(i)*: the number of fault injection campaigns performed in the i^{th} component, where $1 \leq i \leq n$.

Fault injection process:

```
z = 4; SoC_FM = {FF, SDC, CD/IT, IL};
for i = 1 to n                //fault injection experiments for the ith component;//
{for j = 1 to no_fi(i)
  {//injecting a fault into the ith component, and investigating the effect of component's
  fault on the SoC behavior by failure mode classification procedure; the result of classification
  is recorded in the parameter 'classification'.//
    switch (classification)
   { case 'FF': counter(i, 1) = counter(i, 1) + 1;
     case 'SDC': counter(i, 2) = counter(i, 2) + 1;
     case 'CD/IT': counter(i, 3) = counter(i, 3) + 1;
     case 'IL': counter(i, 4) = counter(i, 4) + 1;
     case 'NE': counter(i, 5) = counter(i, 5) + 1;}

}}
```

The failure mode classification procedure is used to classify the *SoC* failure modes caused by the component's faults. For a specific benchmark program, we need to perform a fault-free simulation to acquire the golden results that are used to assist the failure mode classification procedure in identifying which failure mode or no effect the *SoC* encountered in this fault injection campaign.

Failure mode classification procedure:

Inputs: fault-free simulation golden data and fault simulation data for an injection campaign;

Output: *SoC* failure mode caused by the component's fault or no effect of the fault in this injection campaign.

```
{if (execution of fault simulation is complete)
   then if (execution time of fault simulation is the same as execution time of fault-free
            simulation)
       then if (execution results of fault simulation are the same as execution results of
                fault-free simulation)
           then classification := 'NE';
           else classification := 'SDC';
       else if (execution results of fault simulation are the same as execution results of fault-
               free simulation)
            then classification := 'CD/IT';
            else classification := 'SDC';
    else if (execution of benchmark exceeds the 1.5 times of normal execution time)
        then classification := 'IL';
        else //execution of fault simulation was hung or crash due to the injected fault;//
            classification := 'FF';
}
```

After carrying out the above injection experiments, the parameter of *P(i, FM(K))* can be computed by

$$P(i,FM(K)) = \frac{counter(i,k)}{no\_fi(i)}$$

Where $1 \le i \le n$ and $1 \le k \le z$. The following expressions are exploited to evaluate the terms of *P(i, SF)* and *P(i, NE)*.

$$P(i,SF) = \sum_{k=1}^{z} P(i,FM(k))$$

$$P(i,NE) = 1 - P(i,SF)$$

The derivation of the component's raw error rate is out of the scope of this paper, so we here assume the data of *ER_C(i)*, for $1 \le i \le n$, are given. The part of *SoC* failure rate contributed from error rate of the i^{th} component can be calculated by

$$SFR\_C(i) = ER\_C(i) \times P(i,SF)$$

If each component *C(i)*, $1 \le i \le n$, must operate correctly for the *SoC* to operate correctly and also assume that other components not shown in *C(i)* list are fault-free, the *SoC* failure rate can be written as

$$SFR = \sum_{i=1}^{n} SFR\_C(i)$$

The meaning of the parameter *SR_FM(k)* and the role it playing can be explained from the aspect of FMEA process [Mollah, 2005]. The method of FMEA is to identify all possible failure modes of a *SoC* and analyze the effects or consequences of the identified failure modes. In general, an FMEA records each potential failure mode, its effect in the next level, and the cause of failure. We note that the faults occurring in different components could cause the same *SoC* failure mode, whereas the severity degree of the consequences resulting from various *SoC* failure modes could not be identical. The parameter *SR_FM(k)* is exploited to express the severity rate of the consequence resulting from the k^{th} failure mode, where $1 \le k \le z$.

We illustrate the risk evaluation with FMEA idea using the following example. An ECU running engine control software is employed for automotive engine control. Its outputs are used to control the engine operation. The ECU could encounter several types of output failures due to hardware or software faults in ECU. The various types of failure mode of ECU outputs would result in different levels of risk/criticality on the controlled engine. A risk assessment is performed to identify the potential failure modes of ECU outputs as well as the likelihood of failure occurrence, and estimate the resulting risks of the ECU-controlled engine.

In the following, we propose an effective *SoC*-level FMEA method to assess the risk-priority number (*RPN*) for the components inside the *SoC* and for the potential *SoC* failure modes. A component's *RPN* aims to rate the risk of the consequences caused by component's faults. In other words, a component's *RPN* represents how serious is the impact of component's errors on the system safety. A risk assessment should be carried out to identify the critical components within a *SoC* and try to mitigate the risks caused by those critical components. Once the critical components and their risk scales have been identified, the risk-reduction process, for example fault-tolerant design, should be activated to improve the system dependability. *RPN* can also give the protection priority among the analyzed components. As a result, a feasible risk-reduction approach can be developed to effectively protect the vulnerable components and enhance the system robustness and safety.

The parameter *RPN_C(i)*, i.e. risk scale of failures occurring in the i^{th} component, can be computed by

$$RPN\_C(i) = ER\_C(i) \times \sum_{k=1}^{z} P(i, FM(k)) \times SR\_FM(k)$$

where $1 \le i \le n$. The expression of *RPN_C(i)* contains three terms which are, from left to right, error rate of the i^{th} component, probability of *FM(K)* if a fault occurs in the i^{th} component, and severity rate of the k^{th} failure mode. As stated previously, a component's fault could result in several different system failure modes, and each identified failure mode has its potential impact on the system safety. So, *RPN_C(i)* is the summation of the following expression *ER_C(i)* × *P (i, FM(K))* × *SR_FM(k)*, for *k* from one to *z*. The term of *ER_C(i)* × *P (i, FM(K))* represents the occurrence rate of the k^{th} failure mode, which is caused by the i^{th} component failing to perform its intended function.

The *RPN_FM(k)* represents the risk scale of the k^{th} failure mode, which can be calculated by

$$RPN\_FM(k) = SR\_FM(k) \times \sum_{i=1}^{n} ER\_C(i) \times P(i, FM(k))$$

where $1 \le k \le z$. $\sum_{i=1}^{n} ER\_C(i) \times P(i, FM(k))$ expresses the occurrence rate of the k^{th} failure mode in a *SoC*. This sort of assessment can reveal the risk levels of the failure modes to its system and identify the major failure modes for protection so as to reduce the impact of failures to the system safety.

4. System safety verification platform

We have created an effective safety verification platform to provide the capability to quickly handle the operation of fault injection campaigns and dependability analysis for the system design with SystemC. The core of the verification platform is the fault injection tool [Chang & Chen, 2007; Chen et al., 2008] under the environment of *CoWare Platform Architect* [CoWare, 2006], and the vulnerability analysis and risk assessment tool. The tool is able to deal with the fault injection at the following levels of abstraction [Chang & Chen, 2007; Chen et al., 2008]: bus-cycle accurate level, untimed functional TLM with primitive channel sc_fifo, and timed functional TLM with hierarchical channel. An interesting feature of our fault injection tool is to offer not only the time-triggered but also the event-triggered methodologies to decide when to inject a fault. Consequently, our injection tool can significantly reduce the effort and time for performing the fault injection campaigns. Combining the fault injection tool with vulnerability analysis and risk assessment tool, the verification platform can dramatically increase the efficiency of carrying out the system robustness validation and vulnerability analysis and risk assessment. For the details of our fault injection tool, please refer to [Chang & Chen, 2007; Chen et al., 2008].

However, the IP-based *SoCs* designed by *CoWare Platform Architect* in SystemC design environment encounter the injection controllability problem. The simulation-based fault injection scheme cannot access the fault targets inside the IP components imported from other sources. As a result, the injection tool developed in SystemC abstraction level may lack the capability to inject the faults into the inside of the imported IP components, such as CPU or DSP. To fulfill this need, we exploit the software-implemented fault injection scheme [Sieh, 1993; Kanawati et al., 1995] to supplement the injection ability. The software-implemented fault injection scheme, which uses the system calls of Unix-type operating system to implement the injection of faults, allows us to inject the faults into the targets of storage elements in processors, like register file in CPU, and memory systems. As discussed, a complete IP-based *SoC* system-level fault injection tool should consist of the software-implemented and simulation-based fault injection schemes.

Due to the lack of the support of Unix-type operating system in *CoWare Platform Architect*, the current version of safety verification platform cannot provide the software-implemented fault injection function in the tool. Instead, we employed a physical system platform built by ARM-embedded *SoC* running Linux operating system to validate the developed software-implemented fault injection mechanism. We note that if the *CoWare Platform Architect* can support the UNIX-type operating system in the SystemC design environment, our software-implemented fault injection concept should be brought in the SystemC design platform. Under the circumstances, we can implement the so called hybrid fault injection approach, which comprises the software-implemented and simulation-based fault injection methodologies, in the SystemC design environment to provide more variety of injection functions.

5. Case study

An ARM926EJ-based *SoC* platform provided by *CoWare Platform Architect* [CoWare, 2006] was used to demonstrate the feasibility of our risk model. The illustrated *SoC* platform was modeled at the timed functional TLM abstraction level. This case study is to investigate three important components, which are register file in ARM926EJ, AMBA Advanced High-performance Bus (AHB), and the memory sub-system, to assess their risk scales to the *SoC*-controlled system. We exploited the safety verification platform to perform the fault injection process associated with the risk model presented in Section 3 to obtain the risk-related parameters for the components mentioned above. The potential *SoC* failure modes classified from the fault injection process are fatal failure (FF), silent data corruption (SDC), correct data/incorrect time (CD/IT), and infinite loop (IL). In the following, we summarize the data used in this case study.

- *n* = 3, {*C(1)*, *C(2)*, *C(3)*} = {AMBA AHB, memory sub-system, register file in ARM926EJ}.
- *z* = 4, {*FM(1)*, *FM(2)*, *FM(3)*, *FM(4)*} = {FF, SDC, CD/IT, IL}.
- The benchmarks employed in the fault injection process are: JPEG (pixels: 255 × 154), matrix multiplication (M-M: 50 × 50), quicksort (QS: 3000 elements) and FFT (256 points).

5.1 AMBA AHB experimental results

The system bus, such as AMBA AHB, provides an interconnected platform for IP-based *SoC*. Apparently, the robustness of system bus plays an important role in the *SoC* reliability. It is evident that the faults happening in the bus signals will lead to the data transaction errors and finally cause the system failures. In this experiment, we choose three bus signals HADDR[31:0], HSIZE[2:0], and HDATA[31:0] to investigate the effect of bus errors on the system. The results of fault injection process for AHB system bus under various benchmarks are shown in Table 1 and 2. The results of a particular benchmark in Table 1 and 2 were derived from the six thousand fault injection campaigns, where each injection campaign injected 1-bit flip fault to bus signals. The fault duration lasts for the length of one-time data transaction. The statistics derived from six thousand times of fault injection campaigns have been verified to guarantee the validity of the analysis.

From Table 1, it is evident that the susceptibility of the *SoC* to bus faults is benchmark-dependent and the rank of system bus vulnerability over different benchmarks is JPEG > M-M > FFT > QS. However, all benchmarks exhibit the same trend in that the probabilities of FF show no substantial difference, and while a fault arises in the bus signals, the occurring probabilities of SDC and FF occupy the top two ranks. The results of the last row offer the average statistics over four benchmarks employed in the fault injection process. Since the probabilities of *SoC* failure modes are benchmark-variant, the average results illustrated in Table 1 give us the expected probabilities for the system bus vulnerability of the developing *SoC*, which are very valuable for us to gain the robustness of the system bus and the probability distribution of failure modes. The robustness measure of the system bus is only 26.78% as shown in Table 1, which means that a fault occurring in the system bus, the *SoC* has the probability of 26.78% to survive for that fault.

The experimental results shown in Table 2 are probability distribution of failure modes with respect to the various bus signal errors for the used benchmarks. From the data illustrated in the NE column, we observed that the most vulnerable part is the address bus HADDR[31:0]. Also from the data displayed in the FF column, the faults occurring in address bus will have the probability between 38.9% and 42.3% to cause a serious fatal failure for the used benchmarks. The HSIZE and HDATA signal errors mainly cause the SDC failure. In summary, our results reveal that the address bus HADDR should be protected first in the design of system bus, and the SDC is the most popular failure mode for the demonstrated *SoC* responding to the bus faults or errors.

	FF (%)	SDC (%)	CD/IT (%)	IL(%)	SF (%)	NE (%)
JPEG	18.57	45.90	0.16	15.88	80.51	19.49
M-M	18.95	55.06	2.15	3.57	79.73	20.27
FFT	20.18	21.09	15.74	6.38	63.39	36.61
QS	20.06	17.52	12.24	5.67	55.50	44.50
Avg.	19.41	38.16	7.59	8.06	73.22	26.78

Table 1. *P (1, FM(K))*, *P (1, SF)* and *P (1, NE)* for the used benchmarks.

	FF (%)				SDC (%)				CD/IT (%)			
	1	2	3	4	1	2	3	4	1	2	3	4
HADDR	38.9	39.7	42.3	42	42.9	43.6	18.2	15.2	0.08	1.94	14.4	11.4
HSIZE	0.16	0.0	0.0	0	68.2	67.6	25.6	22.6	0.25	9.64	37.4	38.5
HDATA	0.0	0.0	0.0	0	46.8	65.4	23.6	19.4	0.24	1.66	15.0	10.6

	IL (%)				NE (%)			
	1	2	3	4	1	2	3	4
HADDR	11.5	2.02	3.41	2.02	6.62	12.7	21.7	29.4
HSIZE	11.6	2.38	6.97	7.53	19.8	20.4	30.0	31.4
HDATA	20.7	5.23	9.29	9.15	32.3	27.7	52.1	60.9

Table 2. Probability distribution of failure modes with respect to various bus signal errors for the used benchmarks (1, 2, 3 and 4 represent the jpeg, m-m, fft and qs benchmark, respectively).

5.2 Memory sub-system experimental results

The memory sub-system could be affected by the radiation articles, which may cause the bit-flipped soft errors. However, the bit errors won't cause damage to the system operation if one of the following situations occurs:

- Situation 1: The benchmark program never reads the affected words after the bit errors happen.
- Situation 2: The first access to the affected words after the occurrence of bit errors is the 'write' action.

Otherwise, the bit errors could cause damage to the system operation. Clearly, if the first access to the affected words after the occurrence of bit errors is the 'read' action, the bit errors will be propagated and could finally lead to the failures of *SoC* operation. So, whether the bit errors will become fatal or not, it all depends on the occurring time of bit errors, the locations of affected words, and the benchmark's memory access patterns after the occurrence of bit errors.

According to the above discussion, two interesting issues arise; one is the propagation probability of bit errors and another is the failure probability of propagated bit errors. We define the propagation probability of bit errors as the probability of bit errors which will be read out and propagated to influence the execution of the benchmarks. The failure probability of propagated bit errors represents the probability of propagated bit errors which will finally result in the failures of *SoC* operation.

Initially, we tried performing the fault injection campaigns in the *CoWare Platform Architect* to collect the simulation data. After a number of fault injection and simulation campaigns, we realized that the length of experimental time will be a problem because a huge amount of fault injection and simulation campaigns should be conducted for each benchmark and several benchmarks are required for the experiments. From the analysis of the campaigns, we observed that a lot of bit-flip errors injected to the memory sub-system fell into the Situation 1 or 2, and therefore, we must carry out an adequate number of fault injection campaigns to obtain the validity of the statistical data.

To solve this dilemma, we decide to perform two types of experiments termed as Type 1 experiment and Type 2 experiment, or called hybrid experiment, to assess the propagation probability and failure probability of bit errors, respectively. As explained below, Type 1 experiment uses a software tool to emulate the fault injection and simulation campaigns to quickly gain the propagation probability of bit errors, and the set of propagated bit errors. The set of propagated bit errors will be used in the Type 2 experiment to measure the failure probability of propagated bit errors.

Type 1 experiment: we develop the experimental process as described below to measure the propagation probability of bit errors. The following notations are used in the experimental process.

- N_{bench}: the number of benchmarks used in the experiments.
- $N_{inj}(j)$: the number of fault injection campaigns performed in the j^{th} benchmark's experiment.
- $C_{p\text{-}b\text{-}err}$: counter of propagated bit errors.
- $N_{p\text{-}b\text{-}err}$: the expected number of propagated bit errors.
- S_m: address space of memory sub-system.
- $N_{d\text{-}t}$: the number of read/write data transactions occurring in the memory sub-system during the benchmark execution.
- T_{error}: the occurring time of bit error.
- A_{error}: the address of affected memory word.
- $S_{p\text{-}b\text{-}err}(j)$: set of propagated bit errors conducted in the j^{th} benchmark's experiment.
- $P_{p\text{-}b\text{-}err}$: propagation probability of bit errors.

Experimental Process: We injected a bit-flipped error into a randomly chosen memory address at random read/write transaction time for each injection campaign. As stated earlier, this bit error could either be propagated to the system or not. If yes, then we add one to the parameter $C_{p\text{-}b\text{-}err}$. The parameter $N_{p\text{-}b\text{-}err}$ is set by users and employed as the terminated condition for the current benchmark's experiment. When the value of $C_{p\text{-}b\text{-}err}$ reaches to $N_{p\text{-}b\text{-}err}$, the process of current benchmark's experiment is terminated. The $P_{p\text{-}b\text{-}err}$ can then be derived from $N_{p\text{-}b\text{-}err}$ divided by N_{inj}. The values of N_{bench}, S_m and $N_{p\text{-}b\text{-}err}$ are given before performing the experimental process.

```
for $j$ = 1 to $N_{bench}$
{
Step 1: Run the $j^{th}$ benchmark in the experimental SoC platform under CoWare Platform
        Architect to collect the desired bus read/write transaction information that include
        address, data and control signals of each data transaction into an operational profile
        during the program execution. The value of $N_{d-t}$ can be obtained from this step.
Step 2: $C_{p\text{-}b\text{-}err} = 0$; $N_{inj}(j) = 0$;
    While $C_{p\text{-}b\text{-}err} < N_{p\text{-}b\text{-}err}$ do
        {$T_{error}$ can be decided by randomly choosing a number $x$ between one and $N_{d\text{-}t}$. It
        means that $T_{error}$ is equivalent to the time of the $x^{th}$ data transaction occurring in the
        memory sub-system. Similarly, $A_{error}$ is determined by randomly choosing an address
        between one and $S_m$. A bit is randomly picked up from the word pointed by $A_{error}$,
        and the bit selected is flipped. Here, we assume that the probability of fault
        occurrence of each word in memory sub-system is the same.
            If ((Situation 1 occurs) or (Situation 2 occurs))
            then {the injected bit error won't cause damage to the system operation;}
            else {$C_{p\text{-}b\text{-}err} = C_{p\text{-}b\text{-}err} + 1$;
                    record the related information of this propagated bit error to $S_{p\text{-}b\text{-}err}(j)$
                    including $T_{error}$, $A_{error}$ and bit location.}
            //Situation 1 and 2 are described in the beginning of this Section. The operational
            profile generated in Step 1 is exploited to help us investigate the resulting situation
            caused by the current bit error. From the operational profile, we check the memory
            access patterns beginning from the time of occurrence of bit error to identify which
            situation the injected bit error will lead to. //
            $N_{inj}(j) = N_{inj}(j) + 1$;}
}
```

For each benchmark, we need to perform the Step 1 of Type 1 experimental process once to obtain the operational profile, which will be used in the execution of Step 2. We then created a software tool to implement the Step 2 of Type 1 experimental process. We note that the created software tool emulates the fault injection campaigns required in Step 2 and checks the consequences of the injected bit errors with the support of operational profile derived from Step 1. It is clear to see that the Type 1 experimental process does not utilize the simulation-based fault injection tool implemented in safety verification platform as described in Section 4. The reason why we did not exploit the safety verification platform in this experiment is the consideration of time efficiency. The comparison of required simulation time between the methodologies of hybrid experiment and the pure simulation-based fault injection approach implemented in *CoWare Platform Architect* will be given later.

The Type 1 experimental process was carried out to estimate $P_{p\text{-}b\text{-}err}$, where N_{bench}, S_m and $N_{p\text{-}b\text{-}err}$ were set as the values of 4, 524288, and 500 respectively. Table 3 shows the propagation probability of bit errors for four benchmarks, which were derived from a huge amount of fault injection campaigns to guarantee their statistical validity. It is evident that the propagation probability is benchmark-variant and a bit error in memory would have the probability between 0.866% and 3.551% to propagate the bit error from memory to system. The results imply that most of the bit errors won't cause damage to the system. We should emphasize that the size of memory space and characteristics of the used benchmarks (such as amount of memory space use and amount of memory read/write) will affect the result of $P_{p\text{-}b\text{-}err}$. Therefore, the data in Table 3 reflect the results for the selected memory space and benchmarks.

Type 2 experiment: From Type 1 experimental process, we collect $N_{p\text{-}b\text{-}err}$ bit errors for each benchmark to the set $S_{p\text{-}b\text{-}err}(j)$. Those propagated bit errors were used to assess the failure probability of propagated bit errors. Therefore, $N_{p\text{-}b\text{-}err}$ simulation-based fault injection

Benchmark	N_{inj}	$N_{p\text{-}b\text{-}err}$	$P_{p\text{-}b\text{-}err}$
M-M	14079	500	3.551%
QS	23309	500	2.145%
JPEG	27410	500	1.824%
FFT	57716	500	0.866%

Table 3. Propagation probability of bit errors.

campaigns were conducted under CoWare Platform Architect, and each injection campaign injects a bit error into the memory according to the error scenarios recorded in the set $S_{p\text{-}b\text{-}err}(j)$. Therefore, we can examine the *SoC* behavior for each injected bit error.

As can be seen from Table 3, we need to conduct an enormous amount of fault injection campaigns to reach the expected number of propagated bit errors. Without the use of Type 1 experiment, we need to utilize the simulation-based fault injection approach to assess the propagation probability and failure probability of bit errors as illustrated in Table 3, 5, and 6, which require a huge number of simulation-based fault injection campaigns to be conducted. As a result, an enormous amount of simulation time is required to complete the injection and simulation campaigns. Instead, we developed a software tool to implement the experimental process described in Type 1 experiment to quickly identify which situation the injected bit error will lead to. Using this approach, the number of simulation-based fault injection campaigns performed in Type 2 experiment decreases dramatically. The performance of software tool adopted in Type 1 experiment is higher than that of simulation-based fault injection campaign employed in Type 2 experiment. Therefore, we can save a considerable amount of simulation time.

The data of Table 3 indicate that without the help of Type 1 experiment, we need to carry out a few ten thousand simulation-based fault injection campaigns in Type 2 experiment. As opposite to that, with the assistance of Type 1 experiment, only five hundred injection campaigns are required in Type 2 experiment. Table 4 gives the experimental time of the Type 1 plus Type 2 approach and pure simulation-based fault injection approach, where the data in the column of ratio are calculated by the experimental time of Type 1 plus Type 2 approach divided by the experimental time of pure simulation-based approach. The experimental environment consists of four machines to speed up the validation, where each machine is equipped with Intel® Core™2 Quad Processor Q8400 CPU, 2G RAM, and CentOS 4.6. In the experiments of Type 1 plus Type 2 approach and pure simulation-based approach, each machine is responsible for performing the simulation task for one benchmark. According to the simulation results, the average execution time for one simulation-based fault injection experiment is 14.5 seconds. It is evident that the performance of Type 1 plus Type 2 approach is quite efficient compared to the pure simulation-based approach because Type 1 plus Type 2 approach employed a software tool to effectively reduce the number of simulation-based fault injection experiments to five hundred times compared to a few ten thousand simulation-based fault injection experiments for pure simulation-based approach.

Given $N_{p\text{-}b\text{-}err}$ and $S_{p\text{-}b\text{-}err}(j)$, i.e. five hundred simulation-based fault injection campaigns, the Type 2 experimental results are illustrated in Table 5. From Table 5, we can identify the potential failure modes and the distribution of failure modes for each benchmark. It is clear that the susceptibility of a system to the memory bit errors is benchmark-variant, and the M-

M is the most critical benchmark among the four adopted benchmarks, according to the results of Table 5.

We then manipulated the data of Table 3 and 5 to acquire the results of Table 6. Table 6 shows the probability distribution of failure modes if a bit error occurs in the memory sub-system. Each datum in the row of 'Avg.' was obtained by mathematical average of the benchmarks' data in the corresponding column. This table offers the following valuable information: the robustness of memory sub-system, the probability distribution of failure modes and the impact of benchmark on the *SoC* dependability. Probability of *SoC* failure for a bit error occurring in the memory is between 0.738% and 3.438%. We also found that the *SoC* has the highest probability to encounter the SDC failure mode for a memory bit error. In addition, the vulnerability rank of benchmarks for memory bit errors is M-M > QS > JPEG > FFT.

Table 7 illustrates the statistics of memory read/write for the adopted benchmarks. The results of Table 7 confirm the vulnerability rank of benchmarks as observed in Table 6. Situation 2 as mentioned in the beginning of this section indicates that the occurring probability of Situation 2 increases as the probability of performing the memory write operation increases. Consequently, the robustness of a benchmark rises with an increase in the probability of Situation 2.

Benchmark	Type 1 + 2 (minute)	Pure approach (minute)	Ratio
M-M	312	1525	20.46%
QS	835	2719	30.71%
JPEG	7596	15760	48.20%
FFT	3257	9619	33.86%

Table 4. Comparison of experimental time between type 1 + 2 & pure simulation-based approach.

Benchmark	FF	SDC	CD/IT	IL	NE
M-M	0	484	0	0	16
QS	0	138	103	99	160
JPEG	0	241	1	126	132
FFT	0	177	93	156	74

Table 5. Type 2 experimental results.

	FF (%)	SDC (%)	CD/IT (%)	IL (%)	SF (%)	NE (%)
M-M	0.0	3.438	0.0	0.0	3.438	96.562
QS	0.0	0.592	0.442	0.425	1.459	98.541
JPEG	0.0	0.879	0.004	0.460	1.343	98.657
FFT	0.0	0.307	0.161	0.270	0.738	99.262
Avg.	0.0	1.304	0.152	0.289	1.745	98.255

Table 6. *P (2, FM(K))*, *P (2, SF)* and *P (2, NE)* for the used benchmarks.

	#R/W	#R	R(%)	#W	W(%)
M-M	265135	255026	96.187%	10110	3.813%
QS	226580	196554	86.748%	30027	13.252%
JPEG	1862291	1436535	77.138%	425758	22.862%
FFT	467582	240752	50.495%	236030	49.505%

Table 7. The statistics of memory read/write for the used benchmarks.

5.3 Register file experimental results

The ARM926EJ CPU used in the experimental *SoC* platform is an IP provided from *CoWare Platform Architect*. Therefore, the proposed simulation-based fault injection approach has a limitation to inject the faults into the register file inside the CPU. This problem can be solved by software-implemented fault injection methodology as described in Section 4. Currently, we cannot perform the fault injection campaigns in register file under *CoWare Platform Architect* due to lack of the operating system support. We note that the literature [Leveugle et al., 2009; Bergaoui et al., 2010] have pointed out that the register file is vulnerable to the radiation-induced soft errors. Therefore, we think the register file should be taken into account in the vulnerability analysis and risk assessment. Once the critical registers are located, the SEU-resilient flip-flop and register design can be exploited to harden the register file. In this experiment, we employed a similar physical system platform built by ARM926EJ-embedded *SoC* running Linux operating system 2.6.19 to derive the experimental results for register file.

The register set in ARM926EJ CPU used in this experiment is R0 ~ R12, R13 (SP), R14 (LR), R15 (PC), R16 (CPSR), and R17 (ORIG_R0). A fault injection campaign injects a single bit-flip fault to the target register to investigate its effect on the system behavior. For each benchmark, we performed one thousand fault injection campaigns for each target register by randomly choosing the time instant of fault injection within the benchmark simulation duration, and randomly choosing the target bit to inject 1-bit flip fault. So, eighteen thousand fault injection campaigns were carried out for each benchmark to obtain the data shown in Table 8. From Table 8, it is evident that the susceptibility of the system to register faults is benchmark-dependent and the rank of system vulnerability over different benchmarks is QS > FFT > M-M. However, all benchmarks exhibit the same trend in that while a fault arises in the register set, the occurring probabilities of CD/IT and FF occupy the top two ranks. The robustness measure of the register file is around 74% as shown in Table 8, which means that a fault occurring in the register file, the *SoC* has the probability of 74% to survive for that fault.

	FF (%)	SDC (%)	CD/IT (%)	IL (%)	SF (%)	NE (%)
M-M	6.94	1.71	10.41	0.05	19.11	80.89
FFT	8.63	1.93	15.25	0.04	25.86	74.14
QS	5.68	0.97	23.44	0.51	30.59	69.41
Avg.	7.08	1.54	16.36	0.2	25.19	74.81

Table 8. *P (3, FM(K))*, *P (3, SF)* and *P (3, NE)* for the used benchmarks.

REG #	*SoC* failure probability			REG #	*SoC* failure probability		
	M-M (%)	FFT (%)	QS (%)		M-M (%)	FFT (%)	QS (%)
R0	7.9	13.0	5.6	R9	12.4	7.3	20.6
R1	31.1	18.3	19.8	R10	23.2	32.5	19.9
R2	19.7	14.6	19.2	R11	37.5	25.3	19.2
R3	18.6	17.0	15.4	R12	22.6	13.1	25.3
R4	4.3	12.8	21.3	R13	34.0	39.0	20.3
R5	4.0	15.2	20.4	R14	5.1	100.0	100.0
R6	7.4	8.8	21.6	R15	100.0	100.0	100.0
R7	5.0	14.6	23.9	R16	3.6	8.3	49.4
R8	4.0	9.7	24.7	R17	3.6	15.9	24.0

Table 9. Statistics of *SoC* failure probability for each target register with various benchmarks.

Table 9 illustrates the statistics of *SoC* failure probability for each target register under the used benchmarks. Throughout this table, we can observe the vulnerability of each register for different benchmarks. It is evident that the vulnerability of registers quite depends on the characteristics of the benchmarks, which could affect the read/write frequency and read/write syndrome of the target registers. The bit errors won't cause damage to the system operation if one of the following situations occurs:

- Situation 1: The benchmark never uses the affected registers after the bit errors happen.
- Situation 2: The first access to the affected registers after the occurrence of bit errors is the 'write' action.

It is apparent to see that the utilization and read frequency of R4 ~ R8 and R14 for benchmark M-M is quite lower than FFT and QS, so the *SoC* failure probability caused by the errors happening in R4 ~ R8 and R14 for M-M is significantly lower than FFT and QS as illustrated in Table 9. We observe that the usage and write frequency of registers, which reflects the features and the programming styles of benchmark, dominates the soft error sensitivity of the registers. Without a doubt, the susceptibility of register R15 (program counter) to the faults is 100%. It indicates that the R15 is the most vulnerable register to be protected in the register set. Fig. 2 illustrates the average *SoC* failure probabilities for the registers R0 ~ R17, which are derived from the data of the used benchmarks as exhibited in Table 9. According to Fig. 2, the top three vulnerable registers are R15 (100%), R14 (68.4%), as well as R13 (31.1%), and the *SoC* failure probabilities for other registers are all below 30%.

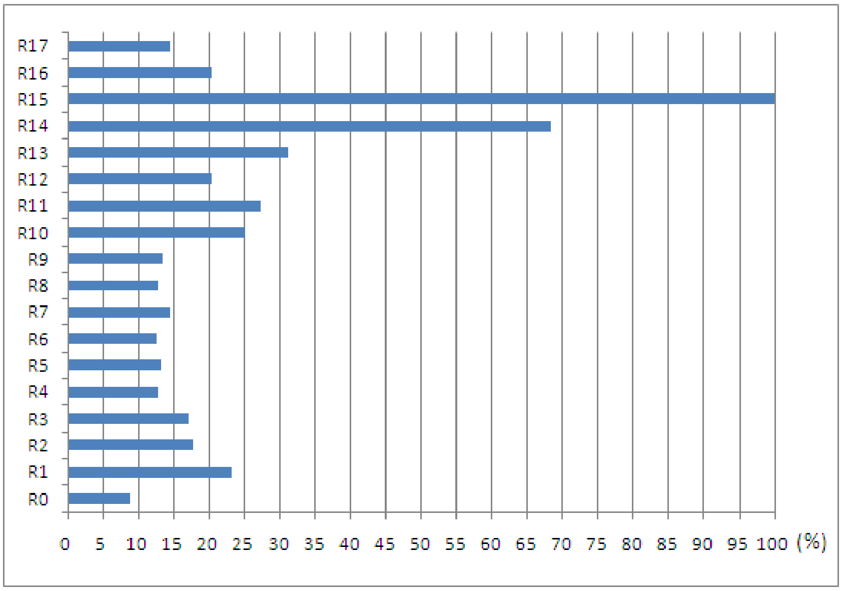


Fig. 2. The average *SoC* failure probability from the data of the used benchmarks.

5.4 SoC-level vulnerability analysis and risk assessment

According to IEC 61508, if a failure will result in a *critical effect* on system and lead human's life to be in danger, then such a failure is identified as a *dangerous failure or hazard*. IEC 61508 defines a system's safety integrity level (SIL) to be the Probability of the occurrence of a dangerous Failure per Hour (PFH) in the system. For continuous mode of operation (high demand rate), the four levels of SIL are given in Table 10 [IEC, 1998-2000].

SIL	PFH
4	$\geq 10^{-9}$ to $<10^{-8}$
3	$\geq 10^{-8}$ to $<10^{-7}$
2	$\geq 10^{-7}$ to $<10^{-6}$
1	$\geq 10^{-6}$ to $<10^{-5}$

Table 10. Safety integrity levels.

In this case study, three components, ARM926EJ CPU, AMBA AHB system bus and memory sub-system, were utilized to demonstrate the proposed risk model to assess the scales of failure-induced risks in a system. The following data are used to show the vulnerability analysis and risk assessment for the selected components {*C(1), C(2), C(3)*} = {AMBA AHB, memory sub-system, register file in ARM926EJ}: {*ER_C(1), ER_C(2), ER_C(3)*} = {10^{-6} ~ 10^{-8}/hour }; {*SR_FM(1), SR_FM(2), SR_FM(3), SR_FM(4)*} = {10, 8, 4, 6}. According to the expressions presented in Section 3 and the results shown in Section 5.1 to 5.3, the *SoC* failure rate, *SIL* and *RPN* are obtained and illustrated in Table 11, 12 and 13.

ER_C/hour	1×10^{-6}	0.5×10^{-6}	1×10^{-7}	0.5×10^{-7}	1×10^{-8}
SFR_C(1)	7.32×10^{-7}	3.66×10^{-7}	7.32×10^{-8}	3.66×10^{-8}	7.32×10^{-9}
SFR_C(2)	1.75×10^{-8}	8.73×10^{-9}	1.75×10^{-9}	8.73×10^{-10}	1.75×10^{-10}
SFR_C(3)	2.52×10^{-7}	1.26×10^{-7}	2.52×10^{-8}	1.26×10^{-8}	2.52×10^{-9}
SFR	1.0×10^{-6}	5.0×10^{-7}	1.0×10^{-7}	5.0×10^{-8}	1.0×10^{-8}
SIL	1	2	2	3	3

Table 11. *SoC* failure rate and SIL.

ER_C/hour	1×10^{-6}	0.5×10^{-6}	1×10^{-7}	0.5×10^{-7}	1×10^{-8}
RPN_C(1)	5.68×10^{-6}	2.84×10^{-6}	5.68×10^{-7}	2.84×10^{-7}	5.68×10^{-8}
RPN_C(2)	1.28×10^{-7}	6.38×10^{-8}	1.28×10^{-8}	6.38×10^{-9}	1.28×10^{-9}
RPN_C(3)	1.5×10^{-6}	7.49×10^{-7}	1.5×10^{-7}	7.49×10^{-8}	1.5×10^{-8}

Table 12. Risk priority number for the target components.

ER_C/hour	1×10^{-6}	0.5×10^{-6}	1×10^{-7}	0.5×10^{-7}	1×10^{-8}
RPN_FM(1)	2.65×10^{-6}	1.32×10^{-6}	2.65×10^{-7}	1.32×10^{-7}	2.65×10^{-8}
RPN_FM(2)	3.28×10^{-6}	1.64×10^{-6}	3.28×10^{-7}	1.64×10^{-7}	3.28×10^{-8}
RPN_FM(3)	9.64×10^{-7}	4.82×10^{-7}	9.64×10^{-8}	4.82×10^{-8}	9.64×10^{-9}
RPN_FM(4)	5.13×10^{-7}	2.56×10^{-7}	5.13×10^{-8}	2.56×10^{-8}	5.13×10^{-9}

Table 13. Risk priority number for the potential failure modes.

We should note that the components' error rates used in this case study are only for the demonstration of the proposed robustness/safety validation process, and the more realistic components' error rates for the considered components should be determined by process and circuit technology [Mukherjee et al., 2003]. According to the given components' error rates, the data of SFR in Table 11 can be used to assess the safety integrity level of the system. One thing should be pointed out that a *SoC* failure may or may not cause the dangerous effect on the system and human life. Consequently, a *SoC* failure could be classified into safe failure or dangerous failure. To simplify the demonstration, we make an assumption in this assessment that the *SoC* failures caused by the faults occurring in the components are always the dangerous failures or hazards. Therefore, the SFR in Table 11 is used to approximate the PFH, and so the SIL can be derived from Table 10.

With respect to safety design process, if the current design does not meet the SIL requirement, we need to perform the risk reduction procedure to lower the PFH, and in the meantime to reach the SIL requirement. The vulnerability analysis and risk assessment can be exploited to identify the most critical components and failure modes to be protected. In such approach, the system safety can be improved efficiently and economically.

Based on the results of *RPN_C(i)* as exhibited in Table 12, for *i* = *1, 2, 3*, it is evident that the error of AMBA AHB is more critical than the errors of register set and memory sub-system. So, the results suggest that the AHB system bus is more urgent to be protected than the register set and memory. Moreover, the data of *RPN_FM(k)* in Table 13, *k* from one to four, infer that SDC is the most crucial failure mode in this illustrated example. Throughout the above vulnerability and risk analyses, we can identify the critical components and failure modes, which are the major targets for design enhancement. In this demonstration, the top priority of the design enhancement is to raise the robustness of the AHB HADDR bus signals to significantly reduce the rate of SDC and the scale of system risk if the system reliability/safety is not adequate.

6. Conclusion

Validating the functional safety of system-on-chip (*SoC*) in compliance with international standard, such as IEC 61508, is imperative to guarantee the dependability of the systems before they are being put to use. It is beneficial to assess the *SoC* robustness in early design phase in order to significantly reduce the cost and time of re-design. To fulfill such needs, in this study, we have presented a valuable *SoC*-level safety validation and risk reduction process to perform the hazard analysis and risk assessment, and exploited an ARM-based *SoC* platform to demonstrate its feasibility and usefulness. The main contributions of this study are first to develop a useful SVRR process and risk model to assess the scales of robustness and failure-induced risks in a system; second to raise the level of dependability validation to the untimed/timed functional TLM, and to construct a *SoC*-level system safety verification platform including an automatic fault injection and failure mode classification tool on the SystemC *CoWare Platform Architect* design environment to demonstrate the core idea of SVRR process. So the efficiency of the validation process is dramatically increased; third to conduct a thorough vulnerability analysis and risk assessment of the register set, AMBA bus and memory sub-system based on a real ARM-embedded *SoC*.

The analyses help us measure the robustness of the target components and system safety, and locate the critical components and failure modes to be guarded. Such results can be used to examine whether the safety of investigated system meets the safety requirement or not, and if not, the most critical components and failure modes are protected by some effective risk reduction approaches to enhance the safety of the investigated system. The vulnerability analysis gives a guideline for prioritized use of robust components. Therefore, the resources can be invested in the right place, and the fault-robust design can quickly achieve the safety goal with less cost, die area, performance and power impact.

7. Acknowledgment

The author acknowledges the support of the National Science Council, R.O.C., under Contract No. NSC 97-2221-E-216-018 and NSC 98-2221-E-305-010. Thanks are also due to the

National Chip Implementation Center, R.O.C., for the support of SystemC design tool – *CoWare Platform Architect*.

8. References

Austin, T. (1999). DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design, *Proceedings of 32nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 196-207, ISBN 076950437X, Haifa, Israel, Nov. 1999

Baumann, R. (2005). Soft Errors in Advanced Computer Systems. *IEEE Design & Test of Computers*, Vol. 22, No. 3, (May-June 2005), pp. (258 – 266), ISSN 0740-7475

Bergaoui, S.; Vanhauwaert, P. & Leveugle, R. (2010) A New Critical Variable Analysis in Processor-Based Systems. *IEEE Transactions on Nuclear Science*, Vol. 57, No. 4, (August 2010), pp. (1992-1999), ISSN 0018-9499

Brown, S. (2000). Overview of IEC 61508 Design of electrical/electronic/programmable electronic safety-related systems. *Computing & Control Engineering Journal*, Vol. 11, No. 1, (February 2000), pp. (6-12), ISSN 0956-3385

International Electrotechnical Commission [IEC], (1998-2000). CEI International Standard IEC 61508, 1998-2000

Chang, K. & Chen, Y. (2007). System-Level Fault Injection in SystemC Design Platform, *Proceedings of 8th International Symposium on Advanced Intelligent Systems*, pp. 354-359, Sokcho-City, Korea, Sept. 05-08, 2007

Chen, Y.; Wang, Y. & Peng, J. (2008). SoC-Level Fault Injection Methodology in SystemC Design Platform, *Proceedings of 7th International Conference on System Simulation and Scientific Computing*, pp. 680-687, Beijing, China, Oct. 10-12, 2008

Constantinescu, C. (2002). Impact of Deep Submicron Technology on Dependability of VLSI Circuits, *Proceedings of IEEE International Conference on Dependable Systems and Networks*, pp. 205-209, ISBN 0-7695-1597-5, Bethesda, MD, USA, June 23-26, 2002

CoWare, (2006). Platform Creator User's Guide, IN: CoWare Model Library Product Version V2006.1.2

Grotker, T.; Liao, S.; martin, G. & Swan, S. (2002). *System Design with SystemC*, Kluwer Academic Publishers, ISBN 978-1-4419-5285-1, Boston, Massachusetts, USA

Hosseinabady, M.; Neishaburi, M.; Lotfi-Kamran P. & Navabi, Z. (2007). A UML Based System Level Failure Rate Assessment Technique for SoC Designs, *Proceedings of 25th IEEE VLSI Test Symposium*, pp. 243 – 248, ISBN 0-7695-2812-0, Berkeley, California, USA, May 6-10, 2007

Kanawati, G.; Kanawati, N. & Abraham, J. (1995). FERRARI: A Flexible Software-Based Fault and Error Injection System. *IEEE Transactions on Computers*, Vol. 44, No. 2, (Feb. 1995), pp. (248-260), ISSN 0018-9340

Karnik, T.; Hazucha, P. & Patel, J. (2004). Characterization of Soft Errors Caused by Single Event Upsets in CMOS Processes. *IEEE Transactions on Dependable and Secure Computing*, Vol. 1, No. 2, (April-June 2004), pp. (128-143), ISSN 1545-5971

Kim, S. & Somani, A. (2002). Soft Error Sensitivity Characterization for Microprocessor Dependability Enhancement Strategy, *Proceedings of IEEE International Conference on Dependable Systems and Networks*, pp. 416-425, ISBN 0-7695-1597-5, Bethesda, MD, USA, June 23-26, 2002

Leveugle, R.; Pierre, L.; Maistri, P. & Clavel, R. (2009). Soft Error Effect and Register Criticality Evaluations: Past, Present and Future, *Proceedings of IEEE Workshop on*

Silicon Errors in Logic - System Effects, pp. 1-6, Stanford University, California, USA, March 24-25, 2009

Mariani, R.; Boschi, G. & Colucci, F. (2007). Using an innovative *SoC*-level FMEA methodology to design in compliance with IEC61508, *Proceedings of 2007 Design, Automation & Test in Europe Conference & Exhibition*, pp. 492-497, ISBN 9783981080124, Nice, France, April 16-20, 2007

Mikulak, R.; McDermott, R. & Beauregard, M. (2008). *The Basics of FMEA* (Second Edition), CRC Press, ISBN 1563273772, New York, NY, USA

Mitra, S.; Seifert, N.; Zhang, M.; Shi, Q. & Kim, K. (2005). Robust System Design with Built-in Soft-Error Resilience. *IEEE Computer*, Vol. 38, No. 2, (Feb. 2005), pp. 43-52, ISSN 0018-9162

Mollah, A. (2005). Application of Failure Mode and Effect Analysis (FMEA) for Process Risk Assessment. *BioProcess International*, Vol. 3, No. 10, (November 2005), pp. (12–20)

Mukherjee, S.; Weaver, C.; Emer, J.; Reinhardt, S. & Austin, T. (2003). A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High Performance Microprocessor, *Proceedings of 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 29-40, ISBN 0-7695-2043-X, San Diego, California, USA, Dec. 03-05, 2003

Open SystemC Initiative (OSCI), (2003). SystemC 2.0.1 Language Reference Manual (Revision 1.0), IN: *Open SystemC Initiative*, Available from: < homes.dsi.unimi.it/~pedersin/AD/SystemC_v201_LRM.pdf>

Rotenberg, E. (1999). AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessor, *Proceedings of 29th Annual IEEE International Symposium on Fault-Tolerant Computing,* pp. 84-91, ISBN 076950213X, Madison , WI, USA, 1999

Ruiz, J.; Yuste, P.; Gil, P. & Lemus, L. (2004). On Benchmarking the Dependability of Automotive Engine Control Applications, *Proceedings of IEEE International Conference on Dependable Systems and Networks,* pp. 857 – 866, ISBN 0-7695-2052-9, Palazzo dei Congressi, Florence, Italy, June 28 – July 01, 2004

Sieh, V. (1993). Fault-Injector using UNIX ptrace Interface, IN: *Internal Report No.: 11/93, IMMD3, Universität Erlangen-Nürnberg,* Available from: < http://www3.informatik.uni-erlangen.de/Publications/Reports/ir_11_93.pdf>

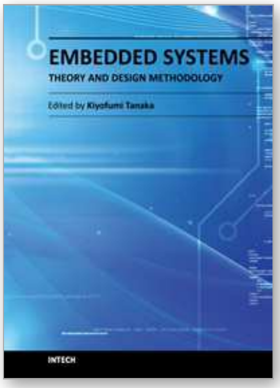
Slegel, T. et al. (1999). IBM's S/390 G5 Microprocessor Design. *IEEE Micro*, Vol. 19, No. 2, (March/April, 1999), pp. (12-23), ISSN 0272-1732

Stamatelatos, M.; Vesely, W.; Dugan, J.; Fragola, J.; Minarick III, J. & Railsback, J. (2002). Fault Tree Handbook with Aerospace Applications (version 1.1), IN: *NASA*, Available from: <www.hq.nasa.gov/office/codeq/doctree/fthb.pdf>

Tony, S.; Mohammad, H.; Mathew, J. & Pradhan, D. (2007). Soft-Error induced System-Failure Rate Analysis in an SoC, *Proceedings of 25th Norchip Conf.*, pp. 1-4, Aalborg, DK, Nov. 19-20, 2007

Wang, N.; Quek, J.; Rafacz, T. & Patel, S. (2004). Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline, *Proceedings of IEEE International Conference on Dependable Systems and Networks*, pp. 61-70, ISBN 0-7695-2052-9, Palazzo dei Congressi, Florence, Italy, June 28 – July 01, 2004

Zorian, Y.; Vardanian, V.; Aleksanyan, K. & Amirkhanyan, K. (2005). Impact of Soft Error Challenge on SoC Design, *Proceedings of 11th IEEE International On-Line Testing Symposium,* pp. 63 – 68, ISBN 0-7695-2406-0, Saint Raphael, French Riviera, France, July 06-08, 2005



Embedded Systems - Theory and Design Methodology
Edited by Dr. Kiyofumi Tanaka

ISBN 978-953-51-0167-3
Hard cover, 430 pages
Publisher InTech
Published online 02, March, 2012
Published in print edition March, 2012

How to reference
In order to correctly reference this scholarly work, feel free to copy and paste the following:

Yung-Yuan Chen and Tong-Ying Juang (2012). Vulnerability Analysis and Risk Assessment for SoCs Used in Safety-Critical Embedded Systems, Embedded Systems - Theory and Design Methodology, Dr. Kiyofumi Tanaka (Ed.), ISBN: 978-953-51-0167-3, InTech, Available from: http://www.intechopen.com/books/embedded-systems-theory-and-design-methodology/vulnerability-analysis-and-risk-assessment-for-socs-used-in-safety-critical-embedded-systems