

Hybrid Error-Detection Approach with No Detection Latency for High-Performance Microprocessors

Yung-Yuan Chen, Kuen-Long Leu and Li-Wen Lin

Department of Computer Science and Information Engineering
Chung-Hua University, Hsin-Chu, Taiwan

Abstract - *Error detection plays an important role in fault-tolerant computer systems. Two primary parameters concerned for error detection are the latency and coverage. In this paper, a new, hybrid error-detection approach offering a very high coverage with no detection latency is proposed to protect the data paths of high-performance microprocessors. The feature of no detection latency is essential to real-time error recovery. The hybrid detection approach is to combine the duplication with comparison, triple modular redundancy (TMR) and self-checking mechanisms to construct a formal framework, which allows the error-detection schemes of varying hardware complexity and performance to be incorporated. We develop three error-detection schemes using the concept of hybrid approach to demonstrate the design compromise among the hardware overhead, performance degradation and error-detection coverage (EDC). Three detection schemes are then implemented in an experimental 32-bit VLIW core respectively. The hardware implementations in VHDL and simulated fault injection experiments are performed to measure the design metrics.*

Keywords: Error detection, error-detection coverage, error-detection latency, hybrid detection approach, performance degradation.

1 Introduction

The rate of radiation-induced soft errors increases rapidly especially in combinational logic while the chip fabrication enters the deep submicron technology [1, 2, 3]. Such influence raises the urgent need to incorporate the fault tolerance into the high-performance microprocessors. Concurrent error detection provides an effective approach to detect the errors caused by transient and intermittent faults [4-7]. One principal concern in the design of error-detection schemes is the error-detection latency, which dominates the time efficiency of the error recovery. The previous researches in reliable microprocessor design are mainly based on the concept of time redundancy approach [5-11] that uses the instruction replication and re-computation to detect the errors by comparing the results of regular and duplicate instructions. The error-detection latency can be calculated from the time of

regular instruction execution to the time of duplicate instruction re-computation. Owing to variable detection 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 detection 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 computing applications. The above discussion brings out that the characteristic of no detection latency is necessary to accomplish the real-time error recovery by simply using the instruction-retry method. In this study, to fulfill the requirement of no detection latency, it demands that the execution results of each instruction must be examined immediately and if errors are found, the erroneous instructions are retried at once to overcome the errors. So, the error-detection problem can be formalized as how to verify the execution results promptly for each instruction. Following that, a new, hybrid error-detection approach is proposed to detect the faults occurring in the data paths while the instructions are executed. Our hybrid detection approach is quite comprehensive in that it offers the design options based on the trade-offs between the hardware redundancy and time redundancy.

2 Hybrid Error-Detection Approach

Two classes of faults described below are particularly addressed in the error detection: 1. Correlated transient faults [12, 13] (e.g., a burst of electromagnetic radiation) which could cause multiple module failures. 2. A single event upset (SEU) could possibly generate a multiple transient fault, which may lead to a bidirectional error at the logic circuit output [14]. 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. However, we note that due to the more rigid fault model and severe fault situations considered, it requires developing a more powerful error-detection scheme to raise the error-detection coverage to a sound level.

Basically, the data paths consist of register file and various functional units. We assume that the register file is protected by an error-correcting code.

Therefore, in the following, we focus on the issue of how to detect the faults occurring in the functional units with no detection latency. A high-performance processor core may possess several different types of functional units in the data paths, such as integer ALU and load/store units. A couple of identical units are provided for a specific functional type.

2.1 Hybrid approach

The fundamental concept of our approach is to recover the execution errors promptly for each instruction run. To achieve the real-time error recovery by exploiting the simple instruction retry method, the execution results of each instruction must be checked immediately to detect the errors. We propose a hybrid detection approach combining the duplication with comparison, TMR and self-checking methodologies [15] to fulfill the requirement of no detection latency. Our hybrid approach is quite comprehensive in that it offers the design options based on the trade-offs between the hardware redundancy and time redundancy. Such trade-offs can be achieved through the choices of the following design parameters: how many spare units, comparators (CMP), majority voters (MV) and self-checking functional units employed in the error-detection scheme.

In the following illustration, for simplicity of presentation, we assume only one type of functional unit, namely ALU, in the data paths, and use three identical ALUs to demonstrate our hybrid detection methodology. Each ALU includes a multiplier. Since three ALUs are offered, the processor can issue three ALUs' instructions at most per cycle. The proposed approach, however, can be extended easily to a generic processor core where the data paths have more than one functional type with variable number of identical units.

We construct three error-detection schemes based on the hybrid approach with different numbers of self-checking functional units to present our idea. The self-checking design adopts the mod-3 residue code, also known as low-cost residue code for ALUs. The common features of three schemes as shown in Figures 1-3 are one spare ALU, one triple modular redundancy majority voter (TMR_MV) (including error detection capability) as well as two comparators employed in the error-detection design.

There is a concern about why we exploit the TMR for the error detection? It is because TMR has a benefit to avoid activating the procedure of error recovery while only one faulty unit happens. In contrast to TMR, comparison and self-checking schemes needs to spend time for error recovery. In an extreme case of a permanent fault occurring in one unit, the system utilizing the comparison and self-checking methods needs to perform the error

recovery process to overcome the errors every time when the faulty unit is used and the permanent fault is activated to produce the output errors. That will significantly degrade the performance. Instead, the TMR can tolerate one faulty unit, and therefore, no error recovery is required. Hence, using TMR can lower the performance degradation caused by the error recovery. The concern here is again the consideration of real-time computing applications. However, TMR needs more resources to carry out the error detection compared to the comparison and self-checking methods. According to the above discussion, if resources allow, TMR is the first choice for the instruction checking method.

We further discuss why we combine the self-checking with TMR and comparison schemes? As we know, the advantages of TMR and comparison schemes are as follows: simple concepts, easy design and implementation, and suitable for any hardware entity. More importantly, the SEU phenomenon mentioned above has no impact on the EDC for TMR and comparison schemes. However, they suffer a higher hardware redundancy. Contrary to the above schemes, the self-checking circuits generally enjoy less hardware redundancy, but suffer the SEU interference as mentioned before. Moreover, they have more complicated design concepts, and higher implementation complexity. Therefore, the principal idea of our hybrid approach is to utilize the hardware benefit from the self-checking scheme and the coverage benefit from the TMR and comparison schemes to form a feasible error-detection and recovery framework.

With reference to Figures 1-3, the 1-bit and 2-bit 'Error' Signals are used to indicate that the corresponding outputs are correct or not. While some of outputs are wrong, the instruction retry process is activated immediately to recover the current errors. The 'Faulty Unit(s)' points out which unit(s) is(are) faulty. The three schemes are described as follows.

2.2 Scheme 1

This scheme illustrated in Fig. 1 does not furnish the self-checking design for ALUs. For one/two ALUs' instructions executed at the same cycle, we can use TMR/comparison mechanisms to achieve the concurrent error detection. Note that TMR_MV provides the capabilities of detection of multiple faulty units and location of single faulty unit. In contrast to one/two concurrent instructions, three concurrent ALUs' instructions in an execution packet cannot be issued and verified in parallel due to the limited number of ALUs. It needs to be scheduled to two sequential execution packets where one packet contains two instructions and the other holds the rest one. As a result, one extra ALU's cycle is required to complete the execution of three

concurrent ALUs' instructions in order to accomplish the concurrent check for each instruction execution. This implies that the performance of the program execution will be degraded due to the demand of no error-detection latency.

2.3 Scheme 2

Scheme 2 shown in Fig. 2 provides the self-checking multiplier, self-checking adder as well as self-checking logic unit for ALU_C and ALU_D. The main difference between Scheme 2 and 1 is that the Scheme 2 can issue three concurrent ALUs' instructions in parallel where one instruction is verified by comparison and the other two are checked by self-checking ALUs, i.e. one by ALU_C and another by ALU_D. Therefore, Scheme 2 eliminates the performance degradation as shown in Scheme 1 by paying higher hardware overhead. Another difference is in the TMR_MV and CMP. The TMR_MV and CMP in Scheme 1 are modified to include the self-checking error signals produced from the self-checking functional units. Such enhancements let the modified TMR_MV/CMP have the abilities to overcome the common-mode failures and tolerate up to two/one faulty units. We use the situation of two faulty units to show the spirit of modified TMR_MV. Assume ALU_A and ALU_C are faulty. Therefore, in this situation, the TMR_MV in Scheme 1 can only detect the errors but cannot produce the correct output and locate the faulty units. The modified TMR_MV can solve the above problem by using the error signals furnished from the self-checking units. Based on the error signals delivered from ALU_C and ALU_D, the modified TMR_MV can generate the correct output obtained from ALU_D output and identify the ALU_A and ALU_C as faulty. Further assume ALU_A and ALU_C produce the same, erroneous results. Under the circumstances, the errors will cause the common-mode failure in Scheme 1.

However, the Scheme 2 can conquer this kind of errors by using again the error signals from self-checking units. The concept of modified CMP is similar to the modified TMR_MV. It is apparent that the EDC of self-checking schemes plays a significant role in this scheme because in some situations, like the ones described above and three concurrent ALU instructions executed simultaneously, we rely on the self-checking results to determine the outcomes of the instruction checks.

2.4 Scheme 3

This scheme displayed in Fig. 3 is a compromise between Scheme 1 and 2. We only offer the self-checking multiplier in ALU_C; likewise, self-checking adder as well as self-checking logic unit in ALU_D. Clearly, Scheme 3 has lower hardware overhead than Scheme 2, but cannot completely eliminate the performance degradation resulting from the no detection latency demand. The instruction types of an ALU can be categorized into 'add (+)', 'multiply (\times)', and 'logic (L)' three classes. Since the self-checking design is furnished partially compared with the Scheme 2, some of the combinations of the instruction classes are still required to schedule to two execution packets for three concurrent ALUs' instructions. For example, three instructions are all from the same instruction class, such as 'add' class. In this case, one 'add' instruction can be verified by duplication with comparison (using ALU_A and ALU_B) and the second one by the self-checking adder in ALU_D. However, there is no resource left to check the third instruction because of no self-checking adder provided in ALU_C. Consequently, the third 'add' instruction is postponed to the next cycle. Contrary to the above example, if three instructions are all from different classes, they can be executed at the same cycle as illustrated in Fig. 3.

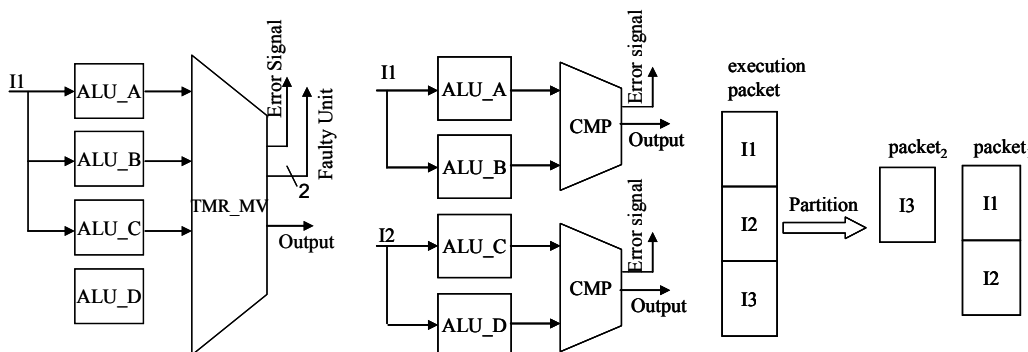


Fig. 1. Scheme 1: left/middle/right for one/two/three instructions.

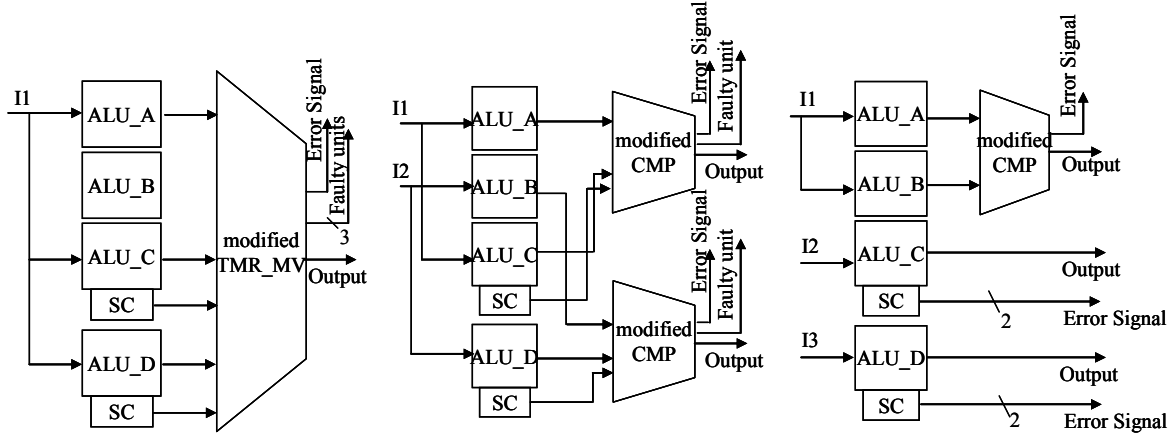


Fig. 2. Scheme 2, where ‘SC’ represents Self-Checking.

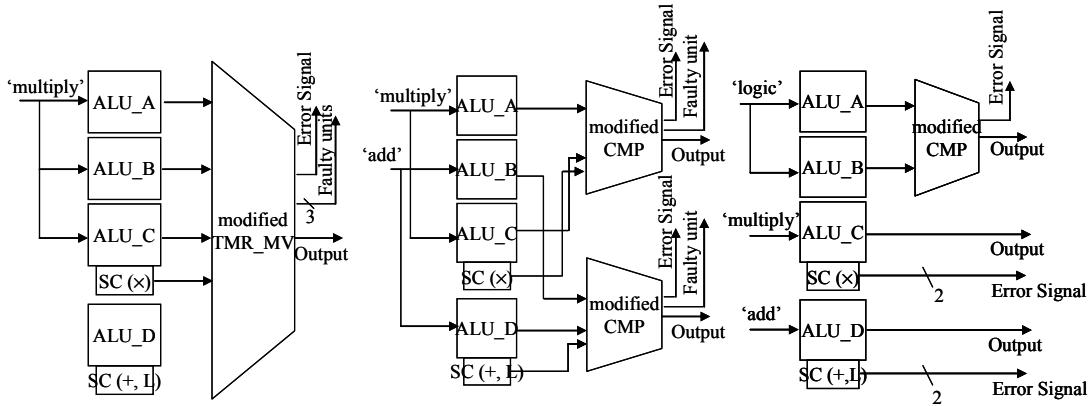


Fig. 3. Scheme 3, where ‘SC (x)’ means the corresponding ALU only supports the SC multiplier; similarly for SC (+, L).

3 Experimental Results and Discussion

To evaluate our hybrid approach, three detection schemes are implemented in an experimental 32-bit VLIW core respectively. 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; • 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 multiplexers 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. Fig. 4 shows the architectural implementation of Scheme 1. The architectures of Schemes 2 and 3 are similar to Fig. 4. The purpose of ALU_Control unit is to carry out the control tasks for error-detection and error-recovery schemes. 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 Fig. 4. The hardware implementations in VHDL and simulated fault injection experiments are performed to measure the design metrics.

The original VLIW core is termed as Scheme 0. Table 1 provides the data of hardware overhead, performance degradation and error-detection coverage (EDC) for Scheme 0, 1, 2 and 3. The implementation technology used here is UMC 0.18μm process. Eight benchmark programs including heap-sort, quick-sort, four queens, 5 × 5 matrix multiplication, FFT and IDCT (8x8), are developed to measure the performance degradation resulting from the error detection for Scheme 1 and 3. According to Table 1, the hardware overhead is Scheme 2 > Scheme 3 > Scheme 1, whereas the performance degradation is Scheme 1 > Scheme 3 > Scheme 2. It is clear that the three schemes exhibit the design trade-off between hardware redundancy and time redundancy.

The benchmark programs are used in the fault injection campaigns to analyze the EDC for Scheme

1, 2 and 3. Our injection 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. To guarantee the statistical validity, a huge amount of fault injection campaigns are conducted to derive the results of EDC. The

injection targets are confined to the four ALUs and if multiple faults exist, the faults can reside either all in the same unit or in different units. We generate three fault environments, which exhibit the low/medium/high probability of multiple faults occurring concurrently during the fault simulation. The lower/upper bounds of EDC in Table 1 are derived from the injection campaigns that have high/low probability of multiple faults occurring concurrently. In other words, the injection campaigns having higher/lower probability of multiple faults occurring concurrently represent the various fault environments with varying degree of severity of faults.

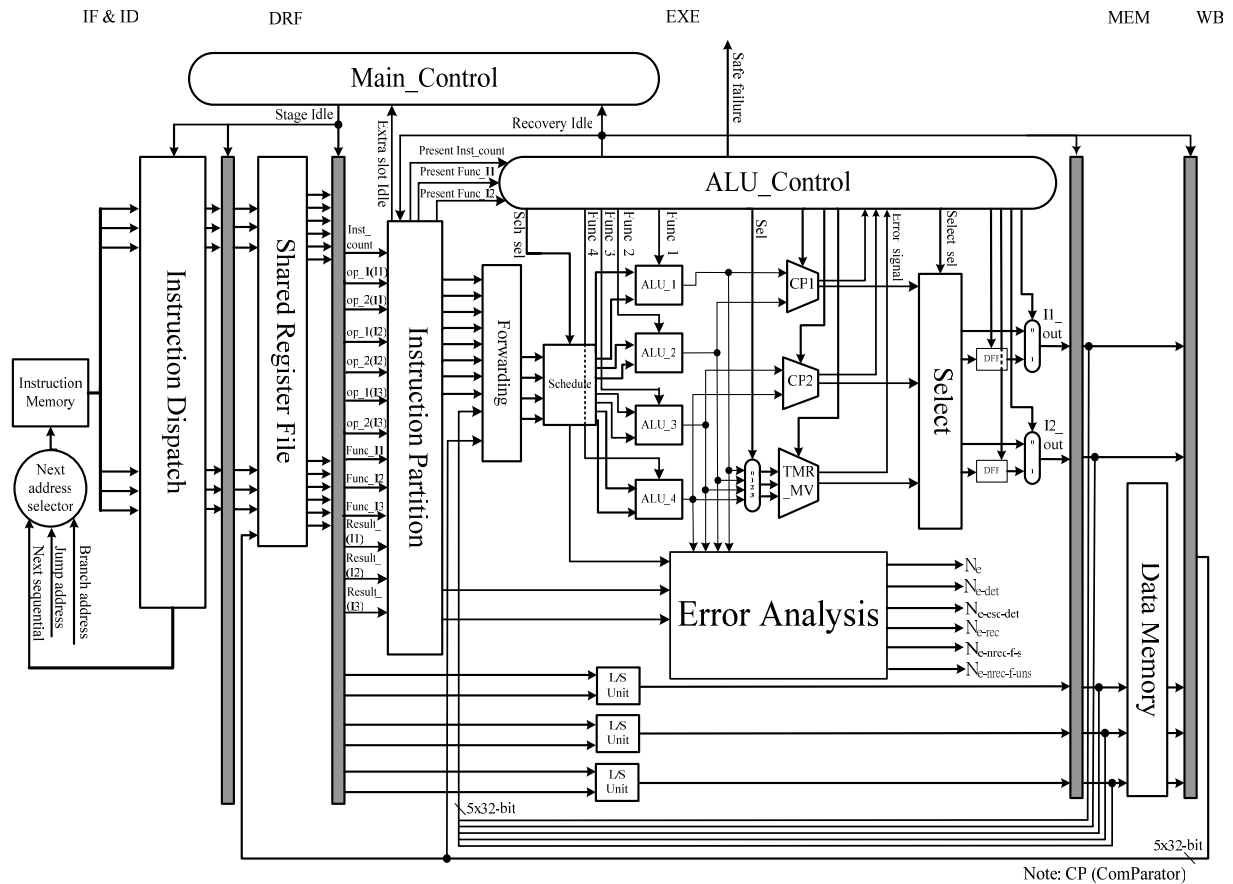


Fig. 4. The fault-tolerant architecture for Scheme 1.

From Table 1, EDC is Scheme 1 > Scheme 2 > Scheme 3. We should note that the EDC are almost the same in upper bound, i.e. in the environment with low probability of multiple faults occurring simultaneously. It is evident that Scheme 2 has higher EDC than Scheme 3. The reason for Scheme 1 > Scheme 2 in severe fault environment is largely due to the effect of multiple faults on self-checking units. As seen in Table 2, the EDC of self-checking schemes decreases rapidly when the multiple faults exist in the same unit. This phenomenon has also been discovered in paper [14]. Since the Scheme 2 and 3 rely on the self-checking techniques to detect the errors for three concurrent instructions, there are

higher probabilities not to detect the errors if the occurring probability of multiple faults increases. As a result, the EDC of Scheme 2 has more than one percent lower than Scheme 1 in more severe fault environment. Expanding the fault injection targets to the entire 'EXE' stage shown in Fig. 4 is in progress and preliminary EDC result for Scheme 1 is 97.5-99.3%. In general, our hybrid detection approach can protect the whole 'EXE' stage and be robust even in the severe fault environment. Moreover, it offers a good design trade-off among the interesting metrics.

The concept of hybrid approach can be further extended by including the self-checking design using various error detecting codes, such as residue, Berger and parity codes. The choice of self-checking techniques has an impact on the hardware overhead, performance degradation and EDC. Table 2 gives a comparison for several self-checking techniques used in the array multiplier. The faults are injected into the inside of the multiplier. The data shown in the columns of Single/Double/Triple in Table 2 are the EDC for single/double/triple

faults injected into the multiplier. The EDC reduces rapidly when the number of faults existing in the same unit increases. It is clear that the various self-checking techniques have significant difference in hardware overhead, performance and EDC in this specific study. Consequently, we need to carefully choose the self-checking mechanisms to gain a good trade-off among the design metrics mentioned in Table 2. For example, if single fault assumption is made, the Mod-3 method is the best choice.

Table 1: The comparison results.

Scheme	Area (μm^2)	Hardware overhead	Performance degradation	EDC (%)
0	9319666	0%	0%	0
1	10708296	14.9%	0.6% - 34.3%	99.77-99.97
2	12152844	30.4%	0%	98.5-99.76
3	11630943	24.8%	0.01% - 15%	98.21-99.71

Table 2: Comparison of various self-checking techniques.

Technique	Area(μm^2)	Overhead	Performance	Single	Double	Triple
Plain Multiplier	759876	0%	9.8 ns	0%	0%	0%
Mod-3	863678	13%	11.56 ns	100%	82%	75%
Mod-7	882406	16%	12.32 ns	100%	97%	93%
Berger [16]	1487615	95%	12.97 ns	100%	95%	90%
Bose-Lin [17]	1297667	70%	11.60 ns	100%	91%	82%

4 Conclusions and Future Work

This paper presents a new hybrid error-detection approach with no detection latency for high-performance microprocessors. No detection latency is essential to real-time error recovery, which is important to the highly dependable real-time computing applications. Our hybrid approach can provide a good design compromise among hardware overhead, performance degradation and error-detection capability. A thorough fault injection campaign has been conducted to assess the error-detection coverage under a variety of fault environments so as to realize the capability of our scheme in different fault scenarios. Thus, 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.

The remaining work is to further investigate the effect of self-checking design using various error detecting codes on the hardware cost, performance and error-detection coverage. To do that, two more experiments will be explored: one is to replace the Mod-3 method with the parity prediction scheme ; the other is to mix the Mod-3 and parity prediction schemes together. As we know, the parity prediction scheme has a lower hardware cost and error-detection coverage compared to the techniques shown in Table 2. The first experiment will be used to compare the parity prediction scheme with the Mod-3 method. The second experiment is to examine how effective of the combination of the different self-checking schemes in one design.

Acknowledgements: 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.

5 References

- [1] Shivakumar, P. et al.: 'Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic', *IEEE Intl. Conf. on Dependable Systems and Networks (DSN'02)*, 2002, pp. 389-398.
- [2] Karnik, T., Hazucha, P., and Patel, J.: 'Characterization of Soft Errors Caused by Single Event Upsets in CMOS Processes', *IEEE Trans. on Dependable and Secure Computing*, 2004, 1, (2), pp. 128-143.
- [3] Saggese, G. P. et al.: 'Microprocessor Sensitivity to Failures: Control vs. Execution and Combinational vs. Sequential Logic', *IEEE Intl. Conf. on Dependable Systems and Networks (DSN'05)*, 2005, pp. 760-769.
- [4] Chen, Y. Y.: 'Concurrent Detection of Control Flow Errors by Hybrid Signature Monitoring', *IEEE Trans. on Computers*, Vol. 54, No. 10, pp. 1298-1313, October 2005.
- [5] J. G. Holm and P. Banerjee, "Low Cost Concurrent Error Detection in A VLIW Architecture Using Replicated Instructions," *Intl. Conf. on Parallel Processing*, pp. 192-195, 1992.
- [6] M. Franklin, "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.
- [7] Nickle, J. B., and Somani, A. K.: 'REESE: A Method of Soft Error Detection in Microprocessors', *IEEE Intl. Conf. on Dependable Systems and Networks (DSN'01)*, 2001, pp. 401-410.
- [8] S. Kim and A. K. Somani, "SSD: An Affordable Fault Tolerant Architecture for Superscalar Processors," *Pacific Rim Intl. Symposium. On Dependable Computing*, pp. 27-34, 2001.
- [9] N. Oh, P. P. Shirvani and E. J. McCluskey, "Error Detection by Duplicated Instructions in Super-Scalar Processors," *IEEE Trans. on Reliability*, Vol. 51, No. 1, pp. 63-75, March 2002.
- [10] Bolchini, C.: 'A Software Methodology for Detecting Hardware Faults in VLIW Data Paths', *IEEE Trans. on Reliability*, 52, (4), 2003, pp. 458-468.
- [11] 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.
- [12] S. W. Kwak and B. K. Kim, "Task-Scheduling Strategies for Reliable TMR Controllers Using Task Grouping and Assignment", *IEEE Trans. on Reliability*, Vol. 49, No. 4, pp. 355-362, December 2000.
- [13] A. Avizienis, J.-C. Laprie, B. Randell and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing", *IEEE Trans. on Dependable and Secure Computing*, Vol. 1, No. 1, 2004.
- [14] Rossi, D. et al.: 'Multiple Transient Faults in Logic: An Issue for Next Generation ICs?', *IEEE Intl. Symposium on Defect and Fault Tolerance in VLSI Systems*, 2005, pp. 352-360.
- [15] Lala, P. K.: 'Self-Checking and Fault-Tolerant Digital Design' (*MORGAN KAUFMANN*, 2001).
- [16] Lo, J. C. et al.: 'An SFS Berger Check Prediction ALU and Its Application to Self-Checking Processor Designs', *IEEE Trans. on CAD*, 11, (4), 1992, pp. 525-540.
- [17] Bose, B., and Lin, D. J.: 'Systematic Unidirectional Error Detecting Codes', *IEEE Trans. on Computers*, 34, (11), 1985, pp. 1026-1032.