行政院國家科學委員會專題研究計畫 成果報告

行為層高效能處理器的容錯設計及快速驗證與容錯能力分析(I)
研究成果報告(精簡版)

計畫類別：個別型
計畫編號：NSC 95-2221-E-216-015-
執行期間：95年08月01日至96年07月31日
執行單位：中華大學資訊工程學系

計畫主持人：陳永源

計畫參與人員：碩士班研究生-兼任助理：張坤鈞、石孟儒、吳耿偉

報告附件：出席國際會議研究心得報告及發表論文

處理方式：本計畫可公開查詢

中華民國 96年10月31日

The following report contains two parts: The first part is based on the fault injection approach to verify the capability of a fault-tolerant system. In this part, we propose a system-level simulation-based fault injection framework in SystemC design platform to assist the dependability assessment. The second part is going to discuss how to analyze the error coverage without using the fault injection and fault simulation mechanisms in order to save the development efforts and simulation time. Our idea is to devise a high-level abstract model to represent the fault-tolerant systems including the interconnection structure of the functional blocks, the propagation tables expressing the relationship between inputs and outputs for each functional block and the Petri Net to model the functional behavior of the fault-tolerant systems.

Abstract (first part)

This report describes the results achieved in the first year of three-year research proposal. As mentioned in the proposal, an important issue in the design of *SoC* with fault tolerance is how to verify the feasibility of the fault-robust design as early in the development phase to reduce the re-design cost. Therefore, a system-level fault-tolerant verification platform is required to assist the designers in assessing the dependability of a system with an efficient manner. The first part of this study is to propose a system-level simulation-based fault injection framework in SystemC design platform to assist the dependability assessment. The proposed fault injection framework is able to inject the faults into the systems modeled at the following levels of abstraction: register-transfer level (RTL)/bus-cycle accurate level, untimed functional transaction-level modeling with primitive channel sc_fifo, and timed functional transaction-level modeling with hierarchical channel. We devise a distributed injection control approach instead of using one centralized control unit to control the injection activity. The proposed distributed injection control approach, which consists of the time-triggered and event-triggered injection technologies, is capable of injecting single or multiple faults with diverse fault types into different abstraction levels. We demonstrate the feasibility of the proposed fault injection framework with the modules modeled at different levels of abstraction.

Another work of this study is to characterize the dependability of fault-tolerant systems modeled at different hardware design environments, SystemC and VHDL. For SystemC, we inject *errors* into the components' outputs, whereas *faults* into the inside of components for VHDL. The difference of the simulation results between SystemC and VHDL is discussed thoroughly through observing two parameters: one is the probability of a fault causing an effective error and another is the relationship between fault duration and error duration. The above two parameters dominate the discrepancy as seen between the two different design platforms. The experimental results show the effect of the fault attributes on the error coverage. This study can promote the fault-tolerant design and verification environment to a higher abstraction level. The preliminary results can be found in the appendix.

Keywords: fault-tolerant design, high-level abstraction modeling, high-level rapid verification, SystemC, system-level fault injection, system-on-chip (*SoC*), transient fault (soft error or SEU).

I. INTRODUCTION

As system-on-chip (*SoC*) becomes more and more complicated, and contains a large number of transistors, 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 [1-3]. Thus, it is essential to employ the fault-tolerant techniques in the design of *SoC* to guarantee a high operational reliability in critical applications. However, due to the high complexity of the *SoC*, the incorporation of the 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 [4]. An important issue in the design of *SoC* with fault tolerance is how to validate the feasibility of the fault-robust design as early in the development phase to reduce the re-design cost. As a result, a system-level fault-tolerant verification platform is required to assist the designers in assessing the dependability of a system with an efficient manner. Normally, the fault injection approach is employed to verify the robustness of a fault-tolerant system.

SystemC [5], a system-level modeling language, provides a wide variety of modeling levels of abstraction and allows us to model a system utilizing one or a mixture of various abstraction levels. It is quite common that the modules within a fault-tolerant *SoC* are modeled at different levels of abstraction using SystemC design language. Therefore, the fault injection framework for SystemC design platform must offer the methodologies to inject the faults into the different modeling levels.

Most of the previous fault injection studies focus on the VHDL design platform, whereas only a few works [6-9] address the fault injection issue in SystemC design platform. In our previous paper [9], we proposed a fault injection methodology for cycle-accurate register-transfer level (RTL) and compared the results of injection campaigns with the outcomes derived from the VHDL RTL. The comparisons show the accuracy and feasibility of our approach. However, the scheme presented in [9] can only apply to RTL, which limits the scope of application. In [6, 7], the authors proposed a fault injection framework that is applicable to functional level and transaction layer 1 in SystemC [10]. The mechanism presented in [6, 7] is based on the insertion of fault injection modules (FIMs) into the interconnections of the functional blocks, where a FIM is to control the fault injection activity for a selected fault target. The injection activity of a fault can be characterized by the following attributes: time instant, fault type/value, and fault duration. A centralized fault injection control unit is used to control the FIMs. So, the centralized control unit is responsible for the determination when to inject a fault into which target and for what fault value and duration. Once the centralized control unit decides to inject a fault, the related control signals are sent to the designated FIM. The merit of [6, 7] scheme is no need to modify the SystemC source code for each fault injection campaign once the FIMs have been inserted into the simulation model. The only source to be prepared for each injection campaign is the fault injection controller that implements the injection script commands.

Several interesting issues deserved to be explored further are described as follows. One is the control complexity of the centralized injection control methodology and its effect on the simulation time. As system is getting more complex, the injection control complexity rises too. Consequently, the simulation performance could be degraded significantly. Another is how to inject the faults into the systems modeled using mixed levels of abstraction. Since it is rare that all modules within a system are modeled at the same level of abstraction, the injection approach developed should possess the ability to inject the faults into different levels of abstraction. Third is how to generate the fault injection script file at higher levels of abstraction in SystemC design platform, which represents the fault scenario for each injection campaign. The last issue is the feasibility of fault/error model employed at high level of abstraction. The precision of fault/error model will affect the accuracy of the results of injection campaign. Paper [9] presents a preliminary study of this fault/error modeling issue.

In this work, we propose an effective system-level simulation-based fault injection framework in SystemC design platform to assist the dependability assessment. The framework of fault injection proposed consists of the following modeling levels of abstraction: bus-cycle accurate (BCA) level, untimed functional transaction-level modeling (TLM) with primitive channel sc_fifo, and timed functional transaction-level modeling with hierarchical channel. We devise a distributed injection control approach instead of using one centralized control unit to control the injection activity. The proposed distributed injection control approach is capable of injecting single or multiple faults with diverse fault types into different abstraction levels. Our scheme can inject the faults into a system modeled at mixed levels of abstraction in SystemC. As we see, the control of injection activity is distributed to the fault injection modules (FIMs), which may lower the control complexity of fault injection and the simulation time compared to the centralized control approach. However, our approach needs to construct the source code of SystemC simulation model for each fault injection campaign because the fault injection script commands are distributively implemented in each FIM. The comparison of our distributed approach with the centralized control method in terms of experiment setup, compiling time, fault injection efficiency and simulation time will be discussed in the future.

The remaining report is organized as follows. In Section 2, the fault injection framework is presented. We demonstrate the feasibility of our fault injection approach in Section 3. The conclusions and future work appear in Section 4.

II. FAULT INJECTION FRAMEWORK IN SYSTEMC

In this section, we consider the fault injection into the communication channels at the following abstraction levels. The first one is sc_signal at BCA level; the second one is the primitive channel sc_fifo at untimed functional transaction level and the last on is the hierarchical channel [5, 10] at timed functional transaction level. The principal idea of our approach is based on the insertion of FIMs into the interconnections of the functional blocks, where a FIM is to control the fault injection activity for the selected fault target. Since we distribute the injection control to each FIM, the FIMs are responsible for the determination of the fault injection activity including when to inject a fault, what the fault value and its duration. The core of the FIM design is how to decide when to activate a fault injection.

A. *Fault injection at BCA level*

Fig. 1 shows the fault injection structure for BCA level that includes a FIM used for the control of fault injection. The FIM contains an injection list of the faults, which depicts the injection activity for each fault collected in the injection list. Since the BCA level is clock-cycle accurate, the FIM can use the sc_simulation_time ( ) to get the time instant of the beginning of each clock cycle. Then, the FIM checks the current time instant obtained with the injection time list of the faults pre-specified in the FIM. If the time instant is equal to the injection time of a fault, FIM will activate the fault injection by generating the desired fault type/value to the ‘MUX’ input port and keep the fault stayed active for a pre-defined length of time; otherwise, the original signal is delivered.

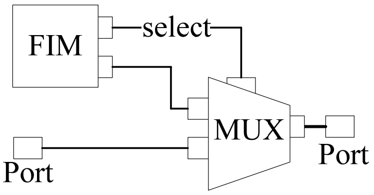
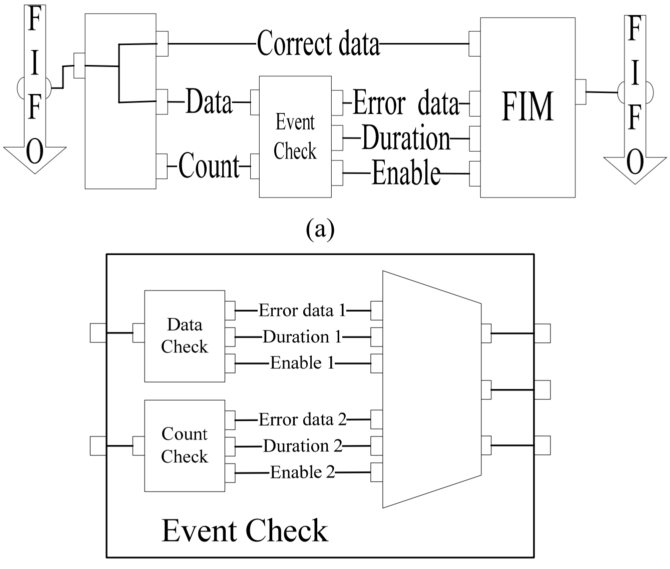


Fig. 1. Fault injection structure for BCA level.

B. *Fault injection at untimed functional transaction level*

As no clock exists in this level of abstraction, the event-driven method is utilized to trigger the FIM as illustrated in Fig. 2. An event is used to represent a condition that may occur during the course of simulation and to control the triggering of fault injection. We create the ‘Event Check’ module to monitor the occurrence of a specific event to control the FIM when to trigger the fault injection. The event could be, for example, a particular instruction address or a counter whose value reaches to a specific count. When the declared event occurs, the ‘Event Check’ module will send a trigger signal ‘Enable’ to FIM to activate the fault injection.

Fig. 2(b) exhibits the circuit diagram of ‘Event Check’ block. The ‘Data Check’ module can check the data-related events, such as a particular address and data. The ‘Count Check’ can check, for example, whether the number of data read out from the FIFO channel has reached to a specific count. Table I presents the operation of ‘Event Check’. ‘Event Check’ is expandable if more types of events need to check.



(b)

Fig. 2. (a) Fault injection structure for channel sc_fifo. (b) The circuit diagram of 'Event Check'

TABLE I THE OPERATION OF 'Event Check'.

Data Enable1	Count Enable 2	Data	Duration	Enable
0	0	0	0	0
0	1	Error data2	Duration2	Enable2
1	0	Error data1	Duration1	Enable1
1	1	Error data1	Duration1	Enable1

C. Fault injection at timed functional transaction level

Fig. 3 shows the transaction-level channel structure. Fault injection structure for transaction-level hierarchical channel is illustrated in Fig. 4. We note that a redundant channel is inserted between FIM and Slave modules. It is because the port of Slave module is the connection type for channel. To keep the original source code unchanged, we add one more channel to connect the FIM and Slave modules. This redundant channel is injection-induced component. Apparently, the input port and output port of FIM are slave port and master port for channel connection. Here, the concept of FIM is similar to that of sc_fifo.

In next section, we will exploit a popular hierarchical channel: AMBA bus to demonstrate the fault injection platform displayed in Fig. 4. We use the AMBA bus library [12] and AMBA bus API (Application Programming Interface) [13] provided by CoWare Platform Architect to implement the FIM as shown in Fig. 4.

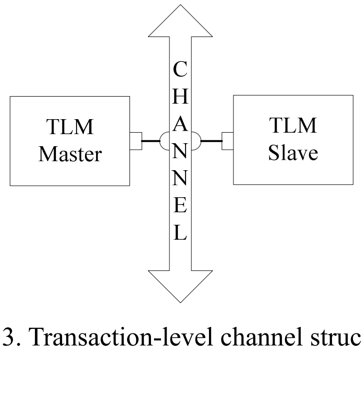


Fig. 3. Transaction-level channel structure.

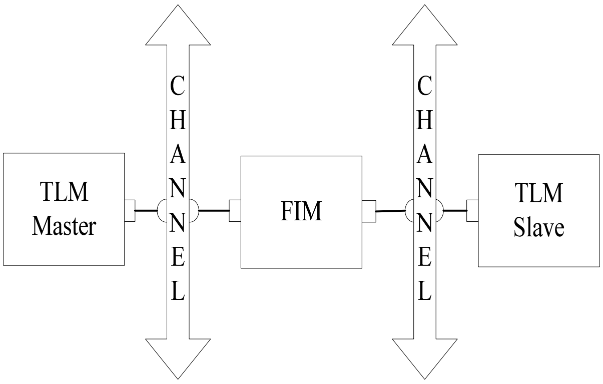


Fig. 4. Fault injection structure for transaction-level hierarchical channel.

III. FAULT INJECTION DEMONSTRATION

The following experimental studies were performed to validate the feasibility of our fault injection framework proposed in Section II. Fig. 5 shows a common circuit structure used in fault injection experiments. In Fig. 5, 'Driver' module is responsible for generating the augend and addend to the adder; 'Monitor' module is for printing out the results of adder. We employ the CoWare Platform Architect v2005.1.1 to build up the experimental environment.

A. Experiment at BCA level

Fig. 6 illustrates the fault injection structure of Fig. 5 circuit, where the modeling level of this experiment is BCA style. The clock cycle is 5 *ns* in this fault injection experiment. 'Driver' module sends out the augend and addend to the adder every 5 *ns*, where we assign zero to augend all the time and one to eight in sequence to addend. Two faults are injected into the augend through FIM at different simulation time. The first fault is injected at 15 *ns*. At that time, FIM delivers the fault value twelve to 'MUX' input port and sets 'select' signal to pass the fault value to the input port of 'Adder'. The duration of this fault is two clock cycles. The second fault is injected at 30 *ns* with fault value twenty and lasts one clock cycle. Fig. 7 presents the simulation results of fault-free and fault injection experiments. As can be seen from Fig. 7, it is evident that the first fault appears at time 15 *ns* and sustains two clock cycles. Another fault happens at time 30 *ns* and lasts one cycle. The injection capability of our approach at BCA level is justified from the results of fault injection campaign as shown in Fig. 7.

B. Experiment at untimed functional transaction level

Fig. 8 illustrates the fault injection structure at untimed functional transaction level. Table II lists a fault scenario for the injection campaign. From Table II, we see that the 'Count Check' will activate the fault injection while the 'Count' event , i.e. the number of augend data read out from the FIFO channel, has reached to a particular count, 1, 4 and 7, respectively. In this experiment, the augend values are (0, 2, 0, 4, 0, 6, 0, 8, 0, 10, 0, 12, 0, 14, 0, 16) and the addend values are (1, 0, 3, 0, 5, 0, 7, 0, 9, 0, 11, 0, 13, 0, 15, 0). Fig. 9 shows the results of the fault-free and fault injection experiments. From Fig. 9, it is easy to see that the first three faults are count-triggered faults and sustain one, two and three transactions, respectively. The last fault is data-triggered fault and maintains two transactions long. The simulation outcomes of Fig. 9 confirm the feasibility of event-triggered injection approach for untimed functional transaction level.

C. *Experiment at timed functional transaction level*

In this experiment, we employ the AMBA bus to demonstrate the injection of faults into the hierarchical channels modeled at timed functional transaction level. Since hierarchical channel plays an important role in *SoC*, offering the injection capability of faults into the hierarchical channels is imperative in SystemC design platform. We utilize the AMBA bus library [12] and AMBA bus API [13] furnished by CoWare Platform Architect to implement the circuit diagram shown in Fig. 10. The ‘Driver (master)’ and ‘Display (slave)’ modules in Fig. 10 are responsible for sending data to AHB and receiving plus printing data from AHB respectively. The concept of FIM module is like ‘AHB to AHB Bridge’ except FIM is able to pollute the bus data during the fault injection campaign. The count of data transaction in AHB is used in FIM to decide the injection time of faults. In this demonstration, two faults are injected into the AHB channel. The first fault occurs at the second data transaction and lasts the length of two data transactions. The second fault happens at the tenth data transaction and sustains one data transaction. The simulation results of fault-free and fault injection experiments are exhibited in Fig. 11.

D. *Experiment at mixed levels of abstraction*

SystemC, as a system-level design platform, employs the concepts of intellectual property (IP) reuse and hierarchical channel to reduce the *SoC* design complexity, effort and time. However, the provided IP modules may be modeled at various levels of abstraction such that a system is often modeled at mixed abstraction levels. Therefore, the inclusion of fault injection at mixed levels of abstraction is important in the development of system-level fault injection framework. The goal of this experiment is to show the feasibility of our fault injection framework, which is capable of injecting the faults into a system modeled at mixed levels of abstraction.

Fig. 12 demonstrates a fault injection structure at mixed levels of abstraction. In Fig. 12, ‘Driver_1’ module modeled at RTL provides the augend data for ‘Adder’; ‘Driver_2’ module modeled at timed functional transaction level offers the addend data through the AHB channel to ‘Adder’, and right part of Fig. 12 including ‘Adder’ and ‘Monitor’ modules is modeled at untimed functional transaction level with primitive channel sc_fifo. ‘Driver_1’ sends out an augend every 5 *ns* following the data sequence 0 ~ 14. ‘Driver_2’ sends out an addend every 10 *ns* following the data sequence 0 ~ 14. The ‘Adder’ module synchronizes the input sequences of augend and addend, and therefore, the results of ‘Adder’ are $2 \times i,\ i = 0 \text{ to } 14$. The fault scenario for this experiment is as follows: ‘FIM’ in ‘Driver_1’ part injects faults into augend at time 15*ns* and 25 *ns*; ‘FIM’ in ‘Driver_2’ part injects faults into addend at transaction count 6 and 9; ‘FIM’ in ‘Monitor’ part injects faults into ‘Monitor’ input at transaction count 11 and 12, and at ‘Adder’ output data equal to 28. Fig. 13 illustrates the simulation results of fault-free and fault injection experiments. As can be seen from Fig. 13(a), a situation of multiple faults occurs when augend is 55 and addend is 200. This confirms the multiple fault injection ability of our mechanism.

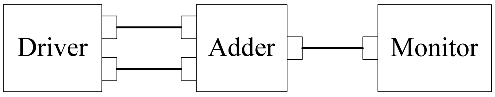


Fig. 5. A common circuit structure for injection experiments.

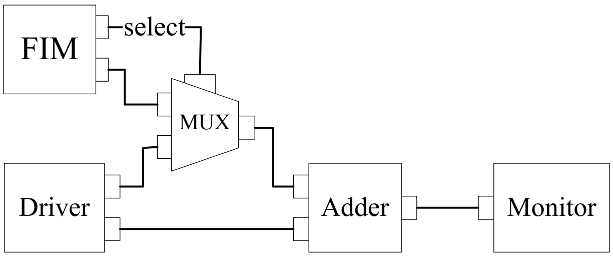
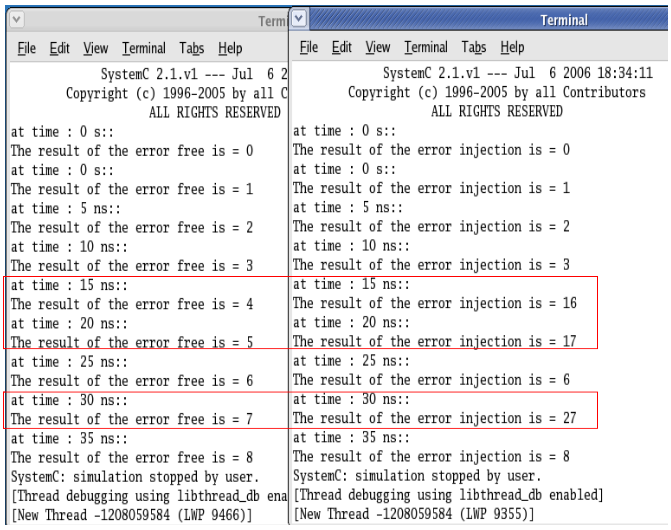


Fig. 6. Fault injection structure of Fig. 5 circuit modeled at BCA level.


Termi
Terminal
File Edit View Terminal Tabs Help
File Edit View Terminal Tabs Help

```
              SystemC 2.1.v1 --- Jul  6 2
          Copyright (c) 1996-2005 by all C
                       ALL RIGHTS RESERVED
at time : 0 s::
The result of the error free is = 0
at time : 0 s::
The result of the error free is = 1
at time : 5 ns::
The result of the error free is = 2
at time : 10 ns::
The result of the error free is = 3
at time : 15 ns::
The result of the error free is = 4
at time : 20 ns::
The result of the error free is = 5
at time : 25 ns::
The result of the error free is = 6
at time : 30 ns::
The result of the error free is = 7
at time : 35 ns::
The result of the error free is = 8
SystemC: simulation stopped by user.
[Thread debugging using libthread_db ena
[New Thread -1208059584 (LWP 9466)]
```

```
              SystemC 2.1.v1 --- Jul  6 2006 18:34:11
          Copyright (c) 1996-2005 by all Contributors
                       ALL RIGHTS RESERVED
at time : 0 s::
The result of the error injection is = 0
at time : 0 s::
The result of the error injection is = 1
at time : 5 ns::
The result of the error injection is = 2
at time : 10 ns::
The result of the error injection is = 3
at time : 15 ns::
The result of the error injection is = 16
at time : 20 ns::
The result of the error injection is = 17
at time : 25 ns::
The result of the error injection is = 6
at time : 30 ns::
The result of the error injection is = 27
at time : 35 ns::
The result of the error injection is = 8
SystemC: simulation stopped by user.
[Thread debugging using libthread_db enabled]
[New Thread -1208059584 (LWP 9355)]
```

Fig. 7. Simulation results of fault-free (left side) and fault injection (right side) experiments at BCA level.

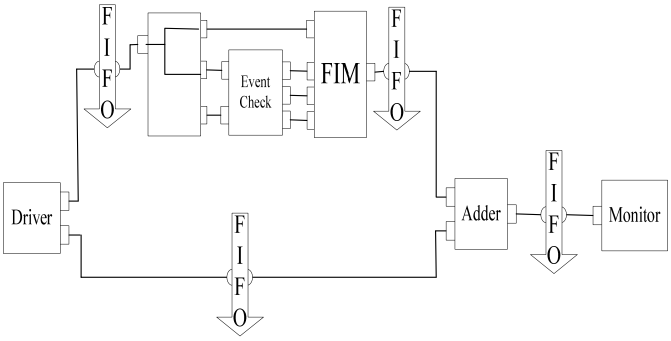
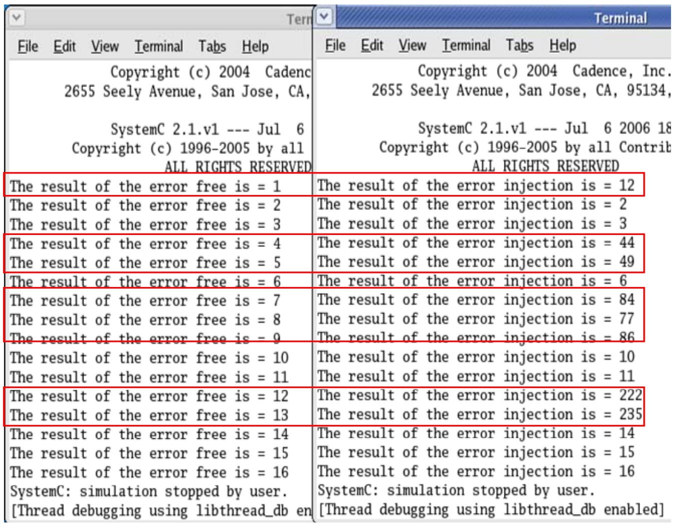


Fig. 8. Fault injection structure at untimed functional transaction level.

TABLE II FAULT EVENTS, VALUES AND DURATION

Count check	Fault value	Fault duration
1	11	1
4	44	2
7	77	3

Data check	Fault value	Fault duration
12	222	2



```
                 Copyright (c) 2004  Cadenc   |                 Copyright (c) 2004  Cadence, Inc.
          2655 Seely Avenue, San Jose, CA,    |          2655 Seely Avenue, San Jose, CA, 95134,
                                              |
                 SystemC 2.1.v1 --- Jul  6    |                 SystemC 2.1.v1 --- Jul  6 2006 18
          Copyright (c) 1996-2005 by all      |          Copyright (c) 1996-2005 by all Contrib
                          ALL RIGHTS RESERVED |                          ALL RIGHTS RESERVED
The result of the error free is = 1           | The result of the error injection is = 12
The result of the error free is = 2           | The result of the error injection is = 2
The result of the error free is = 3           | The result of the error injection is = 3
The result of the error free is = 4           | The result of the error injection is = 44
The result of the error free is = 5           | The result of the error injection is = 49
The result of the error free is = 6           | The result of the error injection is = 6
The result of the error free is = 7           | The result of the error injection is = 84
The result of the error free is = 8           | The result of the error injection is = 77
The result of the error free is = 9           | The result of the error injection is = 86
The result of the error free is = 10          | The result of the error injection is = 10
The result of the error free is = 11          | The result of the error injection is = 11
The result of the error free is = 12          | The result of the error injection is = 222
The result of the error free is = 13          | The result of the error injection is = 235
The result of the error free is = 14          | The result of the error injection is = 14
The result of the error free is = 15          | The result of the error injection is = 15
The result of the error free is = 16          | The result of the error injection is = 16
SystemC: simulation stopped by user.          | SystemC: simulation stopped by user.
[Thread debugging using libthread_db en       | [Thread debugging using libthread_db enabled]
```

Term
File Edit View Terminal Tabs Help
Terminal
File Edit View Terminal Tabs Help

Fig. 9. Simulation results of the fault-free and fault injection experiments at untimed functional transaction level.

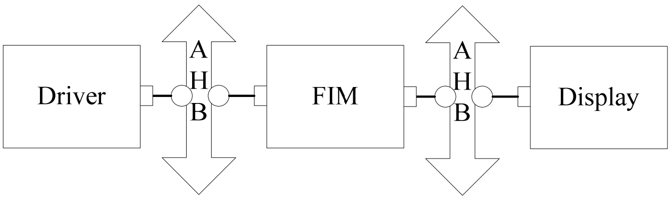
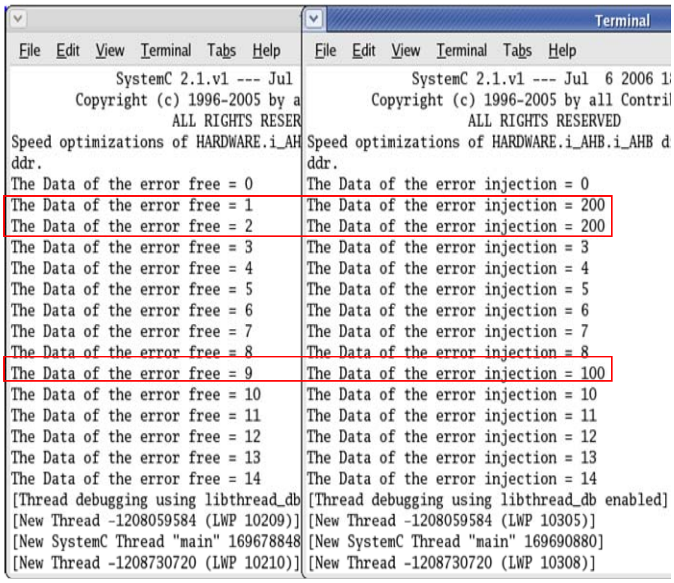


Fig. 10. Fault injection structure at timed functional transaction level.



```
            SystemC 2.1.v1 --- Jul   |            SystemC 2.1.v1 --- Jul  6 2006 1
       Copyright (c) 1996-2005 by a  |       Copyright (c) 1996-2005 by all Contri
                   ALL RIGHTS RESER  |                   ALL RIGHTS RESERVED
Speed optimizations of HARDWARE.i_AH | Speed optimizations of HARDWARE.i_AHB.i_AHB d
ddr.                                 | ddr.
The Data of the error free = 0       | The Data of the error injection = 0
The Data of the error free = 1       | The Data of the error injection = 200
The Data of the error free = 2       | The Data of the error injection = 200
The Data of the error free = 3       | The Data of the error injection = 3
The Data of the error free = 4       | The Data of the error injection = 4
The Data of the error free = 5       | The Data of the error injection = 5
The Data of the error free = 6       | The Data of the error injection = 6
The Data of the error free = 7       | The Data of the error injection = 7
The Data of the error free = 8       | The Data of the error injection = 8
The Data of the error free = 9       | The Data of the error injection = 100
The Data of the error free = 10      | The Data of the error injection = 10
The Data of the error free = 11      | The Data of the error injection = 11
The Data of the error free = 12      | The Data of the error injection = 12
The Data of the error free = 13      | The Data of the error injection = 13
The Data of the error free = 14      | The Data of the error injection = 14
[Thread debugging using libthread_db | [Thread debugging using libthread_db enabled]
[New Thread -1208059584 (LWP 10209)] | [New Thread -1208059584 (LWP 10305)]
[New SystemC Thread "main" 169678848 | [New SystemC Thread "main" 169690880]
[New Thread -1208730720 (LWP 10210)] | [New Thread -1208730720 (LWP 10308)]
```

File Edit View Terminal Tabs Help
Terminal
File Edit View Terminal Tabs Help

Fig. 11. Simulation results of the fault-free and fault injection experiments at timed functional transaction level.

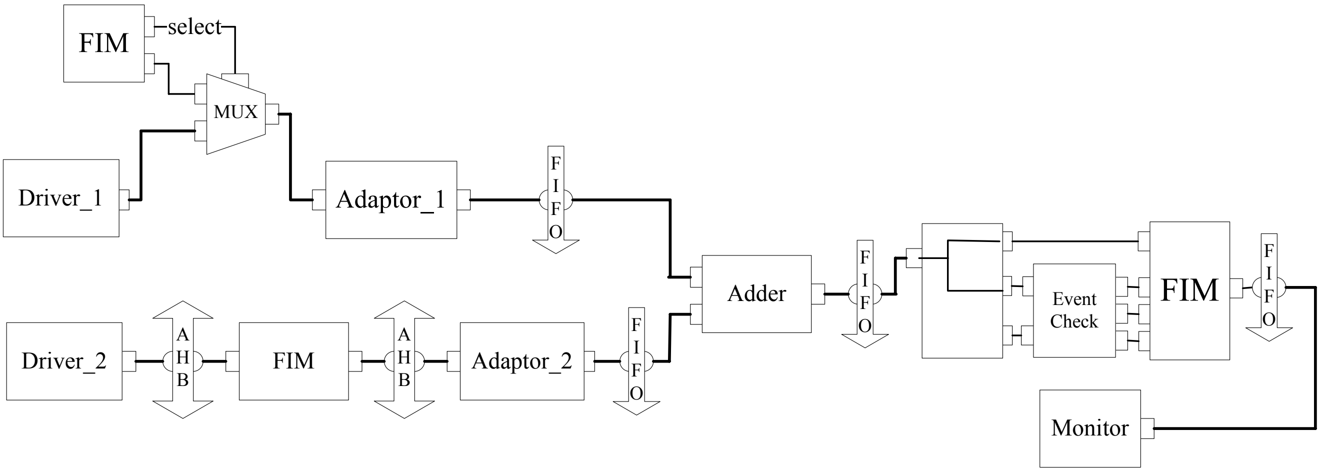
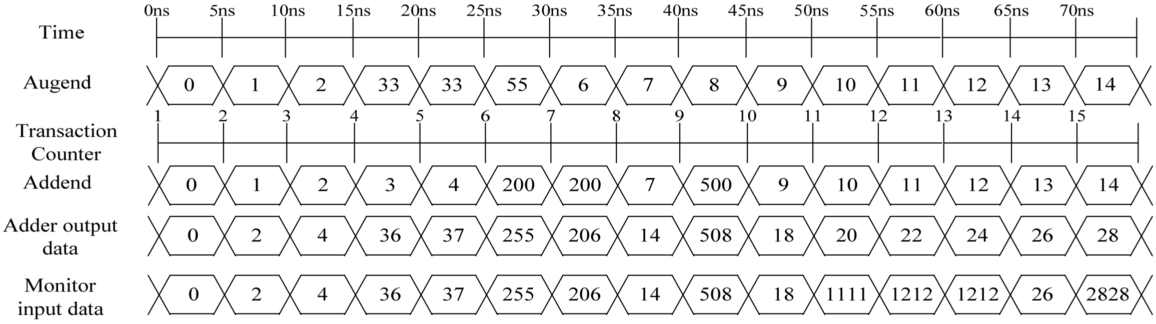
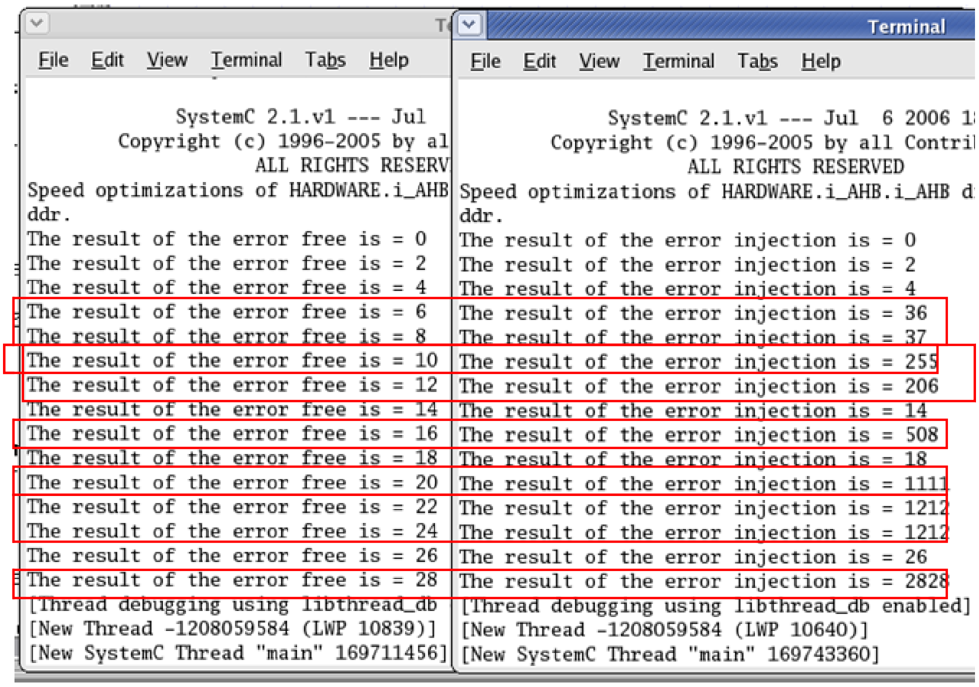


Fig. 12. Fault injection structure at mixed levels of abstraction.



(a)


Terminal
File Edit View Terminal Tabs Help
File Edit View Terminal Tabs Help

```
         SystemC 2.1.v1 --- Jul
     Copyright (c) 1996-2005 by al
                ALL RIGHTS RESERV
Speed optimizations of HARDWARE.i_AHB
ddr.
The result of the error free is = 0
The result of the error free is = 2
The result of the error free is = 4
The result of the error free is = 6
The result of the error free is = 8
The result of the error free is = 10
The result of the error free is = 12
The result of the error free is = 14
The result of the error free is = 16
The result of the error free is = 18
The result of the error free is = 20
The result of the error free is = 22
The result of the error free is = 24
The result of the error free is = 26
The result of the error free is = 28
[Thread debugging using libthread_db
[New Thread -1208059584 (LWP 10839)]
[New SystemC Thread "main" 169711456]
```

```
         SystemC 2.1.v1 --- Jul  6 2006 1
     Copyright (c) 1996-2005 by all Contri
                ALL RIGHTS RESERVED
Speed optimizations of HARDWARE.i_AHB.i_AHB d
ddr.
The result of the error injection is = 0
The result of the error injection is = 2
The result of the error injection is = 4
The result of the error injection is = 36
The result of the error injection is = 37
The result of the error injection is = 255
The result of the error injection is = 206
The result of the error injection is = 14
The result of the error injection is = 508
The result of the error injection is = 18
The result of the error injection is = 1111
The result of the error injection is = 1212
The result of the error injection is = 1212
The result of the error injection is = 26
The result of the error injection is = 2828
[Thread debugging using libthread_db enabled]
[New Thread -1208059584 (LWP 10640)]
[New SystemC Thread "main" 169743360]
```

(b)

Fig. 13. (a) Fault injection scenario. (b) Simulation results of fault-free and fault injection experiments.

IV. CONCLUSIONS AND FUTURE WORK

In this report, a system-level fault injection framework in SystemC design platform is presented. The proposed fault injection framework provides the methodologies for injecting the faults into various levels of abstraction. Three modeling levels considered in the framework are BCA level, untimed functional transaction level, and timed functional transaction level. The experiments based on CoWare Architect Platform were conducted to validate the feasibility of our fault injection approach. Contributions of this work are first to present the idea of distributed fault injection control to lower the control complexity of the fault injection compared to the centralized fault injection control; to develop the methodologies, including the time-triggered and event-triggered concepts, to inject the faults into different abstraction levels, and importantly to provide a solution for injection of faults into a system modeled at various levels of abstraction.

In the future, we will further explore the approach of distributed control of fault injection and compared to the method of centralized control of fault injection in terms of the complexity of experiment setup, compiling time, fault injection efficiency and simulation time. In addition, we will implement the proposed fault injection framework in the EDA tool of CoWare Architect Platform.

REFERENCES

[1] C. Constantinescu, "Impact of Deep Submicron Technology on Dependability of VLSI Circuits," *IEEE Intl. Conf. On Dependable Systems and Networks (DSN)*, pp. 205-209, 2002.

[2] P. Shivakumar et al., "Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic," *DSN*, pp. 389-398, 2002.

[3] T. Karnik, P. Hazucha, and J. Patel, "Characterization of Soft Errors Caused by Single Event Upsets in CMOS Processes," *IEEE Trans. on Dependable and Secure Computing*, Vol. 1, No. 2, pp. 128-143, April-June 2004.

[4] A. Fin, F. Fummi and G. Pravadelli, "AMLETO: a Multilanguage environment for functional test generation", *2001 International Test Conference*, pp. 821-829, Nov. 2001.

[5] Grotker Thorsten et al., "System Design with SystemC," Kluwer Academic Publishers, 2002.

[6] K. Rothbart et al., "High Level Fault Injection for Attack Simulation in Smart Cards," *13th Asian Test Symposium*, pp. 118-121, Nov. 2004.

[7] K. Rothbart et al., "A Smart Card Test Environment Using Multi-Level Fault Injection in SystemC", *6th IEEE Latin-American Test Workshop*, pp. 103-108, March-April 2005.

[8] K. Rothbart et al., "Power Consumption Profile Analysis for Security Attack Simulation in Smart Cards at High Abstraction Level," *EMSOFT*, pp. 214-217, Sept. 2005.

[9] Kuen-Long Leu, Yung-Yuan Chen, and Jwu-E Chen, "A Comparison of Fault Injection Experiments under Different Verification Environments", *IEEE Fourth International Conference on Information Technology and Applications*, pp. 582-587, Jan. 2007.

[10] Open SystemC Initiative (OSCI), "SystemC 2.0 Language Reference Manual," Revision 1.0, www.systemc.org, 2003.

[11] Bhasker Jayaram and J. Bhasker, "A SystemC Primer," Star Galaxy Publisher, 2004.

[12] CoWare Model Library, "AMBA Bus Library," Product Version V2005.2.2.

[13] CoWare Model Library, "TLM API Manual," Product Version V2005.

Abstract (second part)

This report describes the results achieved in the first year of three-year research proposal. As mentioned in the proposal, an important issue in the design of *SoC* with fault tolerance is how to verify the feasibility of the fault-robust design as early in the development phase to reduce the re-design cost. Therefore, a system-level fault-tolerant verification platform is required to assist the designers in assessing the dependability of a system with an efficient manner. The second part is going to discuss how to analyze the error coverage without using the fault injection and fault simulation mechanisms in order to save the development efforts and simulation time. Our idea is to devise a high-level abstract model to represent the fault-tolerant systems including the interconnection structure of the functional blocks, the propagation tables expressing the relationship between inputs and outputs for each functional block and the Petri Net to model the functional behavior of the fault-tolerant system. The fault-tolerant verification platform proposed here can save the time of detailed hardware implementation, benchmark program development, and fault injection campaigns. As a result, it is efficient to reduce the implementation and validation efforts. However, since our approach employs a high level of abstraction to model the fault-robust systems, the accuracy of the simulation results will decrease. A fault-tolerant VLIW core developed by our team is used to demonstrate the feasibility of our approach by comparing the results obtained from this approach with the results derived from the simulation-based fault injection technique by VHDL.

Keywords: error propagation path, high-level abstraction modeling, high-level rapid verification, Petri Net, system-on-chip (*SoC*), transient fault (soft error or SEU).

INTRODUCTION

Due to the high complexity of the system-on-chip (*SoC*), the behavioral level or higher abstraction level are used to model the *SoC* so as to tackle the complexity of the *SoC* design. It is well known that the rate of radiation-induced soft errors increases rapidly especially in combinational logic while the chip fabrication enters the deep submicron technology [1-3]. Such influences raise the urgent need to incorporate the fault tolerance into the high-performance systems [4-7]. However, the incorporation of the fault-tolerant demand into the *SoC* will further complicate the design problem. Importantly, we need to verify the feasibility of the fault-robust design as early in the development phase to reduce the re-design cost. Therefore, a system-level fault-tolerant verification platform is required to assist the designers in assessing the dependability of a system with an efficient manner.

We can validate the dependability of fault-tolerant systems by fault injection campaigns [8-10]. In general, the verification process of system robustness is performed by injecting the faults into the system and monitoring whether the faults are detected/recovered or cause the system failure, etc. The fault injection techniques presented in the previous literature can be classified as physical [11], software-implemented [12] and simulation-based [13, 14] fault injection approaches. The different classes of fault injection approaches provide a compromise between implementation efforts, simulation time and the accuracy of the experimental results. A major limitation of physical and software-implemented approaches is that dependability evaluation is performed after physical systems have been built. While dependability evaluation is necessary after systems have been built, the costs of re-designing systems due to inadequate dependability can be prohibitively expensive. The simulation-based fault injection uses the simulation to inject faults in simulation models of systems. The simulation model of systems can be described in hardware description language like VHDL. The advantage of simulation-based mechanism is that the system dependability can be assessed as early in the design phase, and if necessary to re-design the system, the cost of re-design is reduced significantly. Although the simulation-based approach shows a valuable means to support the validation of the fault-tolerant systems, it still requires considerable efforts to model the system implementation at different abstraction levels, to develop the benchmark programs as well as the fault injection tools, and to perform the fault injection campaigns. The goal of this study is to propose a new fault-tolerant verification approach that can significantly reduce the validation efforts compared to the simulation-based approach.

As discussed before, several issues should be addressed in the fault-tolerant verification process: First is the way of faults/errors injected; second is the paths of faults/errors propagated and third is the outcomes of faults/errors processed and analyzed. It is clear that the verification efficiency can be enhanced if we can more effectively cope with the above issues. For this purpose, we devise a high-level abstract modeling methodology to modeling the fault-tolerant systems where the emphasis of system modeling is more on the error propagation and error handling, and less on the details of the implementation of the functional units. Since the proposed modeling methodology focuses on the function of fault-robust validation only, the complexity of system models will decrease. Therefore, it reduces the efforts to modeling the systems and the time to performing the fault injection campaigns and error coverage analysis. However, since our fault-tolerant verification approach employs a high level of abstraction to modeling the systems, the accuracy of the simulation results could be hurt.

The rest of the report is organized as follows: In Section 2, the methodology to modeling the fault-robust systems is proposed. Section 3 uses a fault-tolerant VLIW core to demonstrate the proposed fault-tolerant verification approach. In Section 4, the simulation results are provided and compared with the results derived from the simulation-based fault injection approach by VHDL. The conclusions appear in Section 5.

FAULT-ROBUST SYSTEM MODELING METHODOLOGY

The goal of the modeling methodology is to lower the complexity of the modeling, simulation and analysis of the fault-tolerant systems. The basic idea is to find out the data flow paths of each system operation. Then, locate all possible errors which could occur in the data flow paths for a particular operation under a specific error model. For each operation, we can inject the desired errors from the error model into the corresponding data flow paths and check whether the detection and recovery schemes embedded in the system can tolerate the errors or not. To support the validation of system robustness, the proposed system model must have the capability to propagate the errors while the system is executed.

Based on the above discussion, we develop a high-level abstract model to modeling the fault-robust systems. The simulation model of systems comprises the following three parts:

1. the interconnection structure of the functional units;
2. the propagation tables expressing the relationship between inputs and outputs for each functional unit;
3. the Petri net structure [15] to model the functional behavior of the fault-tolerant systems.

More specifically, the propagation tables can be utilized to propagate the errors from the inputs to the outputs of each functional unit. And through the interconnections of the functional units, the effect of errors will be propagated. An abstract error model is exploited to generate the desired error patterns for the system under validation. The function of Petri net model is to control the operations of the system. For each operation represented by a place in the Petri net graph, we also need to store its control signals for the corresponding functional units which are responsible for the execution of the operation. We can count on the Petri net model and the control signals for each operation to derive the corresponding data flow paths for a particular operation. Then, all possible errors which could happen in those data flow paths for a specific operation can be located. In that way, we can generate the error list for each operation. In other words, our verification approach can produce the propagation paths for each error to see whether the paths of error propagation have the detection and recovery protection or not. Therefore, we can examine the error patterns one by one for a particular operation to acquire the dependability data for robustness validation. Finally, the error coverage of a system can be derived from the detailed analysis of the error coverage related to each operation. In addition to the error coverage evaluation, the analysis can also discover the single failure points or weak points of the systems that can be utilized to improve the system dependability further.

CASE STUDY

A fault-tolerant VLIW core [16] is used to demonstrate the concept of our approach. For simplicity of demonstration, we adopt the portion of the execution stage of VLIW core as shown in Figure 1 to illustrate the modeling methodology and the fault-robust verification process. In Figure 1, ‘CP’ and ‘TMR_MV’ denote the ‘comparator’ and ‘triple modular redundancy majority voter’, respectively. The fault-tolerant scheme employed in [16] is briefly described as follows:

```
while (not end of program)
  {switch (Number of instructions in an execution packet
           for ALU.)
    {case ‘1’: TMR_MV(ALU_1, ALU_2, ALU_3); if
           (TMR_MV detects more than one ALU
           failure) then the “Error-recovery process”
           is activated to recover the failed
           instruction.
     case ‘2’: the execution packet contains two
              instructions: $I_1$ and $I_2$.
              $I_1$: CP1(ALU_1, ALU_2);
              $I_2$: CP2(ALU_3, ALU_4);
              if ($I_1$ fails) then the “Error-recovery
              process” is activated to recover $I_1$.
              if ($I_2$ fails) then the “Error-recovery
              process” is activated to recover $I_2$.
     case ‘3’:the packet is divided to two packets and
           executed sequentially.
    }}
```

Error-recovery process:

```
    $i \leftarrow 1$;
    While (number of retries $r\_no > 0$)
      {TMR_MV(ALU_$i$, ALU_$i+1$, ALU_$i+2$);
       if (TMR_MV succeeds) then the error recovery
       succeeds → exit;
       else { $r\_no \leftarrow r\_no - 1$ ; $i \leftarrow i+1$ ; if
    ($i \geq 3$) then $i \leftarrow 1$;}}
  recovery failure and the system enters the fail-safe
  state.
```

Figure 2 exhibits the simulation model of the system illustrated in Figure 1, where $r\_no = 2$. In this case study, there are two normal system operations: an execution packet containing one ALU instruction or two ALU instructions. As can be seen from Figure 2(b), these two operations termed as target operations are notated by the places of ‘P1inst’ and ‘P2inst’, respectively. The other places are used to model the operations/functions of the fault-tolerant scheme presented above. We now exploit Figure 2 to explain the modeling methodology and fault-robust verification approach.

Modeling methodology:

1. The interconnection description of the functional units; we create a file to describe the interconnection relationship among the functional units.
2. Figure 2(a) shows the propagation tables for the functional units, where ‘eq’ and ‘neq’ represent ‘equal’ and ‘not equal’, respectively.
3. Figure 2(b) shows the Petri net graph created to model the system as exhibited in Figure 1. The control signals for target operation ‘P1inst’ and its associated operations ‘Prc1’ and ‘Prc2’ are provided in the control table as displayed in Figure 2(c). For example, when the operation ‘P1inst’ is executed, the control table is employed to produce the corresponding control signals, such as Sch_control = ‘000’ and mux_a = ‘0’, to perform the execution of one ALU instruction with TMR protection. Figure 2(d) gives the conditions for firing the transition from the input place to the output place.

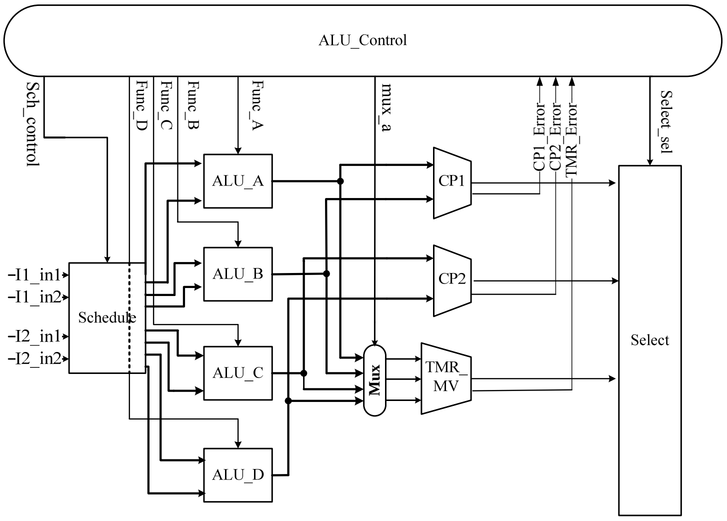
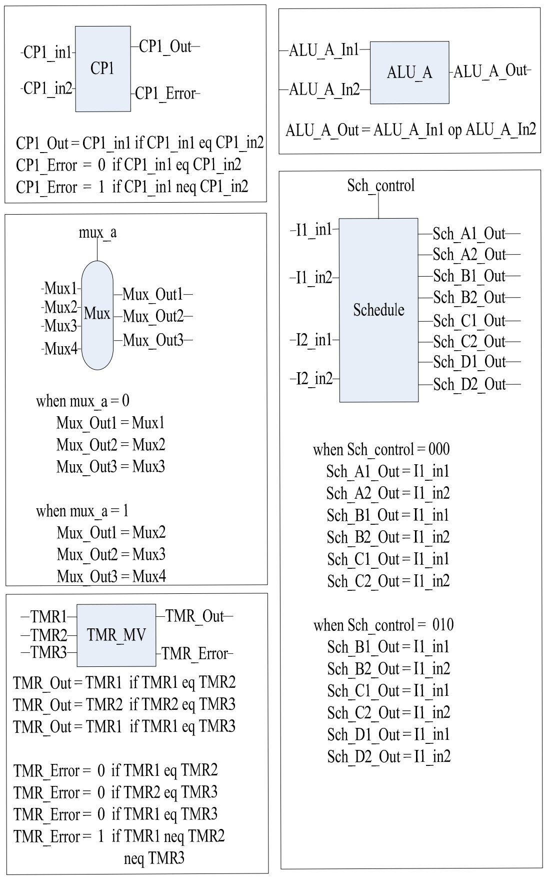


Figure 1. Fault-robust case study.



-> P1inst
Sch_control=000
mux_a=0

-> Prc1
Sch_control=000
mux_a=0

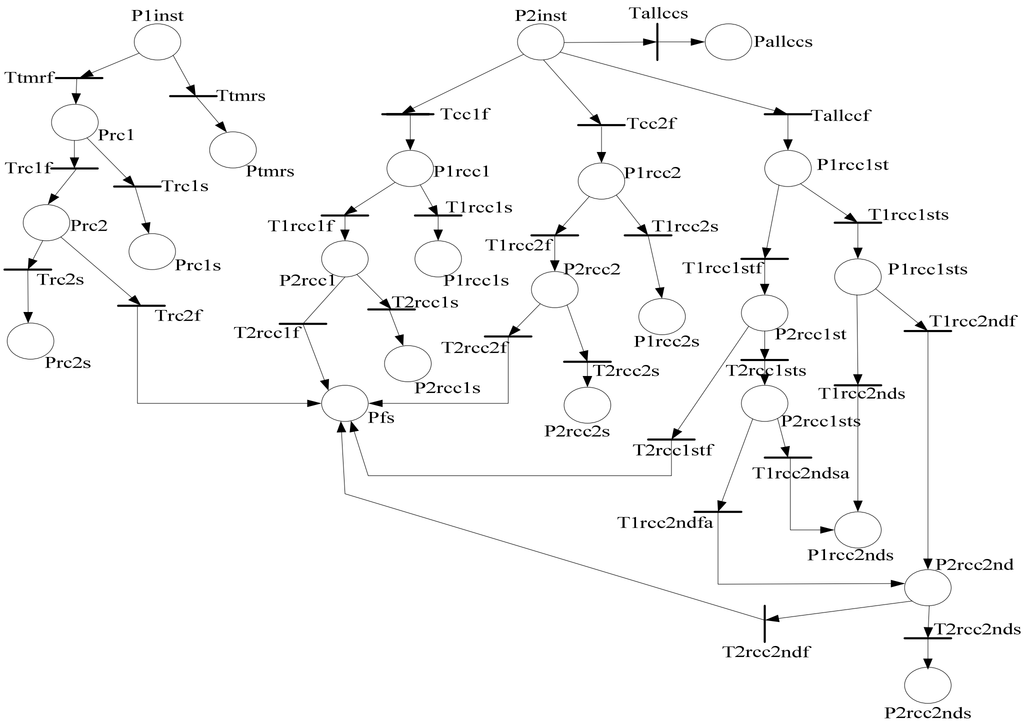
-> Prc2
Sch_control=010
mux_a=1

(c). Control table

Ttmrs	-> TMR_Error(0)
Ttmrf	-> TMR_Error(1)
Trc1s	-> TMR_Error(0)
Trc1f	-> TMR_Error(1)
Trc2s	-> TMR_Error(0)
Trc2f	-> TMR_Error(1)

(d). Transition fire condition table

(a). Propagation tables



(b). Petri net system modeling

Figure 2. The simulation model of the system illustrated in Figure 1, where $r\_no = 2$.

In the following, the target operation 'P1inst' is employed to explain our fault-tolerant verification approach.

Step 1: The current place in Figure 2 (b) is 'P1inst'. The data flow paths of this operation can be generated by applying the required control signals offered in Figure 2(c), i.e. Sch_control = '000' and mux_a = '0'. The procedure of path generation is briefly depicted below (note that the interconnections of the functional units are implicitly applied in the following demonstration of path generation).

- Schedule: Since Sch_control = '000', according to the propagation table of Schedule, I1_in1 (data1) is propagated to Sch_A1_Out, Sch_B1_Out and Sch_C1_Out, and I1_in2 (data2) is propagated to Sch_A2_Out, Sch_B2_Out and Sch_C2_Out.
- ALU: Based on the propagation table of ALU, ALU_A_Out, ALU_B_Out and ALU_C_Out are all equal to 'data1 op data2'.
- Mux: mux_a = '0', and therefore, 'data1 op data2' is propagated to Mux_Out1, Mux_Out2 and Mux_Out3.
- TMR_MV: According to the propagation table of TMR_MV, TMR_Out is 'data1 op data2' and TMR_Error is 0.

Step 2: In Step 1, we have collected the data flow paths for the operation 'P1inst'. Then, find out all possible errors which could occur in the data flow paths for 'P1inst' under a selected error model. Here, we adopt the following error model to generate the error patterns that will be used to check the system robustness while 'P1inst' operation is executed.

Error model: To simplify the modeling complexity and reduce the simulation time, we omit the details of the functional units in the system modeling. However, there is no way to inject the faults into the inside of the functional units. Therefore, the errors only can be injected in the outputs of the units. We consider the errors either occurring in a single output port or in the two different output ports.

According to the above error model, we can create the possible errors, which could happen in the data flow paths of the operation 'P1inst'. So, if an error occurs in the 'P1inst' paths and meanwhile the system is executing the operation of 'P1inst', then this error could affect the execution result. For each operation, we can inject the possible errors from the error model into the corresponding data flow paths and investigate whether the detection and recovery schemes built in the system can tolerate the errors or not.

Case 1: A single output port error; an error is injected into the Sch_A1_out port as shown in Figure 2(a), propagation table of Schedule unit. As can be seen from Step 1, 'data1' is changed to 'wrong_data'. Clearly, the error will be propagated to Mux_Out1, then TMR1. So, TMR1 becomes 'wrong_data op data2'. In the meantime, TMR2 and TMR3 contain the expression 'data1 op data2'. As a result, TMR_Out is 'data1 op data2' and TMR_Error is 0. It means that the error can be overcome. Next, there are two output places, 'Prc1' and 'Ptmrs', for input place 'P1inst', where the outcome of TMR_Error decides which transition will be enabled. Since TMR_Error is 0, from Figure 2(d), the transition 'Ttmrs' is fired and the place is transited from 'P1inst' to 'Ptmrs'. There is no more transition when the place is in 'Ptmrs'. Record the result and activate the next error injection.

Case 2: Two output port errors; an error is injected into the Sch_A1_out port and the other into Sch_B1_out port. Similarly, the errors will be propagated to TMR_MV inputs, and they are 'wrong_data1 op data2', 'wrong_data2 op data2', and 'data1 op data2', respectively. Consequently, TMR_MV fails to produce the correct answer, and sets TMR_Error is one. Next, the transition 'Ttmrf' is fired and the place 'Prc1' is executed. The operation 'Prc1' is the first error recovery, and if it succeeds, then the transition 'Trc1s' is fired; else, the operation of second recovery 'Prc2' will be activated. Finally, if error recovery succeeds, then the system enters the place 'Prc2s'; else, the system goes into the fail-safe state.

Figure 3 illustrates the complete fault-robust verification process. A platform based on the verification process is developed and used to evaluate the error coverage of the fault-tolerant systems.

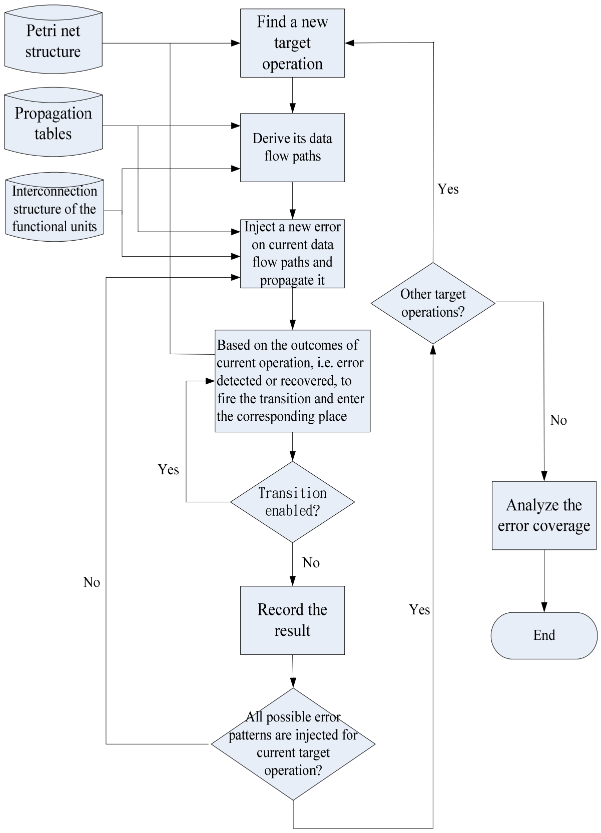

Petri net structure
Propagation tables
Interconnection structure of the functional units
Find a new target operation
Derive its data flow paths
Inject a new error on current data flow paths and propagate it
Based on the outcomes of current operation, i.e. error detected or recovered, to fire the transition and enter the corresponding place
Yes
Transition enabled?
No
Record the result
No
All possible error patterns are injected for current target operation?
Yes
Yes
Other target operations?
No
Analyze the error coverage
End

Figure 3. The complete fault-robust verification process.

SIMULATION RESULTS

In this section, we use the proposed verification platform to assess the error coverage of system as shown in Figure 1. The design metrics as described below are exploited to justify our approach:

- $Ce-\det$: Error-detection coverage, i.e. probability of errors detected;
- $Ce-rec$: Error-recovery coverage, i.e. probability of errors recovered given errors detected;
- Ce: Error coverage, i.e. probability of errors detected and recovered;
- $Pf-uns$: Probability of system entering the fail-unsafe state;
- $Pt-\det-f-s$: State transition probability from 'detected' state to 'fail-safe' state.

Table 1 presents the simulation results of the design metrics. The data shown in Model1 are derived from the assumption that the occurring probability is the same for each error in the error model. As shown in [2], the fault/error rate is proportional to the circuit area. Therefore, the occurring probability for the errors located at various functional units should not be identical. The data shown in Model2 take the area effect on the error rate into account. However, the proposed high-level modeling methodology is for the verification purpose of the system robustness. It cannot be used to estimate the area of the functional units. The VHDL design flow is adopted to obtain the area of the units as displayed in Figure 1. To justify the feasibility of our approach and the accuracy of the simulation results, we also conduct the simulation-based fault injection campaigns at RTL level by VHDL design language. Table 2 lists the experimental results.

Table 1. The simulation results based on our method.

	$C_{e\text{-det}}$	$C_{e\text{-rec}}$	C_e	$P_{t\text{-det-f-s}}$	$P_{f\text{-uns}}$
Model1	0.9269	0.8710	0.8073	0.0358	0.1595
Model2	0.9971	0.9187	0.9160	0.0390	0.0450

Table 2. Results from simulation-based approach.

	Faults rate	Ce-det	Ce-rec	Ce	Pt-det-fs	Pf-uns
Workload1 Inst1 : 80% Inst2 : 20%	**w1-1** 1 fault:90% 2 faults:10%	0.999311	0.998376	0.997688	0.001238	0.001076
	w1-2 1 fault : 50% 2 faults : 50%	0.999305	0.995810	0.995118	0.003662	0.001223
	w1-3 1 fault : 10% 2 faults : 90%	0.999217	0.993393	0.992616	0.005986	0.001403
Workload2 Inst1 : 50% Inst2 : 50%	**w2-1** 1 fault : 90% 2 faults : 10%	0.999010	0.998110	0.997122	0.001501	0.001379
	w2-2 1 fault : 50% 2 faults : 50%	0.998754	0.995374	0.994133	0.004214	0.001658
	w2-3 1 fault : 10% 2 faults : 90%	0.998657	0.993268	0.991934	0.006301	0.001673
Workload3 Inst1 : 20% Inst2 : 80%	**w3-1** 1 fault : 90% 2 faults : 10%	0.998607	0.997900	0.996510	0.001486	0.002006
	w3-2 1 fault : 50% 2 faults : 50%	0.998140	0.995215	0.993364	0.004115	0.002529
	w3-3 1 fault : 10% 2 faults : 90%	0.997549	0.992020	0.989588	0.007017	0.003412

Three workloads are developed for the experiments. For fair comparison, three workloads have the various ratios of one and two ALU instructions in an execution packet. According to Table 2, we confirm that the area factor plays an important role in the evaluation of the error coverage. The comparison results between VHDL and our approach with area consideration are summarized as follows (represented by the percentage of the difference): -0.17 ~ 0.05% for error-detection coverage, -7.24 ~ -6.64% for error-recovery coverage, and -7.4 ~ -6.64 % for error coverage.

CONCLUSIONS

A new fault-tolerant verification platform has been proposed to drastically reduce the validation effort and time compared to the previous methodologies. Our fault-tolerant verification platform does not require the detailed hardware implementation, benchmark program development, and fault injection campaigns. However, since our verification approach employs a high level of abstraction to model the fault-robust system, the accuracy of the simulation results will decrease. The preliminary results show that the accuracy of our approach is acceptable, and the verification flow can achieve a rapid dependability assessment. Such a verification flow can significantly decrease the iteration time between different design levels.

REFERENCES

[1] C. Constantinescu, “Impact of Deep Submicron Technology on Dependability of VLSI Circuits,” *IEEE Intl. Conf. On Dependable Systems and Networks (DSN’02)*, pp. 205-209, 2002.

[2] P. Shivakumar et al., “Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic,” *DSN’02*, pp. 389-398, 2002.

[3] T. Karnik, P. Hazucha, and J. Patel, “Characterization of Soft Errors Caused by Single Event Upsets in CMOS Processes,” *IEEE Trans. on Dependable and Secure Computing*, Vol. 1, No. 2, pp. 128-143, April-June 2004.

[4] N. Quach, “High Availability and Reliability in The Itanium Processor,” *IEEE Micro*, Vol. 20, issue: 5, pp. 61-69, September-October 2000.

[5] J. B. Nickle and A. K. Somani, “REESE: A Method of Soft Error Detection in Microprocessors,” *DSN’01*, pp. 401-410, 2001.

[6] S. Mitra et al., “Robust System Design with Built-In Soft-Error Resilience”, *IEEE computer*, pp. 43-52, Feb. 2005.

[7] M. K. Qureshi, O. Mutlu and Y. N. Patt, “Microarchitecture-Based Introspection: A Technique for Transient-Fault Tolerance in Microprocessors”, *DSN’05*, pp. 434 – 443, June-July 2005.

[8] J. Clark and D. Pradhan, “Fault Injection: A Method for Validating Computer-System Dependability,” *IEEE Computer*, 28(6), pp. 47-56, June 1995.

[9] M. C. Hsueh, T. K. Tsai and R. K. Iyer, “Fault Injection Techniques and Tools,” *IEEE Computer*, 30(4), pp. 75-82, April 1997.

[10] C. Constantinescu, “Experimental Evaluation of Error-Detection Mechanisms,” *IEEE Trans. on Reliability*, Vol. 52, No. 1, pp. 53-57, March 2003.

[11] J. Karlsson, P. Liden, P. Dahlgren, R. Johansson and U. Gunneflo, Using Heavy-Ion Radiation to Validate Fault-Handling Mechanisms, *IEEE Micro*, *14*(1), Feb. 1994, 8-23.

[12] G. A. Kanawati, N. A. Kanawati and J. A. Abraham, FERRARI: A Tool for the Validation of System Dependability Properties, *FTCS-22,* 1992, 336-344.

[13] E. Jenn et al., “Fault Injection into VHDL Models: The MEFISTO Tool,” *24th IEEE FTCS*, pp. 66-75, 1994.

[14] J. Gracia et al., “Comparison and Application of Different VHDL-Based Fault Injection Techniques,” *DFT’01*, pp. 233-241, 2001.

[15] J. L. Peterson, “Petri Net Theory and the Modeling of System”, Prentice-Hall, Inc., Englewood Cliffs, New York (1981).

[16] Yung-Yuan Chen**,** Kuen-Long Leu and Chao-Sung Yeh, “Fault-Tolerant VLIW Processor Design and Error Coverage Analysis,” The 2006 IFIP International Conference on Embedded and Ubiquitous Computing, pp. 754-765, August 2006.

Self-Evaluation of Research Results:

- The above report summarizes the first-year results accomplished from this three-year research project. It is evident that 99% of the work has been achieved and the preliminary results have been published. The extended versions of the results will be submitted to be considered for journal publication. However, the subjects described in our proposal are big and deserve to be further explored. We definitely achieve the first-year goals set in the proposal.
- We are going to develop a system-level fault-injection tool, which exploits the simulation-based fault injection scheme proposed in this research and can be installed in the CoWare Architect Platform. The tool takes the fault scenario description from the user and then automatically generates the system platform supplemented with the fault injection capability. This kind of fault injection tool can not only facilitate the failure mode and effect analysis (FMEA) and the fault-tolerant validation process, but raise the validation efficiency. The embedded fault-tolerant systems have found fertile ground in intelligent system applications, such as intelligent driver assistance system or intelligent robot system, which require a stringent dependability while the systems are in operation. Since more works depend on the intelligent machines, the reliability issue becomes more important than ever. The fault-tolerant verification platform developed from this research can be applied to the design and analysis of the fault-tolerant systems modeled at high level of abstraction to enhance the overall system dependability. The previous study for the fault injection approach mainly focuses on the VHDL modeling level and rarely discusses the fault injection in system-level design. We want to fulfill this lack.

Publications associated with this research:

- Kuen-Long Leu, Yung-Yuan Chen and Jwu-E Chen, “A Comparison of Fault Injection Experiments under Different Verification Environments”, *IEEE Fourth International Conference on Information Technology & Applications*, pp. 582-587, Jan. 2007. (EI)
- Yung-Yuan Chen and Geng-Wei Wu, “Fault-Tolerant Verification Platform for Systems Modeled at High Level of Abstraction”, *1st IEEE Systems conference*, pp. 1-7, April 2007. (EI)
- Kun-Jun Chang and Yung-Yuan Chen, “System-Level Fault Injection in SystemC Design Platform,” *8th International Symposium on Advanced Intelligent Systems*, pp. 354-359, Sept. 2007.

A Comparison of Fault Injection Experiments under Different Verification Environments

[1]Kuen-Long Leu, [2]Yung-Yuan Chen and [1]Jwu-E Chen

[1]Department of Electrical Engineering
National Central University
Tao-Yuan, Taiwan
E-mail: 945401025@cc.ncu.edu.tw
jechen@ee.ncu.edu.tw

[2]Department of Computer Science
and Information Engineering
Chung-Hua University
Hsin-Chu, Taiwan
E-mail: chenyy@chu.edu.tw

***Abstract*--The main work of this paper is to characterize the dependability of fault-tolerant systems by using two different hardware design environments (SystemC and VHDL). For SystemC, we inject *errors* into the components' outputs, whereas *faults* into the inside of components for VHDL. The difference of the simulation results between SystemC and VHDL is discussed thoroughly through observing two parameters: one is the probability of a fault causing an effective error and another is the relationship between fault duration and error duration. The above two parameters dominate the discrepancy between the two different platforms. The experimental results show the effect of the parameters on the error coverage. This study can promote the fault-tolerant design and verification environment to a higher abstraction level.**

***Index Terms*--Error/fault injection, fault-tolerant verification platform, hardware design language, SystemC.**

I. INTRODUCTION

As system-on-chip (*SoC*) becomes more and more complicated, and contains a large number of transistors, 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 deep submicron technology [1]-[3]. Thus, it is essential to employ the fault-tolerant techniques in the design of *SoC* to guarantee a high operational reliability in critical applications. Recently, the reliability issue in high-end processors is getting more and more attention [4]-[7]. For example, the Intel Itanium processor provides fault-tolerant features [7], such as enhanced machine check abort (MCA) architecture with extensive error correcting code (ECC), to maximize system reliability and availability.

Generally, there are two kinds of methodologies used to verify the dependability of fault-tolerant systems. One is *physical fault injection* [2] that injects the faults at the IC pin-level, by heavy-ion radiation, by interference with the IC power supplies, or by mutating code and corrupting program state variables. The other is *simulated fault injection* [8]-[10] that uses the simulation to inject faults in simulation model of systems. We can describe the simulation model of systems by hardware description language like VHDL and SystemC. The advantage of simulated fault injection mechanism is that the system dependability can be assessed as early in the design phase, and if necessary to re-design the system, the cost of re-design is reduced significantly.

Recently, the development and verification environment is gradually promoted from RTL to behavioral or system level due to the complexity of *SoC* design. Two popular hardware description languages, Verilog and VHDL, are not adequate to support the system-level design in more abstract description. The SystemC comes to fill the need of system design [11]-[14]. Because each component developed by SystemC may only contain its behavioral description, the detailed hardware structure is not definite at this level. Therefore, it is impossible to inject a fault into the inside of components. Instead, only the errors can be injected into the components' outputs. The pity is that previous literatures seldom mention the relationship between the fault and error. So designers who want to develop a fault-tolerant system upon higher level of abstraction have no idea about how to link the error scenario to fault scenario. For above reason, this paper wants to propose some practical suggestions to help designers derive more actual simulation results when they are verifying their fault-tolerant systems.

The remaining sections are organized as follows. We briefly introduce the existing error injection methodologies in Section 2 and present ours in Section 3. In Section 4, we discuss and compare the fault injection results derived from the VHDL and SystemC simulation models of a 32-bit fault-tolerant VLIW processor. Finally we propose some valuable conclusions in Section 5 to help derive more accurate experimental results of error simulation at higher abstraction levels.

II. RELATED WORK

There are only a few approaches of error injection at high abstraction level based on SystemC. Rothbart *et al.* [12] inserted fault injection modules (FIM) into the interconnection of the function blocks and fault injection ports (FIP) in the Memory. A fault-injection control unit (FICU) to accomplish the fault simulation at high abstraction level controls the FIM and FIP. Although this methodology will not modify the component description, the FICU will become very complicated if the tested system contains many functional blocks. Fin *et al.* [13] performed the error injection into SystemC models by presenting a multi-language environment for functional test generation, but they did not observe the behavior of the faulty system. Reference [14] proposed an automated synthesis of single-event upset (SEU) Tolerant architecture based on SystemC environment. Moreover, a tool is provided to allow performing error injecting at behavioral level to validate the SEU tolerant circuits. However, the error injection targets only focus on the storage elements.

III. ERROR INJECTION METHODOLOGY IN SYSTEMC

A basic simulation model of SystemC involves three blocks including “Stimulus”, “HW/SW” and “Monitor” [11]. The “Stimulus” is responsible for reading the test input file and passing the test patterns at each clock cycle to “HW/SW” which contains the system description and the “Monitor” records systems’ outputs. Because all of the three blocks are established upon the C/C++ platform, every signal and input/output port in real hardware can be viewed as a variable in high-level language. If a certain signal is selected to be the fault or error injection target, its value can be altered arbitrarily by declaring its scope as global. According to this principle, we append another block called “Error injection file” to construct our error simulation model as shown in Fig. 1. This file specifies the information for each error injection including the injection time instant, injection target, error type and the error duration. Once the simulation time reaches the injection time of an error, the erroneous value will replace the content of the corresponding port or signal.

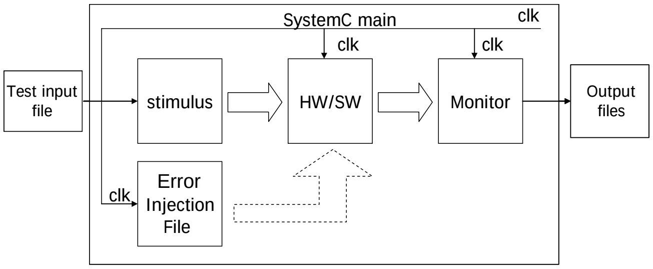


Fig. 1: Error simulation model.

A. *Timing Model of SystemC*

In SystemC, each process will be triggered by certain events. Each event is assigned to a virtual time delay Δ.

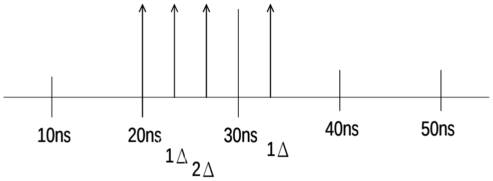


Fig. 2: Virtual timing model and Δ delay of SystemC.

The Δ is used to illustrate the relation between cause and effect of real hardware components. As Fig. 2 depicts, there is an event triggered at 20ns, and this event will activate the second event at 20+1Δ ns. Then the second event further activates the third event at 20+2Δ ns, and so on. Such ripple effect will persist until no more events happen for current iteration.

Due to the Δ delay, the following situation will result in the error injection fail: Assume an error is injected into an ALU output at 20 ns and this error will be propagated to the next stage at 30 ns. However, ALU is triggered until 20+2Δ ns such that the error injected at 20 ns will be overwritten by the new result. Thus, the error injected becomes ineffective. To solve this problem, each component that could be an injection target needs modifying as illustrated in Fig. 3.

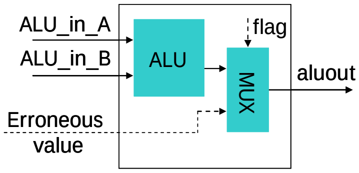


Fig. 3: A modified ALU block for error injection.

We declare the ‘flag’ and ‘Erroneous value’ in Fig. 3 as global variables so that we can control their values easily. Fig. 4 is an error injection file with flag insertion. Once the simulation time reaches 90 ns, the flag will be set to one, and the multiplexer will choose the erroneous value as the output. The flag will return to zero at 110 ns and at that time the ALU block restores to its normal operation. In this case, the error duration is 20 ns.

```
#include “error_injection_file.h"
#include “system_top.h"
extern systemc_top S1 ;
void Error_in::prc_error_in() {
  run_time = sc_simulation_time() ;
  switch(run_time) {
   case 90 : {
    S1.ALU_top_unit->ALU->flag.write(1) ;
    S1.ALU_top_unit->ALU->erroneous_value = 1001987144 ;
   } break ;
   case 110 : {
    S1.ALU_top_unit->ALU->flag.write(0) ;
   } break ;
  }
}
```

Fig. 4: An error injection file example.

Although this methodology will modify the original component description, the modification is very slight. Furthermore, the insertion of flags and multiplexers can be performed automatically due to its regularity, thus there is no additional burden to designers. In addition, this methodology can apply to not only ALU but also any other functional blocks and storage elements. We have integrated this error injection process into our development and verification framework for the validation of the fault-tolerant systems. We describe the framework next.

B. *Development and Verification framework*

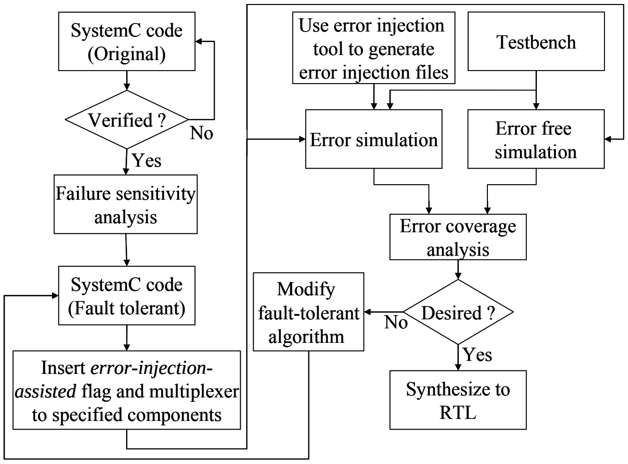

SystemC code (Original)
Verified ?
No
Yes
Failure sensitivity analysis
SystemC code (Fault tolerant)
Insert error-injection-assisted flag and multiplexer to specified components
Use error injection tool to generate error injection files
Testbench
Error simulation
Error free simulation
Error coverage analysis
Modify fault-tolerant algorithm
No
Desired ?
Yes
Synthesize to RTL

Fig. 5: Development and Verification framework.

Fig. 5 shows the framework to develop and validate the fault-tolerant systems based on SystemC platform. After the original design has been verified to meet its functional specification, the failure sensitivity analysis will generate a report providing which components with the higher probability of causing the system failure due to the errors. According to this report, designers can add the protection mechanisms to those vulnerable components to re-build an enhanced design with fault-tolerant capability. Next, the fault-tolerant design is analyzed to determine which components should be inserted the *error-injection-assisted* flag and multiplexer to facilitate the fault simulation. To generate the error injection file mentioned earlier, we have developed an error injection tool to produce the error injection file automatically. By giving four parameters including the total number of errors injected, error duration, simulation time and quantity of experiments, designer can use this tool to not only generate the error injection files but also derive the analysis about the degree of error overlapping as shown in Fig. 6. The *Weibull distribution* is utilized for determining the occurring time of each error. Numbers on the top of Fig. 6 represent how many errors overlap over a certain time fraction. The various degrees of error overlapping represent the different error environments. Designers can alter the parameters stated above to generate the desired error environment.

Next, the error-free simulation and error simulation can be performed with the specified workloads. Both simulation results are used for error coverage analysis. The designer should revise his/her fault-tolerant algorithm and re-run the implementation and simulation iteration until the error coverage reaches the desired level. After that, the design can be synthesized to RTL for further process.

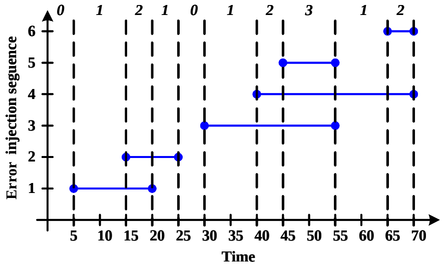


Fig. 6: Error overlapping degree.

The VHDL platform-based framework is similar to the SystmeC except that flag and multiplexer are not required.

IV. EXPERIMENTAL RESULTS

A. *Fault-Tolerant VLIW Data Path Design*

To validate the proposed approach, an experimental fault-tolerant VLIW architecture is developed [15]. This experimental architecture as displayed in Fig. 7 can issue at most three ALU and three load/store instructions per cycle. For simplicity of demonstration, the concurrent error detection and recovery do not apply to the load/store units. The fault-tolerant VLIW processor is briefly described as follows:

1. For only one ALU instruction executed in current clock cycle, the 'Dispatch' circuit will duplicate this instruction to three ALUs simultaneously. Then, the TMR_MV will check the consistency of the ALUs' outputs.
2. For two ALU instructions in current execution slot, the 'Dispatch' circuit will duplicate the first instruction to ALU_1 and ALU_2 and the second to ALU_3 and ALU_4, respectively, and then the CP1 and CP2 will check the consistency of ALUs' outputs, respectively.
3. For three ALU instructions in an execution slot, they will be partitioned into two execution slots. At the first slot, two instructions will be processed like case 2 and then the remaining slot will be processed like case 1 at the second cycle.
4. If there is any inconsistency, the recovery mechanism is activated. The consistency will be checked again by TMR_MV. If the inconsistency disappears, then VLIW can continue to process the next instruction(s); else the whole VLW idles.

Note that the 'Error Analysis' block in execution stage, which was created only to facilitate the measurement of the error coverage during the fault and error injection campaigns, is not a component for the VLIW processor displayed in Fig. 7.

The fault-tolerant VLIW processor based on the architecture of Fig. 7 was realized in VHDL and SystemC, respectively.

B. *Fault-tolerant design metrics*

The design metrics as described below are exploited to justify our fault-tolerant approach:

- P_{f-uns} : Probability of system entering the fail-unsafe state;
- $C_{e-\det}$: Error-detection coverage, i.e. probability of errors detected;
- C_{e-rec} : Error-recovery coverage, i.e. probability of errors recovered given errors detected;
- C_e : Error coverage, i.e. probability of errors detected and recovered;
- P_{f-s} : Probability of system entering the fail- safe state;
- $P_{t-\det-f-s}$: State transition probability from 'detected' state to 'fail-safe' state.
- $P_{t-\det-f-uns}$: State transition probability from 'detected' state to 'fail-unsafe' state.
- $P_{f-uns-\det}$: Probability of system entering the fail-unsafe state due to the detection defects;
- $P_{f-uns-rec}$: Probability of system entering the fail-unsafe state due to the recovery defects;
- P_{f-to-e} : Probability of a fault causing an effective error.

The following parameters *Ne, Ne-det, Ne-esc-det, Ne-rec, Ne-nrec-f-s and Ne-nrec-f-uns* (called the error-related parameters) represent the total number of errors occurred, the number of errors detected, the number of errors escape being detected, the number of errors recovered, the number of errors not recovered and system enters the 'fail-safe' state and the number of errors not recovered and system enters the 'fail-unsafe' state, respectively. The design metrics can be expressed as follows:

$$P_{f-uns-\det} = \frac{N_{e-esc-\det}}{N_e}; C_{e-\det} = \frac{N_{e-\det}}{N_e} = 1 - P_{f-uns-\det}; C_{e-rec} = \frac{N_{e-rec}}{N_{e-\det}}; \qquad (1)$$

$$P_{t-\det-f-s} = \frac{N_{e-nrec-f-s}}{N_{e-\det}}; P_{t-\det-f-uns} = \frac{N_{e-nrec-f-uns}}{N_{e-\det}}; P_{f-uns-rec} = C_{e-\det}\times P_{t-\det-f-uns}; \qquad (2)$$

$$P_{f-s} = C_{e-\det}\times P_{t-\det-f-s}; P_{f-uns} = P_{f-uns-\det} + P_{f-uns-rec}; C_e = C_{e-\det}\times C_{e-rec};$$
$$N_e = N_{e-\det} + N_{e-esc-\det}; N_{e-\det} = N_{e-rec} + N_{e-nrec-f-s} + N_{e-nrec-f-uns} \qquad (3)$$

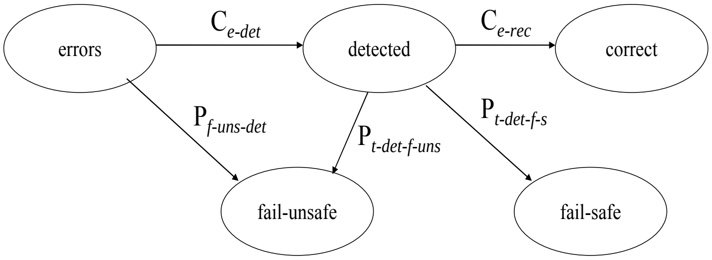

errors
$C_{e\text{-}det}$
detected
$C_{e\text{-}rec}$
correct
$P_{f\text{-}uns\text{-}det}$
$P_{t\text{-}det\text{-}f\text{-}uns}$
$P_{t\text{-}det\text{-}f\text{-}s}$
fail-unsafe
fail-safe

Fig. 8: Predicate graph of fault-tolerant mechanism.

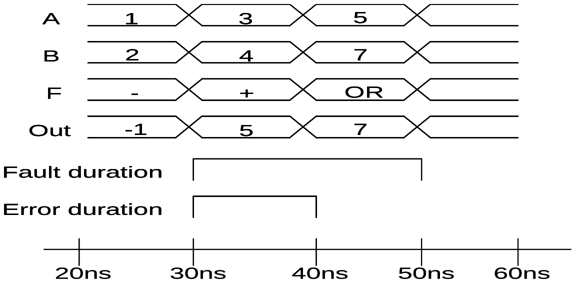


Fig. 9: Fault duration and error duration for an ALU example.

Fig. 8 illustrates the error handling process in our fault-tolerant system. From Fig. 8, if errors occur, the system could enter one of the following states: ‘correct’, ‘fail-safe’ and ‘fail-unsafe’ states.

C. *Experimental Setup*

Three benchmarks including N! (N = 10), 5×5 matrix multiplication, and $2\sum_{i=1}^{5} A_i \times B_i$ have been applied to generate the testbench by copying each benchmark program four times and then combine the twelve programs in random sequence. We first perform the fault simulation based on VHDL simulation platform comprising a simulated fault injection tool, ModelSim VHLD simulator and data analyzer. The common rules of fault injection campaigns are: 1) value of a fault is selected randomly from the s-a-1 and s-a-0; 2) injection targets cover the entire ‘EXE’ stage as shown in Fig. 7. The common data of fault injection parameters are: α=1 (useful-life), failure rate (λ) = 0.001, probability of permanent fault occurrence = 0, fault duration = 5 clock cycles. To generate various fault scenarios, we inject 100, 500, 1000, 1500 and 2000 faults for each injection campaign to represent from slight to serious fault environments. Likewise, various error scenarios are also generated similarly. There are still two parameters required to be determined; one is the P_{f-to-e} and the other is error duration. For the former we give an initial value 0.6. For the latter, each error will choose one value between one and four clocks. The reason is described as follows:

Fig. 9 shows an ALU waveform and the notations ‘A’, ‘B’ and ‘F’ represent the inputs involving two operands and specified operation respectively, and ‘Out’ represents the operating result. As exhibited in Fig. 9, a fault is injected into the adder at 30 ns; therefore, the operating result is incorrect because the addition operation is affected by this fault. However, the OR operation is still correct because no fault is injected into the logical operator. This situation causes the un-equivalence between the fault duration and error duration. It is worthy to note that the error duration will be always equal to or smaller than the fault duration. That’s why we don’t apply the error duration to a constant.

D. *Analysis and Discussion of Simulation Results*

Table 1 illustrates the difference in error coverage between VHDL and System C simulation results. The difference is represented as the deviation calculated by the following expression:

$$Deviation\,(\%) = \frac{SystemC - VHDL}{VHDL} \times 100\% \qquad (4)$$

Table 1: The deviation between SystemC and VHDL design environments.

	VHDL SystemC	100 60	500 300	1000 600	1500 900	2000 1200
$C_{e\text{-}det}$	SystemC	0.992247	0.990883	0.985245	0.982231	0.964229
	VHDL	0.9931	0.992339	0.987017	0.986573	0.975104
	Deviation (%)	-0.08591	-0.14667	-0.17951	-0.44011	-1.11521
$C_{e\text{-}rec}$	SystemC	1	0.999108	0.997564	0.986366	0.970024
	VHDL	1	0.998196	0.99639	0.988165	0.98188
	Deviation (%)	0	0.091351	0.117847	-0.18211	-1.20745
C_e	SystemC	0.992247	0.989999	0.982845	0.968839	0.935326
	VHDL	0.9931	0.990549	0.983453	0.974897	0.957435
	Deviation (%)	-0.08591	-0.05546	-0.06188	-0.62142	-2.30919
$P_{f\text{-}s}$	SystemC	0	0.000884	0.0024	0.013392	0.028904
	VHDL	0	0.00179	0.003564	0.011676	0.017669
	Deviation (%)		-50.6293	-32.6439	14.69876	63.58229
$P_{f\text{-}uns}$	SystemC	0.007753	0.009117	0.014755	0.017769	0.035771
	VHDL	0.0069	0.007661	0.012983	0.013427	0.024896
	Deviation (%)	12.36416	18.99768	13.64725	32.33781	43.67944

Several notable points are observed from Table 1; firstly, the deviations of $Ce_{-\det}$, Ce_{-rec} and Ce are all very small and the maximum is only about 2.3%. This means that the simulation results based on the fault and error injections are quite similar. Nevertheless, the deviations of Pf_{-s} and Pf_{-uns} are very large. It is because the value itself is too small, so a slight difference can cause a serious influence to the deviation; secondly, the deviation rises as the number of injected faults and errors increases especially when the number of faults is greater than 1000. To further understand the influence of error duration and Pf_{-to-e} on deviation, we conduct two additional experiments. One focuses on the former and the other focuses on the latter.

E. *Error Simulation with Various Error Durations*

Table 2: Error duration from 2 to 5 based on *300* errors injected using SystemC platform.

	2	3	4	5	Max diff.
$C_{e\text{-}det}$	0.992959	0.992094	0.990883	0.98858	0.004379
$C_{e\text{-}rec}$	1	0.999545	0.999108	0.998513	0.001487
C_e	0.992959	0.991643	0.989999	0.98711	0.005849
$P_{f\text{-}s}$	0	0.000451	0.000884	0.00147	0.00147
$P_{f\text{-}uns}$	0.007041	0.007906	0.009117	0.01142	0.004379

Table 3: Error duration from 2 to 5 based on *900* errors injected using SystemC platform

	2	3	4	5	Max diff.
$C_{e\text{-}det}$	0.991244	0.988211	0.982231	0.97626	0.014984
$C_{e\text{-}rec}$	1	0.991065	0.986366	0.981624	0.018376
C_e	0.991244	0.979381	0.968839	0.95832	0.032924
$P_{f\text{-}s}$	0	0.00883	0.013392	0.01794	0.01794
$P_{f\text{-}uns}$	0.008756	0.011789	0.017769	0.02374	0.228644

Tables 2 and 3 illustrate the influence of error duration under slight (300 errors injected) and serious (900 errors injected) error scenarios on the error coverage. We observe that values in the *Max diff.* (maximal difference) field of Table 2 are all smaller than those in Table 3. This means that the influence of the error duration becomes higher when the error environment becomes worse.

F. *Error Simulation with Various* Pf_{-to-e}

Table 4: $P_{f\text{-}to\text{-}e}$ = 0.5, 0.55, and 0.6 based on *500* faults experiments.

	250 errors	275 errors	300 errors	Max-Min	500 faults
$C_{e\text{-}det}$	0.991541	0.991182	0.990883	0.000658	0.992339
Deviation (%)	-0.080416	-0.116593	-0.146724		0
$C_{e\text{-}rec}$	0.999545	0.99913	0.999108	0.000437	0.998196
Deviation (%)	0.135144	0.093569	0.91365		0
C_e	0.99109	0.990321	0.989999	0.001091	0.990549
Deviation (%)	0.054616	-0.023018	-0.055525		0
$P_{f\text{-}s}$	0.000451	0.000862	0.000884	0.000433	0.00179
Deviation (%)	-74.804469	-51.843575	-50.614525		0
$P_{f\text{-}uns}$	0.008459	0.008818	0.009117	0.000658	0.007661
Deviation (%)	10.416395	15.102467	19.005352		0

Table 5: $P_{f\text{-}to\text{-}e}$ = 0.5, 0.55, and 0.6 based on *1500* faults experiments.

	750 errors	825 errors	900 errors	Max-Min	1500 faults
$C_{e\text{-}det}$	0.989292	0.984749	0.982231	0.007061	0.986573
Deviation (%)	-0.2756	0.184882	0.440109		0
$C_{e\text{-}rec}$	0.9934	0.988222	0.986366	0.007034	0.988165
Deviation (%)	-0.529770	-0.005768	0.182055		0
C_e	0.982763	0.973151	0.968839	0.013924	0.974897
Deviation (%)	-0.806854	0.179096	0.621399		0
$P_{f\text{-}s}$	0.006529	0.011598	0.013392	0.006863	0.011676
Deviation (%)	44.081877	0.668037	-14.969814		0
$P_{f\text{-}uns}$	0.010708	0.015251	0.017769	0.007061	0.013427
Deviation (%)	20.250242	-13.584568	-32.337827		0

Tables 4 and 5 illustrate the influence of various Pf_{-to-e} on the error coverage. From Tables 4 and 5, we observe the similar phenomenon as shown in Tables 2 and 3.

Summarizing the four tables, we derive an important conclusion: error duration and Pf_{-to-e} are two key factors, which lead to the difference between fault and error simulation results. Furthermore, the influence of error duration on the simulation results is greater than Pf_{-to-e}.

V. CONCLUSIONS

From the simulation results, we recommend the following experimental rules during the fault/error injection campaigns:

1. Do not assign the error duration to a constant value. The error duration for each error should be a random value selected from a range of values, for example, one to four clock cycles for our injection campaigns. Normally, the error duration should be equal to or less than the fault duration.
2. Pf_{-to-e} won't be a constant. It should be adjusted as the degree of fault or error overlapping varies.
3. The various fault/error environments will affect the deviation between different platforms. Worse environment will cause a greater deviation between VHDL and SystemC.

With these rules, designers can set the suitable error duration and P_{f-to-e} for their own error simulation to gain a better quality of simulation results.

ACKNOWLEDGEMENTS

The authors would like to thank Shang-Yu Tao for his help in generating experimental results and acknowledge the support of the National Science Council, Republic of China, under Contract No. NSC 95-2221-E-216-015.

REFERENCES

[1] C. Constantinescu, "Impact of Deep Submicron Technology on Dependability of VLSI Circuits," *IEEE Intl. Conf. On Dependable Systems and Networks (DSN'02)*, pp. 205-209, 2002.

[2] P. Shivakumar et al., "Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic," *DSN'02*, pp. 389-398, 2002.

[3] T. Karnik, P. Hazucha, and J. Patel, "Characterization of Soft Errors Caused by Single Event Upsets in CMOS Processes," *IEEE Trans. on Dependable and Secure Computing*, Vol. 1, No. 2, pp. 128-143, April-June 2004.

[4] D. M. Blough et al., "Fault Tolerance in Super-scalar and VLIW Processors," *IEEE Workshop on Fault Tolerant Parallel and Distributed Systems,* pp. 193-200, 1992.

[5] N. Saxena et al., "Error Detection and Handling in a Superscalar, Speculative Out-of-Order Execution Processor System," *25th IEEE FTCS*, pp. 464-471, 1995.

[6] A. P. Pawlovsky and M. Hanawa, "A Concurrent Fault Detection Method for Superscalar Processors," *IEEE ATS'92*, pp.139-144, 1992.

[7] N. Quach, "High Availability and Reliability in The Itanium Processor," *IEEE Micro*, Vol. 20, issue: 5, pp. 61-69, September-October 2000.

[8] Ward, P.C.; Armstrong, J.R.;" Behavioral fault simulation in VHDL", Design Automation Conference, 1990. Proceedings. 27th ACM/IEEE 24-28 June 1990 Page(s):587 – 593

[9] A. L. White, "Transient Faults and Network Reliability", *IEEE Aerospace Conference*, pp. 78-83, 2004.

[10] Ejlali, A.; Miremadi, S.G.; Zarandi, H.; Asadi, G.; Sarmadi, S.B.; "A Hybrid Fault Injection Approach Based on Simulation and Emulation Co-operation", Dependable Systems and Networks, 2003. Proceedings. 2003 International Conference on 22-25 June 2003 Page(s):479 – 488.

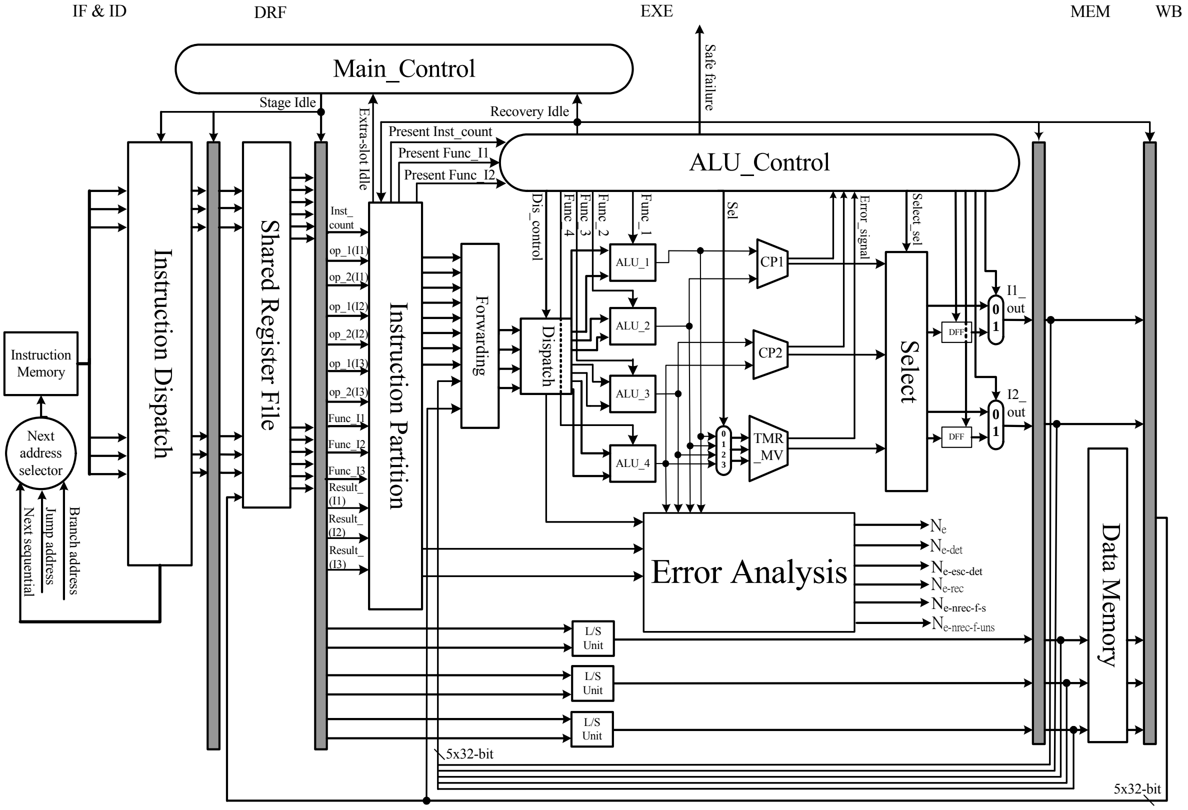
[11] Open SystemC Initiative (OSCI), "SystemC 2.0 Language Reference Manual", Revision 1.0, www.systemc.org, 2003.

[12] Rothbart, K.; Neffe, U.; Steger, Ch.; Weiss, R.; Rieger, E.; Muehlberger, A.;" High level fault injection for attack simulation in smart cards", Test Symposium, 2004. 13th Asian 15-17 Nov. 2004 Page(s):118 – 121

[13] A. Fin, F. Fummi, G. Pravadelli, "AMLETO: a Multilanguage environment for functional test generation", Test Conference, 2001. Proceedings. International , 30 Oct.-1 Nov. 2001, Pages:821 – 829.

[14] S. Chiusano, S. Di Carlo, P. Prinetto, "Automated synthesis of SEU tolerant architectures from OO descriptions", On-Line Testing Workshop, 2002. Proceedings of the Eighth IEEE International , 8-10 July 2002, Pages:26 – 31.

[15] Yung-Yuan Chen, Kuen-Long Leu, and Chao-Sung Yeh, "Fault-Tolerant VLIW Processor Design and Error Coverage Analysis", International Conference, EUC 2006 Seoul, Korea, August 2006 Proceeding, pp. 754-765.



Note: CP (ComParator), MV (Majority Voter)

Fig. 7: Fault-tolerant VLIW architecture.

行政院國家科學委員會補助國內專家學者出席國際學術會議報告

95 年 08 月 17 日

附件三

報告人姓名	陳永源	服務機構 及職稱	中華大學資訊工程學系 副教授
會議 時間 地點	08 月 01-04, 2006 韓國首爾	本會核定 補助文號	NSC 95-2221-E-216-015
會議 名稱	（中文） （英文）**The 2006 IFIP International Conference on Embedded And Ubiquitous Computing**		
發表 論文 題目	（中文） （英文）Fault-Tolerant VLIW Processor Design and Error Coverage Analysis		

報告內容應包括下列各項：

一、　參加會議經過

此會議是在韓國首爾舉行，作者是搭華航班機到首爾。此會議共有110篇regular papers發表，錄取率低於25%，發表的論文收錄在Lecture Notes in Computer Science，為一SCI indexed的會議。論文的主題範圍包括了power aware computing, Security and fault tolerance, agent and distributed computing, wireless communication, real-time systems, embedded software optimization, embedded architecture, mobile computing and embedded systems。參加的學者來自美國，台灣，韓國，大陸以及歐洲的國家。作者的論文被安排在第三天下午報告，講題是 “Fault-Tolerant VLIW Processor Design and Error Coverage Analysis”。當天晚上參加會議所舉辦的晚宴，欣賞了韓國的傳統舞蹈，並與來自西班牙、韓國、大陸以及蘇俄的學者聊天交換研究心得。

二、　與會心得

此會議為一 SCI indexed 的會議，錄取率低於 25%，發表的論文收錄在 Lecture Notes in Computer Science。所以其錄取的論文是經過一嚴格的評審，來達到一高品質的會議。可以透過此會議與其他國家的學者討論交流並且掌握最新的研究題材與研究結果，可以用來檢視作者目前及未來的研究方向與課題的價值性，對於以後的研究有相當的幫助。另外也有機會請教一些國際級的學者，傾聽他們對一些議題的意見及看法，可以幫助作者對一些困惑的地方及觀念做一釐清，對於往後的研究也是有相當的幫助。研究心得是未來輻射線粒子對於深次微米製程的晶片影響力越來越大，造成暫時性錯誤的機率也 越來越高，此問題將會影響處理器晶片的可靠度。所以有幾個問題值得進一步的探討(針對深次微米製程的晶片)：fault model 的完整性，容錯技術的有效性，灌錯及錯誤模擬分析工具環境的建立，系統驗證分析等等。

三、　攜回資料名稱及內容

一本會議的論文集

Fault-Tolerant VLIW Processor Design and Error Coverage Analysis

Yung-Yuan Chen, Kuen-Long Leu, and Chao-Sung Yeh

Department of Computer Science and Information Engineering
Chung-Hua University, Hsin-Chu, Taiwan
chenyy@chu.edu.tw

Abstract. In this paper, a general fault-tolerant framework adopting a more rigid fault model for VLIW data paths is proposed. The basic idea used to protect the data paths is that the execution result of each instruction is checked immediately and if errors are discovered, the instruction retry is performed at once to overcome the faults. An experimental architecture is developed and implemented in VHDL to analyze the impacts of our technique on hardware overhead and performance degradation. We also develop a comprehensive fault tolerance verification platform to facilitate the assessment of error coverage for the proposed mechanism. A paramount finding observed from the experiments is that our system is still extremely robust even in a very serious fault scenario. As a result, the proposed fault-tolerant VLIW core is quite suitable for the highly dependable real-time embedded applications.

1 Introduction

In recent years, VLIW processor has become a major architectural approach for high-performance embedded computing systems. Several notable examples of VLIW are Intel and HP IA-64 [1], TI TMS320C62x/67x DSP devices and Fujitsu FR500. As processor chips become more and more complicated, and contain a large number of transistors, the processors have a limited operational reliability due to the increased likelihood of faults or radiation-induced soft errors especially when the chip fabrication enters the deep submicron technology [2]. Also indicated specifically in [3], it is expected that the bit error rate in a processor will be about ten times higher than in a memory chip due to the higher complexity of the processor. And a processor may encounter a bit flip once every 10 hours. Thus, it is essential to employ the fault-tolerant techniques in the design of high-performance superscalar or VLIW processors to guarantee a high operational reliability in critical applications. Recently, the reliability issue in high-end processors is getting more and more attention [3-9].

The previous researches in reliable microprocessor design are mainly based on the concept of time redundancy approach [3-9] that uses the instruction replication and recomputation to detect the errors by comparing the results of regular and duplicate instructions. The instruction replication, recomputation schedule and result comparison of regular and duplicate instructions can be accomplished either in software level – source code compilation phase to generate redundant code for fault detection [4], [7], [8] or in hardware level [3], [5], [6], [9]. In [7], [8], the authors adopted software

E. Sha et al. (Eds.): EUC 2006, LNCS 4096, pp. 754 – 765, 2006.

techniques for detecting the errors in superscalar and VLIW processors respectively. The compiler-based software redundancy schemes have the advantage of no hardware modifications required, but the performance degradation and code growth increase significantly as pointed out in [3], [5]. The hardware redundancy approach requires extra hardware and architectural modification to manage the instruction replication, recomputation and comparison to detect the errors.

The deficiencies in previous studies are summarized as follows. First, most of the studies in the literature focus only on the aspect of error detection and neglect the issue of error recovery; thereby, those designs are incomplete so that we have difficulty in investigating the effectiveness of the error detection scheme without considering the error recovery jointly. Second, they lack the precise evaluation of the hardware overhead caused by the incorporation of fault tolerance; therefore, it is hard to justify the soundness of the approaches. Thirdly, the performance degradation due to the error detection and error recovery is significant during program execution. Moreover, the performance analysis only takes the performance degradation resulting from the fault detection into account. They are short of the analysis of error recovery time demanded to overcome the transient faults. The error recovery time mainly depends on the error-detection latency, which can be calculated from the time of regular instruction execution to the time of duplicate instruction recomputation. Owing to variable latency, the analysis of latency effect on performance is quite involved, and therefore, it complicates the analysis of the impact of error recovery on performance. Further, the latency may be unacceptably long. If an error cannot be detected in a short time, it will increase the error recovery time as well as program execution time. Such a lengthy recovery may be detrimental to the real-time applications. Last but not least, the previous studies rarely perform the quantitative evaluation of error coverage and the probability of common-mode failures [10] for the systems in various fault environments. Thus, it is hard to validate the fault tolerance ability of the schemes due to lack of the measures of error coverage.

This work is going to address the issues stated above. In Section 2, a fault-tolerant approach concentrating on the dependable data path design of VLIW processors is proposed. The approach proposed is quite comprehensive in that it comprises the error detection and error recovery. Hardware architecture and the measurements of hardware overhead and performance degradation are presented in Section 3. In Section 4, a thorough error coverage analysis is conducted to validate our scheme. The conclusions appear in Section 5.

2 Fault-Tolerant Data Path Design

Two types of faults described below are addressed in the error detection and error recovery: 1. Correlated transient faults [11] (e.g., a burst of electromagnetic radiation) which could cause multiple module failures. 2. Near-coincident faults [12] – recovery can be affected by this kind of faults. It is evident that the adopted fault model in this study is more rigid and complete compared to the single-fault assumption commonly applied before. Besides the concern of the fault model, an important goal for the design of error-recovery process is to simplify its complexity and meanwhile achieve the time efficiency to recover the errors. Overall, the design concern here is to propose a fault-tolerant VLIW core for the highly dependable real-time embedded applications. However, we note that due to the more rigid fault model and severe fault situations considered, it requires developing a more powerful fault-tolerant scheme to raise the system reliability to a sound level.

A VLIW processor core may possess several different types of functional modules in the data paths, such as integer ALU and load/store units. A couple of identical modules are provided for a specific functional type. We assume that the register file is protected by an error-correcting code. In the following, we present the main ideas employed in our scheme to detect and recover errors occurring in the data paths and then use three identical modules to demonstrate our fault-tolerant approach.

2.1 Concurrent Error Detection and Real-Time Error Recovery

We note that the length of error recovery time mainly depends on the error-detection latency. Hence, the error-detection scheme has a significant impact on the efficiency of the error recovery. Most of the previous studies may suffer the lengthy error recovery because the execution results of each instruction cannot be checked immediately. Therefore, to achieve the real-time error recovery, the execution results of each instruction must be examined immediately and if errors are found, the erroneous instruction is retried at once to overcome the errors. So, the error-detection problem can be formalized as how to verify the execution results instantly for each instruction, i.e. how to achieve no error-detection latency. We develop a simple concurrent error-detection (CED) scheme, which combines the duplication with comparison, henceforth referred to as comparison, and majority voting methodologies to solve the above error-detection problem.

CED Scheme. The following notations are developed

- n : Number of identical modules for a specific functional type (we call it type x). n is also the maximum number of instructions that can be executed concurrently in the modules of type x;
- s : Number of spare modules added to the type x, $s \geq 0$;
- m : Number of instructions in an execution packet for type x, $m \leq n$.

An execution packet is defined as the instructions in the same packet can be executed in parallel. There are $n+s$ modules for type x. As we know, if $m \times 2 > n+s$ then it is clear that the system won't have the enough resources to check the instructions of an execution packet concurrently. Under the circumstances, the current execution packet needs to be partitioned into several packets that will be executed sequentially. Given an execution packet, there are three cases to consider:

Case 1: $m \times 2 = n+s$. In this case, each instruction can be checked by the comparison scheme.

Case 2: $m \times 2 < n+s$. We can divide the instructions into two groups: G(1) and G(2). There are $m1$ instructions and $m2$ instructions in G(1) and G(2) respectively, where $m1 + m2 = m$, $m1, m2 \geq 0$. Each instruction in G(1) and G(2) can be examined by the

triple modular redundancy (TMR) scheme and duplication with comparison, henceforth referred to as comparison, scheme respectively. It is worth noting that to deal with the correlated transient faults, which may cause the multiple module failures, the TMR scheme is enhanced to have the ability to detect the multiple module errors. The following equations and criterion are used to decide $m1$ and $m2$. The equations are $m1 \times 3 + m2 \times 2 \leq n + s$; $m1 + m2 = m$; $m1, m2 \geq 0$. There may have several solutions derived from the equations. Since TMR can tolerate and locate one faulty module compared to the comparison, the criterion employed is to choose a solution which has the maximal value of m_1 among the feasible solutions. In other words, TMR has the benefit to avoid activating the procedure of error recovery while only one faulty module occurs. In contrast to TMR, comparison scheme needs to spend time for error recovery. The concern here is again the consideration of real-time applications.
Case 3: $m \times 2 > n + s$. Due to limited resources, m instructions cannot be all checked at the same cycle by TMR and/or comparison schemes. Therefore, we need to partition m instructions into several sequential execution packets such that the instructions in each packet can be examined concurrently. However, some extra cycles are required to guarantee that each instruction can be verified while it is executed. This implies that the performance of program execution will be degraded. The degree of performance degradation depends on the occurring frequency of the Case 3 during the program execution. The compromise between hardware overhead and performance degradation can be accomplished by choosing a proper s .

In general, the performance degradation for program execution in our dependable VLIW processor stems mainly from two sources: first is the extra cycles demanded for detecting the errors; second is the time for error recovery in order to overcome the effect of errors in the system. The error-recovery scheme is presented next.

Error-Recovery Scheme. Since each instruction is executed and verified at the same time, the instruction retry can be adopted to overcome the errors in an effective manner. When control unit of data paths receives the abnormal signals from the detection circuits, the procedure of error recovery will be activated immediately to recover the erroneous instructions. The following notations are used to explain the proposed error-recovery scheme:

- $m_x(i)$: The *ith* module of type x, where $1 \leq i \leq n + s$;
- TMR($m_x(i)$, $m_x(j)$, $m_x(k)$): TMR using $m_x(i)$, $m_x(j)$, $m_x(k)$, where $i \neq j \neq k$. In the following, the term of TMR($m_x(i)$, $m_x(j)$, $m_x(k)$) is abbreviated to TMR_x(*i*, *j*, *k*);
- *r_no*: Number of retries permitted for an incorrect instruction, where $r\_no > 0$.

During the error recovery, each erroneous instruction is retried individually with the TMR scheme. We allow performing *r_no* retries for an instruction to conquer the errors before declaring fail-safe. Since TMR scheme represented as TMR_x(*i*, *j*, *k*) is employed for the instruction retry, an issue arises as how to determine the (*i*, *j*, *k*) for each retry. As we know, there are $\binom{n+s}{3}$ combinations of (*i*, *j*, *k*). Let S_TMR be a set that contains $\binom{n+s}{3}$ combinations of TMR_x(*i*, *j*, *k*). Hence, S_TMR can be represented as {TMR_x(1, 2, 3), …, TMR_x(1, 2, $n+s$), …, TMR_x(1, $n+s-1$, $n+s$), TMR_x(2, 3, 4), …, TMR_x(2, $n+s-1$, $n+s$), …, TMR_x($n+s-2$, $n+s-1$, $n+s$)}, where $n+s \geq 3$. It is clear that selecting the TMR_x(1, 2, 3) constantly for each retry, for example, is the simplest approach, which has the advantage of simple implementation but can only tolerate one faulty module during the recovery process. In contrast to that, selecting elements one by one based on the element sequence in S_TMR for the retries is the highly complicated approach. Such an approach suffers from the high implementation cost, but on the other hand it can tolerate $n+s-2$ faulty modules if we set $r\_no \geq \binom{n+s}{3}$. The remaining question in the design of selection policy for TMR retry is how to compromise between the implementation complexity and the number of faulty modules being tolerated. A sound selection policy for TMR retry is presented next.

Selection Policy. On the basis of the above discussion, a set named SS_TMR, a subset of S_TMR, is created to guide the instruction-retry process. SS_TMR is given below: SS_TMR= {TMR_x(i, $i+1$, $i+2$), where $1 \leq i \leq n+s-2$ }. As seen from SS_TMR, the proposed retry process possesses a high regularity in its selection policy. So, it is easy to implement the SS_TMR policy compared to the S_TMR.

After the analyses for some values of *n* and *s*, we decide to adopt the SS_TMR selection policy due to the following reasons: first, we note that the probability of three or more modules failed concurrently should be low; second, most of the faults are transient type, which may disappear during the recovery process; and last one is the low implementation complexity compared to the S_TMR policy. From the first two reasons, we can infer that both selection policies have the similar fault tolerance capabilities. It is evident that the SS_TMR selection policy can utilize the module resources efficiently so as to recover the errors in a short time. Thus, the program execution can continue without lengthy error-recovery process. In summary, our error-recovery scheme can provide the capability of real-time error recovery, which is particularly important for the applications demanding the reliable computing as well as real-time concern.

2.2 Reliable Data Path Design: Case Study

In the following illustration, without loss of generality, we assume only one type of functional module, namely ALU, in the data paths. In this case study, the original VLIW core contains three ALUs ($n = 3$) and therefore, three ALU instructions can be issued at most per cycle. A spare ALU ($s = 1$) is added to prevent the severe performance degradation as explained below. From CED scheme described in Section 2.1, we note that if no spare is added then $m = 2$ or 3 execution packets will fall into Case 3. Consequently, the performance may be degraded significantly. Hence, the cost of a spare is paid to lower the performance degradation. Clearly, adding three spares in order to eliminate the performance degradation completely is not a feasible choice.

According to CED scheme with $n = 3$ and $s = 1$, $m = 1$ falls into Case 2. The ($m1, m2$) can be (1, 0) or (0, 1). Clearly, (1, 0) is selected as the final solution. So, if an execution packet contains only one ALU instruction then it will be checked by TMR scheme. For $m = 2$, it is Case 1. Each instruction will be checked by comparison scheme. For $m = 3$, it is Case 3. The three concurrent ALU instructions need to be scheduled to two sequential execution packets where one packet contains two instructions and the other holds the rest one; and therefore, one extra ALU cycle is required to complete the execution of three concurrent ALU instructions for error-detection purpose.

CED Process. Given $n = 3$ and $s = 1$, the notation CMP_ALU(i, j) is used to denote an instruction executed with the comparison scheme using the *ith* and *jth* ALUs.

```
while (not end of program)
  {switch ($m$)
    {case '1':
         TMR_ALU(1, 2, 3); if (TMR_ALU detects more than one
         ALU failure) then the "Error-recovery process" is
         activated to recover the failed instruction.
     case '2':
         the execution packet contains two instructions:   $I_1$
         and $I_2$.
         $I_1$: CMP_ALU(1, 2);   $I_2$: CMP_ALU(3, 4);
         if ($I_1$ fails) then the "Error-recovery process" is
         activated to recover $I_1$.
         if ($I_2$ fails) then the "Error-recovery process" is
         activated to recover $I_2$.
     case '3':
         the packet is divided to two packets and executed
         sequentially.
     }}
```

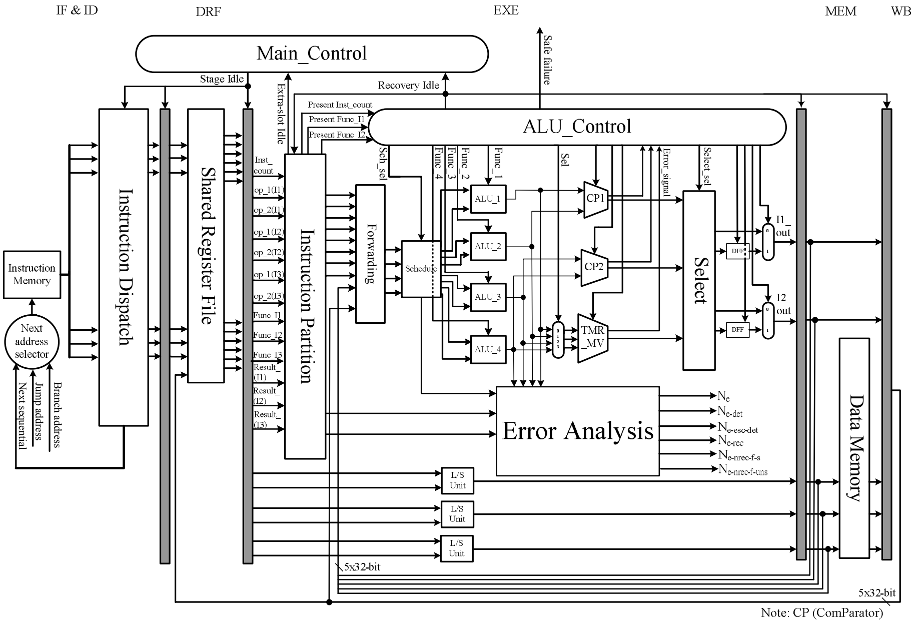
Error-recovery process:

```
    $i \leftarrow 1$;
    While ($r\_no > 0$)
     {TMR_ALU($i$, $i+1$, $i+2$);
      if (TMR_ALU succeeds) then the error recovery succeeds
      → exit;
      else { $r\_no \leftarrow r\_no - 1$ ; $i \leftarrow i+1$ ; if ( $i \geq 3$ ) then
    $i \leftarrow 1$;}}
    recovery failure and the system enters the fail-safe
    state.
```

3 Hardware Implementation and Performance Evaluation

To validate the proposed approach, an experimental fault-tolerant VLIW architecture based on the scheme presented in Section 2.2 is developed. Figure 1 illustrates the architecture implementation, where $n = 3$, and $s = 1$ for ALUs. The features of this 32-bit VLIW processor are stated as follows: • the instruction set is composed of twenty-five 32-bit instructions; • each ALU includes a 32x32 multiplier. For simplicity of demonstration, the proposed approach does not apply to the load/store units; • a register file containing thirty-two 32-bit registers with 12 read and 6 write ports is shared with modules and designed to have bypass multiplexors that bypass written data to the read ports when a simultaneous read and write to the same entry is commanded; • data memory is 1K x 32 bits. The structure consists of five pipeline stages: 'instruction fetch and dispatch', 'decode and operand fetch from register file', 'execution', 'data memory reference' and 'write back into register file' stages. This experimental architecture can issue at most three ALU and three load/store instructions per cycle. Note that the 'Error Analysis' block in execution stage, which was created only to facilitate the measurement of the error coverage during the fault injection campaign, is not a component for the VLIW processor displayed in Figure 1.

A fault-tolerant VLIW processor based on the architecture of Figure 1 and the features mentioned previously was realized in VHDL. The implementation data by UMC 0.18μm process are shown in Table 1. The area does not include the instruction memory as well as the 'Error Analysis' block. For performance consideration, we require that the clock frequency of the fault-tolerant VLIW processor must retain the same as that of non fault-tolerant one. It is worth noting that the overhead of 'ALU_Control' unit is only 0.26 percent compared to the area of the non fault-tolerant VLIW core. This implies that the control task of our scheme is simple and easy to implement. The performance degradation caused from the CED demand is between 0.6% and 34.3% for eight benchmark programs, including heapsort, quicksort, FFT, 5×5 matrix multiplication and IDCT (8×8) etc..



Note: CP (ComParator)

Fig. 1. Fault-tolerant VLIW architecture

Table 1. Comparing our approach with non fault-tolerant VLIW core

	Area (μm^2)	Overhead	ALU_Control(μm^2)	System clock (MHz)
Non fault-tolerant VLIW	9319666			128
Our approach	10708296	14.9%	24215	128

4 Error Coverage Analysis

In this section, the error coverage analysis based on the fault injection [13] is conducted to validate our scheme. A comprehensive fault tolerance verification platform comprising a simulated fault injection tool, ModelSim VHDL simulator and data analyzer has been built. It offers the capability to effectively handle the operations of fault injection, simulation and error coverage analysis. The core of the verification platform is the fault injection tool that can inject the transient and permanent faults into VHDL models of digital systems at chip, RTL and gate levels during the design phase. The tool adopts the built-in commands of VHDL simulators to inject the faults into VHDL simulation models. Injection tool can inject the following classes of faults: ‘0’ and ‘1’ stuck-at faults, ‘Z’: high-impedance and ‘X’: unknown faults. Weibull fault distribution is employed to decide the time instant of fault injection.

Our tool supports a fault injection analysis, which can provide us the useful statistics for each injection campaign. The statistical data for each injection campaign represents a fault scenario. We can exploit the injection tool to produce a variety of fault scenarios such that the fault-tolerant systems can be thoroughly validated. The injection tool can assist us in creating the proper fault environments that can be used to effectively validate the capability of a fault-tolerant system and examine the strength of a fault-tolerant system under various fault scenarios. Therefore, the proposed verification platform helps us raise the efficiency and validity of dependability analysis.

4.1 Fault-Tolerant Design Metrics

Figure 2 illustrates the error handling process in our fault-tolerant system. CED scheme uses the comparison and TMR to detect the errors. Hence, the following types of errors will escape being detected and such detection defects will result in the unsafe failures (or called common-mode failures [10]): one is the two ALUs produce the same, erroneous results to comparator; another is two or three of ALUs produce the identical, erroneous results to TMR. Once errors are detected and need to be recovered, the error-recovery process is activated. Three possible outcomes could happen for each instruction retry using TMR scheme. One possibility is that the recovery is successful; another is retry fails and the system enters the fail-safe state; the last possibility is two or three of ALUs produce the identical, erroneous results to TMR such that the system encounters the fail-unsafe hazard. From Figure 2, if errors happen, the system could enter one of the following states: ‘successful recovery and restore the normal operation’, ‘fail-safe’ and ‘fail-unsafe’ states.

The design metrics as described below are exploited to justify our approach:

- P_{f-uns} : Probability of system entering the fail-unsafe state;
- $C_{e-\det}$: Error-detection coverage, i.e. probability of errors detected;
- C_{e-rec} : Error-recovery coverage, i.e. probability of errors recovered given errors detected;
- C_e : Error coverage, i.e. probability of errors detected and recovered;
- P_{f-s} : Probability of system entering the fail- safe state;
- $P_{t-\det-f-s}$: State transition probability from 'detected' state to 'fail-safe' state.
- $P_{t-\det-f-uns}$: State transition probability from 'detected' state to 'fail-unsafe' state.
- $P_{f-uns-\det}$: Probability of system entering the fail-unsafe state due to the detection defects stated earlier;
- $P_{f-uns-rec}$: Probability of system entering the fail-unsafe state due to the recovery defects stated earlier;

The parameters N_e , $N_{e-\det}$, $N_{e-esc-\det}$, N_{e-rec} , $N_{e-nrec-f-s}$ and $N_{e-nrec-f-uns}$ (called the error-related parameters) represent the total number of errors occurred, the number of errors detected, the number of errors escape being detected, the number of errors recovered, the number of errors not recovered and system enters the 'fail-safe' state and the number of errors not recovered and system enters the 'fail-unsafe' state, respectively. The design metrics can be expressed as follows

$$P_{f-uns-\det} = \frac{N_{e-esc-\det}}{N_e}; C_{e-\det} = \frac{N_{e-\det}}{N_e} = 1 - P_{f-uns-\det}; C_{e-rec} = \frac{N_{e-rec}}{N_{e-\det}}. \quad (1)$$

$$\begin{gathered} P_{t-\det-f-s} = \frac{N_{e-nrec-f-s}}{N_{e-\det}}; P_{t-\det-f-uns} = \frac{N_{e-nrec-f-uns}}{N_{e-\det}}; \\ P_{f-uns-rec} = C_{e-\det} \times P_{t-\det-f-uns}. \end{gathered} \quad (2)$$

$$\begin{gathered} P_{f-s} = C_{e-\det} \times P_{t-\det-f-s}; P_{f-uns} = P_{f-uns-\det} + P_{f-uns-rec}; \\ C_e = C_{e-\det} \times C_{e-rec}; N_e = N_{e-\det} + N_{e-esc-\det}; \\ N_{e-\det} = N_{e-rec} + N_{e-nrec-f-s} + N_{e-nrec-f-uns}. \end{gathered} \quad (3)$$

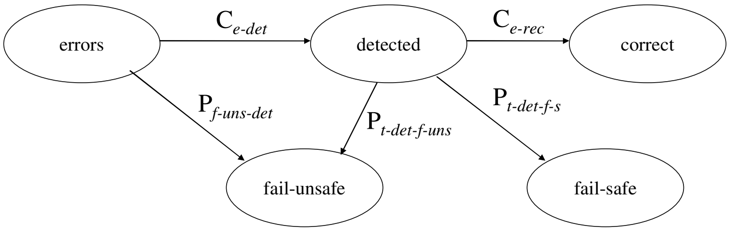

errors
$C_{e\text{-}det}$
detected
$C_{e\text{-}rec}$
correct
$P_{f\text{-}uns\text{-}det}$
$P_{t\text{-}det\text{-}f\text{-}uns}$
$P_{t\text{-}det\text{-}f\text{-}s}$
fail-unsafe
fail-safe

Fig. 2. Predicate graph of fault-tolerant mechanism

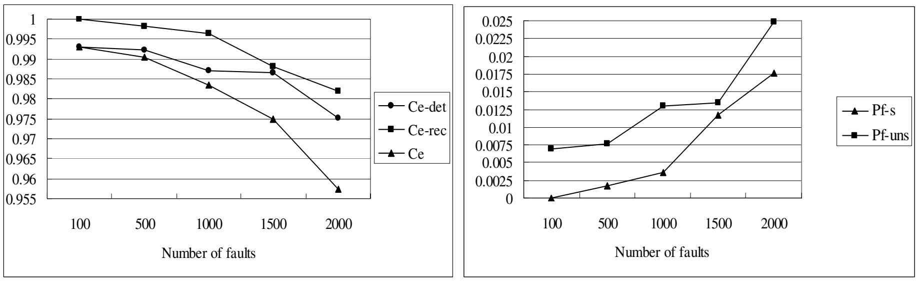
4.2 Simulation Results and Discussion

We have conducted a huge amount of fault injection campaigns to validate the proposed fault-tolerant VLIW scheme under various fault situations. We performed a comprehensive experiment to explore a particular fault-related parameter, namely fault-occurring frequency, to see its impact on the fault-tolerant metrics. By adjusting the fault-occurring frequency, we can create a variety of fault scenarios, which can be used to measure how robust can our fault-tolerant system reach under the different fault environments? The common rules of fault injection campaigns are: 1) value of a fault is selected randomly from the s-a-1 and s-a-0; 2) injection targets cover the entire 'EXE' stage as shown in Figure 1. The common data of fault injection parameters are: $\alpha=1$ (useful-life), failure rate (λ) = 0.001, probability of permanent fault occurrence = 0, fault duration = 5 clock cycles. In addition, the number of retries *r_no* is set to four. Next, we discuss the outcomes obtained from the experiments.

Fault-Occurring Frequency. The goal of this experiment is to observe the effect of the fault-occurring frequency on the design metrics depicted in Section 4.1. In this experiment, we copy each of the following benchmark programs: '*N! (N=10)*', '*5×5 matrix multiplication*', '$2\sum_{i=1}^{5} Ai \times Bi$', four times and then the twelve programs are combined in random sequence to form a workload for the fault simulation. The length of workload is equal to 4384 (clocks) ×30 (ns/clock).

Note that if workload and fault duration are constant, the quantity of faults injected, i.e. fault-occurring frequency, will influence the degree of fault overlap. For instance, while the quantity of faults injected increases, the degree of fault overlap will become more serious. In other words, the various fault-occurring frequencies will lead to the different fault environments. Hence, in order to investigate the effect of the fault-occurring frequency on error coverage, we conduct five fault injection campaigns with various numbers of faults injected. The statistical analysis of an injection campaign is able to disclose the fault activity within the simulation. Clearly, the larger the number of faults injected (i.e. higher fault-occurring frequency), the worse of fault environment will be due to a higher occurring frequency of multiple faults including correlated, mutually independent and near-coincident transient faults. Therefore, the statistical analysis helps designers choose a set of desired fault scenarios to test the ability of fault-tolerant systems. As a result, the proposed fault-tolerant verification platform can furnish more comprehensive and solid error coverage measurements.

Figure 3 characterizes the effect of fault-occurring frequency on the fault-tolerant design metrics. The experimental results obtained have 95% confidence interval of ±0.138% to ±0.983%. The outcomes shown in Figure 3 reveal the fault tolerance capability of our scheme in the various fault environments. It is evident that the error coverage decreases with the increase of fault-occurring frequency. Meanwhile, the system has a higher chance to enter the fail-safe and fail-unsafe states when the probability of occurrence of multiple faults rises. The safe failure occurs once the error-recovery process cannot overcome the errors due to a serious fault situation. Overall, the results presented in Figure 3 are quite positive and sound those declare the effectiveness of our fault-tolerant scheme even in a very bad fault environment.



(a) (b)

Fig. 3. Fault-tolerant metric analysis. (a) coverage. (b) probabilities of fail-safe and fail-unsafe.

5 Conclusions

This paper presents a new fault-tolerant framework for VLIW processors that focuses mainly on the reliable data path design. Based on a more rigid fault model, a CED and real-time error recovery scheme is proposed to enhance the reliability of the data paths. Our approach provides the design compromise between hardware overhead, performance degradation and fault tolerance capability. This framework is quite useful in that it can give the designers an opportunity to choose an appropriate solution to meet their need. Several significant contributions of this study are: 1. Integrate the error detection and error recovery into VLIW cores with reasonable hardware overhead and performance degradation. It is worth noting that the proposed fault-tolerant framework can achieve no error-detection latency and real-time error recovery. Consequently, our scheme is suitable for the real-time computing applications that demand the stringent dependability. 2. Conduct a thorough fault injection campaigns to assess the fault-tolerant design metrics under a variety of fault environments. Importantly, we provide not only the error-detection and error-recovery coverage, but also the fail-safe and fail-unsafe probabilities. Acquiring the fail-unsafe probability is crucial for us to understand how much possibility the system could fail without notice once the errors occur. Moreover, a couple of fault environments, which represent the various degrees of fault's severity, were constructed to validate our scheme so as to realize the capability of our scheme in different fault scenarios. So, such experiments can give us more realistic and comprehensive simulation results. The effectiveness of our mechanism even in a very severe fault environment is justified from the experimental results.

Acknowledgments. The authors acknowledge the support of the National Science Council, Republic of China, under Contract No. NSC 92-2213-E-216-005 and NSC 93-2213-E-216-019.

References

1. Huck, J. et al.: Introducing the IA-64 Architecture. *IEEE Micro*, Vol. 20, issue: 5, pp. 12-23, Sep.-Oct. 2000.
2. Karnik, T., Hazucha, P., Patel, J.: Characterization of Soft Errors Caused by Single Event Upsets in CMOS Processes. *IEEE Trans. on Dependable and Secure Computing*, Vol. 1, issue: 2, pp. 128-143, April-June 2004.

3. Nickle, J. B., Somani, A. K.: REESE: A Method of Soft Error Detection in Microprocessors. *DSN'01*, pp. 401-410, 2001.
4. Holm, J. G., Banerjee, P.: Low Cost Concurrent Error Detection in A VLIW Architecture Using Replicated Instructions. *Intl. Conf. on Parallel Processing*, pp. 192-195, 1992.
5. Franklin, M.: A Study of Time Redundant Fault Tolerance Techniques for Superscalar Processors. *IEEE Intl. Workshop on Defect and Fault Tolerance in VLSI Systems (DFT'95)*, pp. 207-215, 1995.
6. Kim, S., Somani, A. K.: SSD: An Affordable Fault Tolerant Architecture for Superscalar Processors. *Pacific Rim Intl. Symposium. On Dependable Computing*, pp. 27-34, 2001.
7. Oh, N., Shirvani, P. P., McCluskey, E. J.: Error Detection by Duplicated Instructions in Super-Scalar Processors. *IEEE Trans. on Reliability*, Vol. 51, (1), pp. 63-75, March 2002.
8. Bolchini, C.: A Software Methodology for Detecting Hardware Faults in VLIW Data Paths. *IEEE Trans. on Reliability*, Vol. 52, (4), pp. 458-468, December 2003.
9. Qureshi, M. K., Mutlu, O., Patt, Y. N.: Microarchitecture-Based Introspection: A Technique for Transient-Fault Tolerance in Microprocessors. *DSN'05*, pp. 434 – 443, June-July 2005.
10. Mitra, S., Saxena, N. R., McCluskey, E. J.: Common-Mode Failures in Redundant VLSI Systems: A Survey. *IEEE Trans. on Reliability,* Vol. 49, (3), pp. 285 – 295, Sept. 2000.
11. Avizienis, A., Laprie, J.-C., Randell, B., Landwehr, C.: Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. on Dependable and Secure Computing*, Vol. 1, issue: 1, pp. 11-33, Jan.-March 2004.
12. Dugan, J. B., Trivedi, K. S.: Coverage Modeling for Dependability Analysis of Fault-Tolerant Systems. *IEEE Trans. on Computers*, Vol. 38, (6), pp. 775-787, June 1989.
13. Clark, J., Pradhan, D.: Fault Injection: A Method for Validating Computer-System Dependability. *IEEE Computer*, 28(6), pp. 47-56, June 1995.