Datapath Error Detection with No Detection Latency for High-Performance Microprocessors

YUNG-YUAN CHEN[1], KUEN-LONG LEU[2], KUN-CHUN CHANG[1]

[1]Department of Computer Science and Information Engineering
Chung-Hua University
No. 707, Sec. 2, Wu-Fu Rd., Hsin-Chu
TAIWAN
chenyy@chu.edu.tw, m09402023@cc.chu.edu.tw

[2]Department of Electrical Engineering
National Central University
No. 300, Jhongda Rd., JhongLi City, Taoyuan County
TAIWAN
945401025@cc.ncu.edu.tw

Abstract: - Error detection plays an important role in fault-tolerant computer systems. Two primary parameters concerned for error detection are the coverage and latency. In this paper, a new, hybrid error-detection approach offering a very high coverage with zero detection latency is proposed to protect the data paths of high-performance microprocessors. The feature of zero detection latency is essential to real-time error recovery. The hybrid error-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, performance and error-detection coverage to be incorporated. An experimental 32-bit VLIW core was employed to demonstrate the concept of hybrid detection approach. The hardware implementations in VHDL and simulated fault injection experiments were conducted to measure the interesting design metrics, such as hardware overhead, performance degradation and error-detection coverage.

Keywords: Concurrent error detection, error-detection coverage, error-detection latency, fault injection, hybrid detection approach.

1 Introduction

The rate of radiation-induced soft errors increases rapidly, especially in combinational logic, while the chip fabrication enters the very deep submicron technology [1-3]. Such an influence raises the urgent need to incorporate the fault tolerance into the high-performance microprocessors, system-on-chip (*SoC*) and embedded systems for safety-critical applications [4-6]. Concurrent error detection provides an effective approach to detect the errors caused by transient and intermittent faults [7-10]. 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 [7-9], [11-18] that uses the instruction replication and recomputation 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 recomputation. Drawbacks of the previous studies are: • variable detection latency which complicates the analysis of the impact of error recovery on performance; • lengthy detection latency that increases the error-recovery time as well as program execution time. For example, the error-detection scheme presented in [18] holds 692 cycles of detection latency on average, and 36183 cycles for the worst case. Such a lengthy latency requires more time for the error recovery and this will degrade the performance significantly once the errors occur. Such a lengthy recovery may be detrimental to the real-time computing applications. Besides that, error-detection schemes with variable detection latency would pay a higher hardware cost to implement the error-recovery process.

In this study, to minimize the effect of error-detection latency on the recovery performance and hardware complexity, the zero detection latency

is set as our design goal so as to accomplish the real-time error recovery by simply using the cost-effective instruction-retry method. To fulfill the requirement of zero 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. In this work, a new, hybrid error-detection approach is proposed to detect the faults occurring in the data paths during the instruction executions with zero detection latency.

The paper is organized as follows. In Section 2, a general framework of hybrid error-detection approach is proposed and demonstrated by a case study to explore the design options with various hardware redundancy, time redundancy and error-detection coverage (EDC). Section 3 presents the hardware implementations of the demonstrated detection schemes illustrated in Section 2 in an experimental 32-bit VLIW core and the measurements of hardware overhead and performance degradation. Experimental results and a thorough analysis of EDC are given in Section 4. The conclusions appear in Section 5.

2 Hybrid Error-Detection Approach

In addition to the single fault model commonly adopted previously, two classes of faults described below (named 'fault class 1' and 'fault class 2') are particularly addressed in our error-detection approach: 1. Correlated transient faults [19, 20] (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 [21]. Such a SEU effect could seriously degrade the EDC for self-checking circuits generally designed with a single fault assumption [21]. 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 approach to raise the EDC 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 zero 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. One or more than one identical units are provided for a specific functional type.

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, henceforth referred to as comparison, triple modular redundancy (TMR) and self-checking methodologies [22] to fulfill the requirement of zero detection latency. Our hybrid approach is quite comprehensive in that it offers the design options based on the trade-offs among the hardware redundancy, time redundancy and EDC. Such trade-offs can be achieved through the choices of the following design parameters: how many spare units, comparators (CPR), majority voters (MV) and self-checking functional units employed in the error-detection scheme.

To cope 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. It is worth noting that why we exploit the TMR for the error detection? This 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 need 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; this 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 performance concern here is the consideration of the real-time computing applications, which have a strict time constraint. However, TMR needs more resources to carry out the error detection compared to the comparison and self-checking methods.

We further discuss why we combine the self-checking with TMR as well as 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 interference of SEUs as mentioned in the fault class 2 has no impact on the EDC for TMR and comparison schemes. However, they suffer from a higher hardware redundancy. Contrary to the TMR and comparison schemes, the self-checking circuits generally enjoy less hardware redundancy, but suffer from multiple transient faults induced by the SEUs as stated in the fault class 2. Moreover, they have more complicated design concepts, and higher implementation complexity. Therefore, the principal idea of our hybrid approach is to utilize the hardware advantage of the self-checking scheme and the coverage advantage of the TMR and comparison schemes to form a feasible error-detection framework.

2.1 Detection Framework with Zero Latency

The following notations are developed:

n: Number of identical modules for a specific functional type x, $n > 1$; *n* is also the maximum number of instructions that can be executed concurrently in the modules of type x;

$n_{s\text{-}c}$: Number of modules equipped with self-checking ability in the *n* modules of type x, $0 \le n_{s\text{-}c} \le n$.

s: Number of spare modules added to the type x, $s \ge 0$.

$s_{s\text{-}c}$: Number of spare modules equipped with self-checking capability in the *s* spares of type x, $0 \le s_{s\text{-}c} \le s$.

m: Number of instructions in an execution packet for type x, $m \le n$. An execution packet is defined as the instructions in the same packet can be executed in parallel.

ETMR/ECMP: Enhanced TMR/Enhanced comparison schemes are the combination of TMR/comparison schemes with the self-checking methodology. It means that at least one module used in TMR/comparison possesses the self-checking function. The enhanced TMR majority voter (ETMR_MV)/enhanced comparator (E_CPR) receives the module outputs as well as the self-checking error signals produced from the modules equipped with self-checking function. The goal of ETMR and ECMP is to conquer the common-mode failures [23, 24] which could occur in TMR and comparison schemes when two or three modules produce the identical, erroneous results to TMR_MV or two modules produce the same, erroneous results to comparator.

We use the following example to show the power of ETMR. Given identical modules A, B, and C that are employed in TMR operation, where module B is furnished with the self-checking function. We assume that modules A and B are faulty and produce the same, erroneous results. Under the circumstances, the errors will cause a common-mode failure in traditional TMR scheme. However, the ETMR scheme can be aware of this kind of error by using the error signal delivered from the self-checking unit. If the error signal provided from module B shows that the output of module B is wrong, then the common-mode failure caused by the modules A and B is discovered; otherwise, the common-mode failure escapes being detected. It is evident that the EDC of self-checking technique decides the detection capability of the common-mode failures for ETMR.

Following the above illustration, the principal concept behind the ETMR scheme is described below. The ETMR possesses a two-layer fault protection in that it exploits the results of TMR and self-checking techniques to determine the output of ETMR or trigger the error signal to activate the error-recovery process. First of all, we employ the outcome of TMR voter to compare with each module output. The results of such comparisons can be exploited to identify the outputs of three modules that could fall into one of the following situations: 1. three modules have identical outputs; 2. two of three modules hold the same outputs; 3. three modules have different outputs. Then, each situation described above is associated with the self-checking results to decide the outcome of ETMR. The handling processes for each situation are depicted as follows:

Situation 1: In this situation, if the self-checking results show no errors, then the results of self-checking are consistent with the result of TMR, and clearly, ETMR employs the outcome of TMR as its output; else, the common-mode failure is identified, and therefore, the error signal is triggered to activate the error-recovery process.

Situation 2: There are two cases to consider in this situation.

Case 1: At least one module out of the modules having identical outputs is equipped with the self-checking function. In this case, if the results of self-checking are consistent with the result of TMR, then ETMR employs the outcome of TMR as its output; else, the common-mode failure is recognized and ETMR triggers the error signal to activate the error-recovery process.

Case 2: The modules with identical outputs do not hold the self-checking capability. As a result, the module judged as faulty by the TMR owns the

self-checking function. If the self-checking result is normal, then a conflict between TMR and self-checking results occurs; under the circumstances, ETMR triggers the error signal to activate the error-recovery process. Otherwise, ETMR employs the outcome of TMR as its output.

Situation 3: TMR fails if the outputs of three modules fall into this situation. Hence, ETMR triggers the error signal to activate the error-recovery process.

The principal concept behind the ECMP scheme is similar to ETMR and omitted here.

Error-detection framework: The core of the framework is based on the hybrid detection approach which consists of the following basic detection techniques: ETMR, ECMP, TMR, comparison and self-checking techniques as described before. According to the previous discussion, we rank the priority of the usage of the above error-detection techniques from high to low as ETMR, ECMP, TMR, comparison and self-checking, where the technique's rank has reference to the capability of the fault tolerance for the considered schemes. Given n, $n_{s\text{-}c}$, s and $s_{s\text{-}c}$, the design issue is how to check each instruction under the resource constraint to achieve the zero error-detection latency, to minimize the performance degradation and importantly to gain a better EDC. The objective of zero error-detection latency can be accomplished by verifying the execution results promptly for each instruction. Therefore, each instruction execution will require some extra resources such that the results of each instruction can be validated immediately. As a result, more resources are needed to execute the m instructions in a packet simultaneously. The additional resource requirement resulting from the error-detection demand may violate the resource constraint. If such a resource violation occurs, then the processor won't have the adequate resources to execute and check the m instructions in a packet concurrently. Consequently, some instructions in a packet cannot be protected and this flaw will degrade the EDC. For coverage concern, the complete check of each instruction becomes a must. To solve the coverage problem, a method of packet partition is developed and it is to partition such a packet into several packets which will be executed sequentially. However, such partitions will induce some extra cycles, and therefore, degrade the performance of program execution. The error-detection algorithm is presented as follows:

Algorithm 1: Given n, $n_{s\text{-}c}$, s and $s_{s\text{-}c}$, the expression $2(m - n_{s\text{-}c} - s_{s\text{-}c}) \le (n - n_{s\text{-}c}) + (s - s_{s\text{-}c})$, called expression (1), is used to examine whether the m instructions in present packet can be executed and checked simultaneously or not. If expression (1) is true, then the m instructions in the packet can be executed and checked in parallel; else, the packet partition is required to guarantee that each instruction execution will be verified. We now explain the meaning of expression (1). There are two cases to consider in expression (1). The first case is $n_{s\text{-}c} + s_{s\text{-}c} = 0$. Expression (1) becomes $2m \le n + s$. It is obvious that if the number of available modules $n + s$ is greater than or equal to $2m$, then the m instructions in present packet can be executed and checked in parallel. The second case is $n_{s\text{-}c} + s_{s\text{-}c} \neq 0$. In this case, if expression (1) is true, then one possibility is to let $(n_{s\text{-}c} + s_{s\text{-}c})$ of m instructions be checked each by self-checking scheme. After that, the remaining instructions can be examined by comparison and/or TMR. The details are given below:

if $(2(m - n_{s\text{-}c} - s_{s\text{-}c}) \le (n - n_{s\text{-}c}) + (s - s_{s\text{-}c}))$ ***then***

{m instructions in present packet can be executed and checked simultaneously. We first divide the m instructions into five groups: G(1) to G(5); there are m_1, m_2, m_3, m_4 and m_5 instructions in G(1), G(2), G(3), G(4) and G(5) respectively, and $m_1 + m_2 + m_3 + m_4 + m_5 = m$, where m_1, m_2, m_3, m_4, $m_5 \ge 0$. The instructions in G(x), x = 1 to 5, can be examined by ETMR, ECMP, TMR, comparison and self-checking schemes, respectively. The following two equations are derived from the module resource constraint: $3m_1 + 2m_2 + 3m_3 + 2m_4 + m_5 \le n + s$ and $m_{1\text{-sc}} + m_{2\text{-sc}} + m_5 \le n_{s\text{-}c} + s_{s\text{-}c}$, where $m_{1\text{-sc}}$ and $m_{2\text{-sc}}$ are the number of modules equipped with the self-checking functions, which are used in the ETMR and ECMP schemes, respectively. It is clear that the above three equations could have one or several solutions, where a solution is represented as $(m_1, m_2, m_3, m_4, m_5, m_{1\text{-sc}}, m_{2\text{-sc}})$. So the next question is if several solutions exist, how to choose an effective solution that is superior to most of the feasible solutions derived from the above equations. The choice is in accordance with which solution that can provide a better EDC. The rank of the basic detection schemes mentioned earlier is exploited to select a sound solution. The guideline of the selection of an effective solution is presented next.

if $(n_{s\text{-}c} + s_{s\text{-}c} \neq 0)$ ***then***

{A flow chart as shown in Fig. 1 is used to characterize the selection criterion. It is evident that a solution selected based on this criterion is effective from the EDC point of view. }

else {We choose a solution whose m_3 value is maximal.}}

else

{Due to lack of the enough module resources, we need to partition the current packet into two or three packets which will be executed sequentially. Such partitions will induce some extra cycles, and therefore, degrade the performance of program execution. If n is odd with no redundancy added, i.e. $s = n_{s\text{-}c} = s_{s\text{-}c} = 0$, then the worst partition occurs in a packet containing n instructions, which requires partitioning into three packets. It is easy to see that a packet normally requires partitioning into two packets except the worst partition depicted above. As a result, the partition of a packet will normally induce one extra execution cycle except the worst partition, where two extra cycles are needed. Next, the principle of instruction partitioning is described as follows. For two packet's partition, if m is even, then we distribute $\frac{m}{2}$ instructions to each packet; else, $\left\lfloor \frac{m}{2} \right\rfloor + 1$ and $\left\lfloor \frac{m}{2} \right\rfloor$ instructions to the first and second packets respectively. For three packet's partition, we first distribute $\left\lfloor \frac{m}{3} \right\rfloor$ instructions to each packet; next, if $(m - \left\lfloor \frac{m}{3} \right\rfloor \times 3) = 1$, then the remaining instruction is assigned to the first packet; if $(m - \left\lfloor \frac{m}{3} \right\rfloor \times 3) = 2$, then the remaining two instructions are evenly distributed to the first and second packets.}

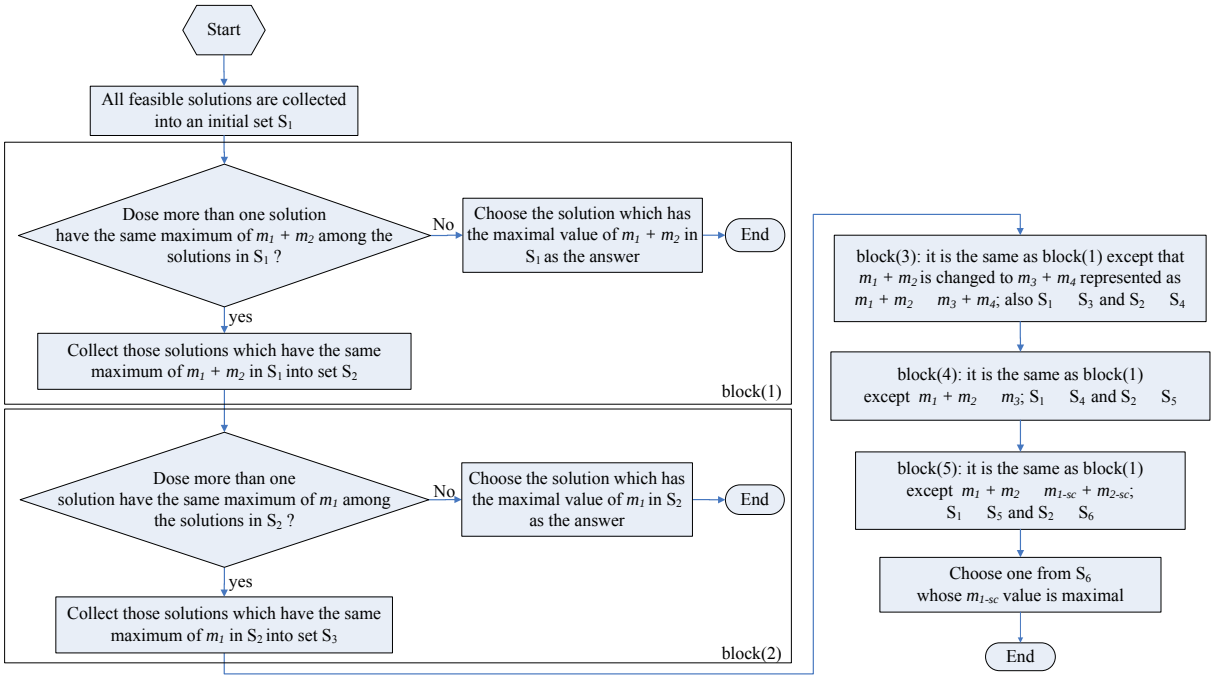

Fig. 1: Selection criterion.

We use the following example to demonstrate our hybrid error-detection approach:

Example 1: Given $n = 4$, $n_{s\text{-}c} = 0$, $s = 1$ and $s_{s\text{-}c} = 1$, according to error-detection framework described above, while $m = 1$, the set of feasible solutions is {(1, 0, 0, 0, 0, 1, 0), (0, 1, 0, 0, 0, 0, 1), (0, 0, 1, 0, 0, 0, 0), (0, 0, 0, 1, 0, 0, 0), (0, 0, 0, 0, 1, 0, 0)}. Next, from block (2) of Fig. 1, (m_1, m_2, m_3, m_4, m_5, $m_{1\text{-sc}}$, $m_{2\text{-sc}}$) = (1, 0, 0, 0, 0, 1, 0) is selected as the answer. So if an execution packet contains only one instruction then it will be checked by ETMR scheme. While $m = 2$, the set of feasible solutions is {(1, 0, 0, 1, 0, 1, 0), (0, 1, 1, 0, 0, 0, 1), (0, 1, 0, 1, 0, 0, 1), (0, 0, 1, 0, 1, 0, 0), (0, 0, 0, 2, 0, 0, 0), (0, 0, 0, 1, 1, 0, 0)}. Again, from block (2) of Fig. 1, (1, 0, 0, 1, 0, 1, 0) is selected as the answer. So if a packet contains two instructions then one instruction will be checked by ETMR and the other by comparison scheme. While $m = 3$, (0, 0, 0, 2, 1, 0, 0) is the only solution. As a result, a packet holds three instructions, two checked by comparison and the rest one verified by self-checking mechanism. For $m = 4$, due to limited resources, we need to partition this packet into two packets, and each packet contains two instructions.

2.2 Case Study

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 ($n = 3$) to demonstrate our hybrid

detection approach. Each ALU includes a multiplier. Since three ALUs are offered, the processor can issue three ALU instructions at most per cycle. Given $n = 3$, $s = 1$, and with various $n_{s\text{-}c}$ and $s_{s\text{-}c}$, we construct three hybrid error-detection schemes based on the framework presented above to explore the design compromise among the hardware overhead, performance degradation and EDC. The self-checking design adopts the mod-3 residue code, also known as low-cost residue code for ALUs. With reference to Fig. 2, 3 and 4, the 'Error Signal' is used to indicate that the corresponding output is correct or not. If 'Error Signal' is on, the instruction-retry process is activated immediately to recover the current errors. We describe three schemes as follows:

Scheme 1: Self-checking technique is not employed in this scheme, i.e. $n_{s\text{-}c} = s_{s\text{-}c} = 0$. According to error-detection framework, (m_1, m_2, m_3, m_4, m_5, $m_{1\text{-sc}}$, $m_{2\text{-sc}}$) = (0, 0, 1, 0, 0, 0, 0) is selected as the solution for $m = 1$. So if an execution packet contains only one ALU instruction then it will be checked by TMR scheme. For $m = 2$, $2m$ is equal to $n + s$, so (m_1, m_2, m_3, m_4, m_5, $m_{1\text{-sc}}$, $m_{2\text{-sc}}$) = (0, 0, 0, 2, 0, 0, 0). Hence, each instruction will be verified by comparison technique. For $m = 3$, $2m > n + s$, 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 concurrent error-detection need. Clearly, the instruction partition resulting from the lack of ALU resources will cause the performance degradation. It is evident that three spare ALUs are required to fully eliminate the performance degradation. However, it pays very high hardware cost.

Scheme 2: This scheme exploits the self-checking methodology to remove the performance degradation as seen in Scheme 1. Let $n_{s\text{-}c} = 1$, and $s_{s\text{-}c} = 1$. Note that an ALU equipped with the self-checking mechanism as illustrated in Fig. 3 comprises a self-checking adder/multiplier/logic circuit. For $m = 1$, (m_1, m_2, m_3, m_4, m_5, $m_{1\text{-sc}}$, $m_{2\text{-sc}}$) can be (1, 0, 0, 0, 0, 1, 0), (1, 0, 0, 0, 0, 2, 0), (0, 1, 0, 0, 0, 0, 1), (0, 1, 0, 0, 0, 0, 2), (0, 0, 1, 0, 0, 0, 0), (0, 0, 0, 1, 0, 0, 0) or (0, 0, 0, 0, 1, 0, 0). Clearly, (1, 0, 0, 0, 0, 2, 0) is selected as the final solution. So, if a packet contains only one ALU instruction, then it will be checked by ETMR scheme. For $m = 2$, (0, 2, 0, 0, 0, 0, 2) is the solution, and therefore, both instructions can be verified by ECMP scheme. For $m = 3$, it satisfies the condition $2(m - n_{s\text{-}c} - s_{s\text{-}c}) = (n - n_{s\text{-}c}) + (s - s_{s\text{-}c})$, and the solution is (0, 0, 0, 1, 2, 0, 0). Compared to Scheme 1, the self-checking technique allows us to spend less hardware to achieve no performance degradation. This is the role of self-checking technique played in the hybrid detection approach. As can be seen from Fig. 3, two instructions will be verified only by self-checking technique when a three instruction's packet is executed. Under the circumstances, how serious of the SEU interference will have a significant impact on the EDC of self-checking circuits. According to our experimental results, we observed that the worse the SEU interference is, the lower the EDC of self-checking circuits. For one and two instructions' packets, the ETMR and ECMP mechanisms are used to examine the instruction executions, which can further enhance the EDC compared to TMR and comparison techniques employed in Scheme 1.

Scheme 3: This scheme represents a compromise between Schemes 1 and 2. We only offer the multiplier in ALU_3, the adder as well as logic unit in ALU_4 having self-checking function. Clearly, Scheme 3 has lower hardware overhead than Scheme 2, but cannot completely eliminate the performance degradation. The instruction types of an ALU can be categorized into 'add (+)', 'multiply (×)', and 'logic (L)' three classes. Since the self-checking design is furnished partially compared with Scheme 2, a packet containing three instructions may still require partitioning to two execution packets. The need of partition or not depends on the type of instruction combinations in a packet. For example, three instructions in a packet are all from the same instruction class, such as 'add' class. In this case, one 'add' instruction can be verified by comparison (using ALU_1 and ALU_2) and the second one by the self-checking adder in ALU_4. However, there is no resource left to check the third 'add' instruction because of no self-checking adder provided in ALU_3. 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 exhibited in Fig. 4. The EDC of Scheme 3 is slightly lower than Scheme 2 due to the following reasons. One is ETMR used in Scheme 3 has only one module equipped with the self-checking function, whereas Scheme 2 has two modules equipped with the self-checking function. Another is in the execution of two instructions' packets. The packets, like ('+', '+'), ('×', '×'), ('L', 'L') and ('+', 'L'), will be verified one by ECMP and the other by comparison in Scheme 3. Contrast to that, all two instructions' packets in Scheme 2 will be checked by ECMP.

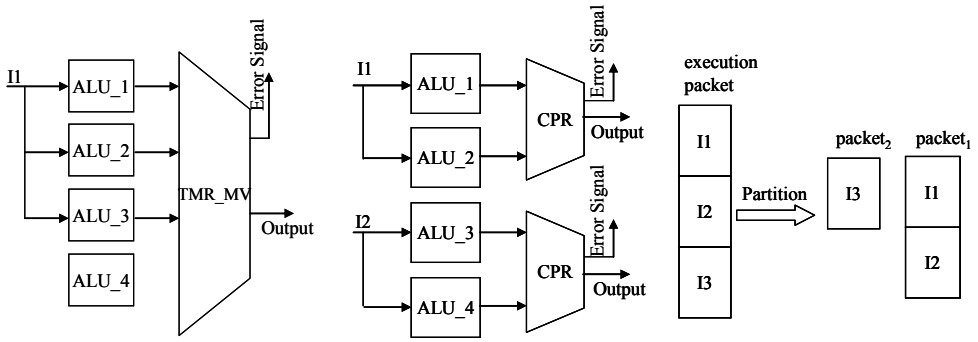

Fig. 2: Scheme 1, left/middle/right for one/two/three instructions in a packet.

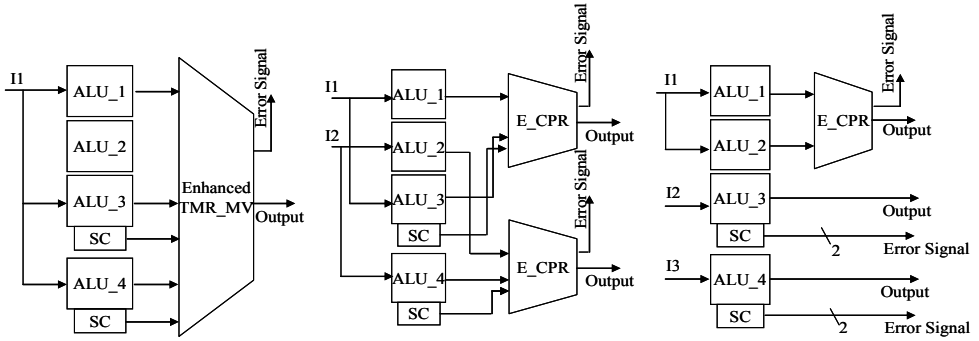

Fig. 3: Scheme 2, where 'SC' represents self-checking.

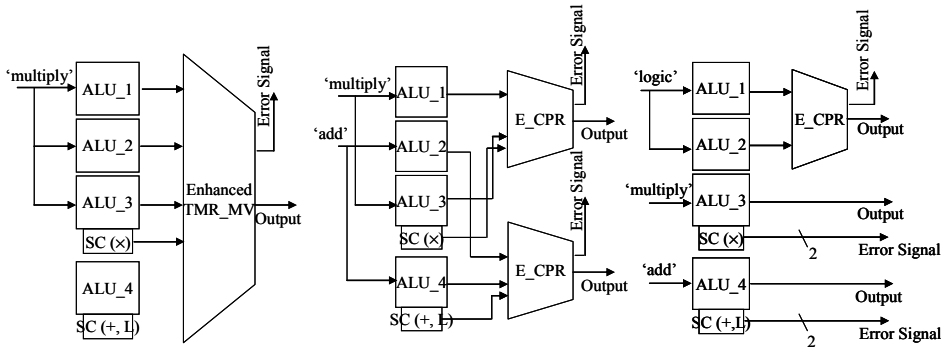

Fig. 4: Scheme 3, where 'SC (×)' means the corresponding ALU only supports the SC multiplier; similarly for SC (+, L).

2.3 Generic Design Consideration

In last section, we use three identical modules to demonstrate our approach. The number of identical modules could be various in data paths of high-performance processors. In this section, we use three to five identical modules to discuss the generic design consideration in terms of hardware overhead, performance degradation and error-detection scheme. Table 1 lists the data of design parameters derived from the error-detection framework presented in Section 2.1 for various numbers of identical modules.

The hardware overhead in Table 1 simply counts the part of modules themselves and does not include other circuit portions resulting from the demand of fault tolerance, such as comparators and majority voters. The notations A and A_s shown in the column of hardware overhead represent the area of an ALU and the area induced by the self-checking design, respectively. As mentioned before, the demand of zero detection latency results in the performance

degradation due to the partition of instructions into several sequential execution packets. We use the notation, for example, 3→ (2, 1) to represent the partition of three instructions in a packet into two sequential packets where one packet contains two instructions and the other holds the rest one. Similarly, 5→ (2, 2, 1) will cause two cycle's performance degradation. The notations shown in the column of error-detection scheme represent the methodologies used to detect the errors. For instance, '2: (1: ETMR, 1: ECMP)' represents an execution packet with two instructions where one instruction is checked by ETMR technology and the other is examined by ECMP scheme; likewise, '2: ETMR' means that it is an execution packet containing two instructions and each instruction is verified by ETMR scheme.

As can be seen from Table 1, for a specific number *n*, our approach offers several design options based on the trade-off among the metrics of hardware overhead, performance degradation and fault tolerance capability. Without loss of generality, we use $n = 5$ as an example to explain the design trade-off. For $n = 5$, there are three design choices shown in Table 1. If the design is performance-oriented, we could choose the design using $s = 2$, i.e., the design using two spares, and three ALUs equipped with the self-checking functions. Clearly, the design with no spare enjoys the lowest hardware overhead but suffers from the highest performance degradation and the lowest EDC among three design options. Generally speaking, the design with more spares has the advantages of lower performance degradation and higher EDC but suffers from higher hardware overhead. From the above discussion, it is obvious that our hybrid approach can be extended easily to a generic processor core where the data paths have more than one functional type.

Table 1: Data of design parameters for various numbers of identical modules, where HO is an abbreviation of hardware overhead, IP is instruction partitioning, COM is comparison, and SC is self-checking.

n	$n_{s\text{-}c}$	s	$s_{s\text{-}c}$	HO	IP	Error-detection Scheme
3	0	1	0	$1/3$	3→ (2, 1)	1: TMR; 2: COM
3	1	1	1	$(A+2A_s)/3A$	None	1: ETMR; 2: ECMP; 3: (1: COM, 2: SC)
3	0	2	2	$2(A+A_s)/3A$	None	1: ETMR;2: (1: ETMR, 1: ECMP); 3: (1: ECMP, 1: COM, 1: SC)
4	0	0	0	0%	3→ (2, 1); 4→ (2, 2)	1: TMR; 2: COM
4	0	1	1	$(A+A_s)/4A$	4→ (2, 2)	1: ETMR; 2: (1: ETMR, 1: COM); 3: (2: COM, 1: SC);
4	2	1	1	$(A+3A_s)/4A$	None	1: ETMR; 2: (1: ETMR, 1: ECMP) 3: (2: ECMP, 1: SC); 4: (1: COM, 3: SC)
5	0	0	0	0%	3→ (2, 1); 4→ (2, 2); 5→ (2, 2, 1);	1: TMR; 2: (1: TMR, 1: COM)

5	1	1	1	$(A+2A_s)/5A$	5→ (3, 2);	1: ETMR; 2: ETMR; 3: (2:ECMP, 1: COM); 4: (2:COM, 2: SC)
5	1	2	2	$(2A+3A_s)/5A$	None	1: ETMR; 2: ETMR;3: (1: ETMR, 2: ECMP);4: (1: ETMR, 1: COM, 2: SC); 5: (2: COM, 3: SC)

3 Hardware Implementation and Performance Evaluation

To evaluate our hybrid approach, Schemes 1, 2 and 3 were 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 core consists of five pipeline stages: 'instruction fetch and instruction dispatch (IF & ID)', 'decode and operand fetch from register file (DRF)', 'execution (EXE)', 'data memory reference (MEM)' and 'write back into register file (WB)' stages. This experimental architecture can issue at most three ALU and three load/store instructions per cycle. Fig. 5 shows the architectural implementation of Scheme 2. The architectures of Schemes 1 and 3 are similar to Fig. 5. The purpose of 'ALU_Control' unit is to carry out the control tasks for error detection and error recovery, where the process of error recovery adopts a simple instruction-retry method. Note that the 'Error Analysis' block in execution stage was created only for the purpose of the measurement of the EDC during the fault injection campaigns.

The hardware implementations for three schemes in VHDL were performed to measure the design metrics. The implementation data by UMC 0.18μm process are shown in Table 2, where HO and PD are the abbreviation of hardware overhead and performance degradation, respectively. The original VLIW core is termed as Scheme 0. The area excludes the instruction memory as well as the 'Error Analysis' block. The term of hardware overhead includes the overheads caused by the circuits developed for error detection and error recovery. It is worth noting that the overhead of 'ALU_Control' unit for three schemes is only 0.25~0.3 percent compared to the area of the non fault-tolerant VLIW core. This implies that the control task of our hybrid schemes is simple and easy to implement. For performance consideration, we require that the clock frequency of the fault-tolerant VLIW processors must retain the same as that of non fault-tolerant one, i.e. 128 MHz. Eight benchmark programs including heapsort, quicksort, four queens, 5 × 5 matrix multiplication, FFT and IDCT (8×8) were developed to measure the performance degradation resulting from the error detection in Schemes 1 and 3. As can be seen from Table 2, Schemes 1, 2 and 3 show clearly the design compromise between the hardware redundancy and time redundancy.

Table 2: The data of hardware overhead and performance degradation.

Scheme	Area (μm^2)	HO	PD
0	9319666	0%	0%
1	10708296	14.9%	0.6% - 34.3%
2	12152844	30.4%	0%
3	11630943	24.8%	0.01% - 15%

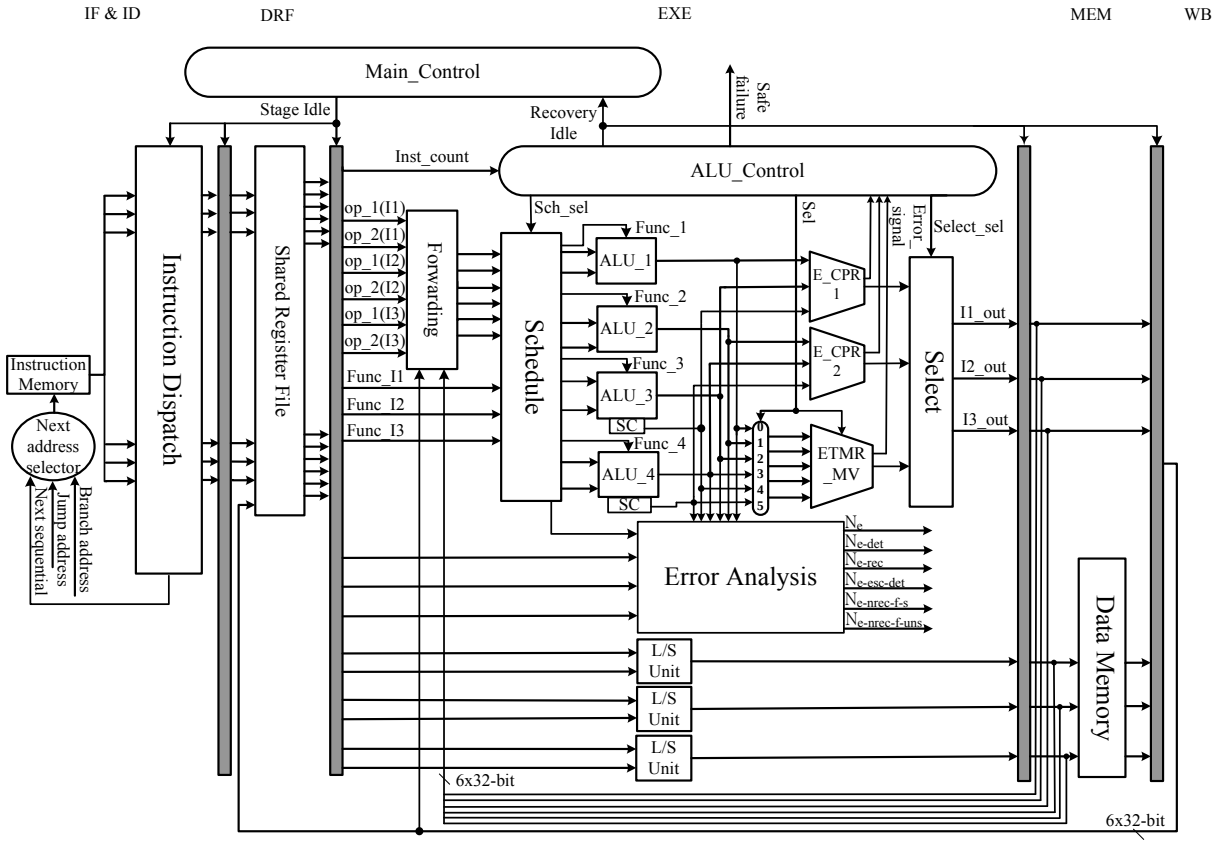

Fig. 5: Fault-tolerant VLIW architecture (Scheme 2).

4 Error-Detection Coverage Analysis

In this section, the analysis of EDC based on the simulation-based fault injection [25-27] is conducted to validate our hybrid schemes. A comprehensive fault tolerance verification platform comprising a simulated fault injection tool [28, 29], 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. 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.

A new characteristic of our injection tool is to offer users the statistical analysis of the injected faults. The statistical data for each injection campaign exhibit the degree of fault's severity, which represents a fault scenario (or called fault environment). The degree of fault's severity is relative to the probability of i faults (denoted as P_i) occurring concurrently while a fault-tolerant system is simulated in the injection campaign, where $i \geq 1$. For example, P_1 = 98% and P_2 = 2% mean that 98 percent is one fault and two percent is two faults when the faults occur throughout the injection campaign. Hence, the injection tool can assist us in creating the proper fault environments that can be used to effectively validate the capability and the strength of a fault-tolerant system under various fault scenarios. As a result, the validation process will be more comprehensive and complete. In a word, the proposed verification platform helps us raise the efficiency and validity of the dependability analysis.

4.1 Interesting Design Metrics

As seen before, five basic error-detection mechanisms, i.e. ETMR, ECMP, TMR, comparison and self-checking, are used in Schemes 1, 2 and 3. It should be pointed out that each basic error-detection mechanism could fail to detect some specific types of errors as described below: • For ETMR, two or three ALUs produce the identical, erroneous results to TMR and the self-checking function also fails to

identify this common-mode failure. • For ECMP, two ALUs produce the same, erroneous results to comparator and the self-checking function also fails to identify this common-mode failure. • For TMR, two or three ALUs produce the identical, erroneous results to TMR. • For comparison, two ALUs produce the same, erroneous results to comparator. • For self-checking, the faults escape being detected due to multiple transient faults. Such detection defects will result in the unsafe failures. In summary, there are two kinds of errors that will lead to the unsafe failure; one is errors resulting in common-mode failures and the other is multiple transient faults going undetected by self-checking circuits.

Let $C_{e\text{-}det}$ denote the EDC, i.e. probability of errors detected; $P_{f\text{-}uns}$ be the probability of system entering the fail-unsafe state. The parameters N_e, $N_{e\text{-}det}$, and $N_{e\text{-}esc\text{-}det}$ represent the total number of errors occurred, the number of errors detected, and the number of errors undetected, respectively. The $C_{e\text{-}det}$ and $P_{f\text{-}uns}$ can be expressed as:

$$C_{e-\det} = \frac{N_{e-\det}}{N_e}; P_{f-uns} = \frac{N_{e-esc-\det}}{N_e} \qquad (2)$$

4.2 Simulation Results and Discussion

We have conducted a huge amount of fault injection campaigns to calculate the expression (2) under various fault scenarios. The benchmark programs mentioned in Section 3 are used in the fault injection campaigns to analyze the design metrics expressed in (2). The common rules of fault injection for the experiments 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 except the 'load/store' and 'Error Analysis' units, as shown in Fig. 5. To inject the faults into the inside of the adders and multipliers, those components were implemented at the gate level. As shown in [1], the fault rate is proportional to the circuit area. So, the fault distribution for each injection campaign is based on the area complexity of the components which are considered to be the injection targets. 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. Table 3 lists the statistical data for five injection campaigns, where P_i is the probability of i faults occurring concurrently while the fault-tolerant system is simulated in the injection campaign. Clearly, from Table 3, five campaigns represent five different fault environments, where the fault environment is getting worse from Campaigns 1 to 5 because the occurring probability of multiple faults P_i ($i \geq 2$) is getting higher from Campaigns 1 to 5. 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 verification platform can furnish more comprehensive and solid measurement of fault-tolerant design metrics.

Table 4 illustrates the experimental results of $C_{e\text{-}det}$ and $P_{f\text{-}uns}$ for Schemes 1, 2 and 3 under various injection campaigns. The results obtained in Table 4 have 95% confidence interval of ±0.14% to ±0.98%. The salient points of the results presented in Table 4 are summarized as follows. One is the EDC decreases and the probability of unsafe failure increases as the fault environment becomes worse. It is evident that the increase of the occurring probability of multiple faults will raise the probability of unsafe failures caused by the error scenarios described in Section 4.1. Another chief point is the rank of scheme's EDC is Scheme 2 > Scheme 3 > Scheme 1. As a result, among Schemes 1, 2 and 3, Scheme 1 enjoys the lowest hardware overhead, but suffers from the highest performance degradation and lowest EDC, whereas Scheme 2 has the advantages of lowest performance degradation and highest EDC, but suffers from the highest hardware overhead. Apparently, our hybrid approach offers the design options based on the trade-offs among the hardware redundancy, time redundancy and EDC. The last point worth to be mentioned is the results presented are quite positive and sound those declare the effectiveness of our fault-tolerant methodology even in a very severe fault environment.

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 5 gives a comparison for several self-checking techniques used in an array multiplier. The faults are injected into the inside of the multiplier. The EDC reduces rapidly when the number of faults existing in the same unit increases. This phenomenon has been discovered in paper [21] as well. 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 interesting design metrics.

Table 3: Statistical data for various fault injection campaigns.

Campaign / P_i (%)	1	2	3	4	5
P_1	96.1 ~ 96.77	76.17 ~ 76.88	55.12 ~ 55.46	37.04 ~ 37.8	22.1 ~ 22.43
P_2	3.02 ~ 3.31	19.39 ~ 20.08	32.19 ~ 33.27	37 ~ 37.57	34.23 ~ 36.4
P_3	0.21 ~ 0.59	2.85 ~ 3.75	9.95 ~ 10.63	19.08 ~ 19.79	28.01 ~ 28.13
P_4		0.15 ~ 0.31	1.42 ~ 1.77	4.88 ~ 5.43	10.93 ~ 12.44
P_5		0.02 ~ 0.07	0.13 ~ 0.19	0.68 ~ 0.72	2.18 ~ 2.77
P_6				0.03 ~ 0.08	0.22 ~ 0.32
P_7					0.02 ~ 0.06

Table 4: $C_{e\text{-}det}$ and $P_{f\text{-}uns}$ for Schemes 1, 2 and 3 under various injection campaigns.

Campaign / Scheme	1		2		3		4		5	
	$C_{e\text{-}det}$	$P_{f\text{-}uns}$	$C_{e\text{-}det}$	$P_{f\text{-}uns}$	$C_{e\text{-}det}$	$P_{f\text{-}uns}$	$C_{e\text{-}det}$	$P_{f\text{-}uns}$	$C_{e\text{-}det}$	$P_{f\text{-}uns}$
1	0.9931	0.0069	0.9923	0.0077	0.987	0.013	0.9866	0.0134	0.9751	0.0249
2	0.998	0.002	0.9942	0.0058	0.9938	0.0062	0.9925	0.0075	0.992	0.008
3	0.9949	0.0051	0.9925	0.0075	0.9884	0.0116	0.9872	0.0128	0.9805	0.0195

Table 5: Comparison of various self-checking techniques, where the data shown in the columns of Single/Double/Triple are the EDC for single/double/triple faults injected into an array multiplier.

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 [30]	1487615	95%	12.97 ns	100%	95%	90%
Bose-Lin [31]	1297667	70%	11.60 ns	100%	91%	82%
Parity [32]	915250	20%	11.83 ns	100%	65%	57%

4.3 Comparisons

As we know, the effectiveness of fault-tolerant schemes for high-performance microprocessors can be measured by: 1. hardware overhead, 2. performance degradation, 3. program space overhead, 4. error-detection coverage, 5. error-detection latency, and 6. error-recovery efficiency. Therefore, the design is to find a good trade-off among those six properties. In other words, each method may focus the attention on some design attributes and meanwhile degrade others. For instance, the compiler-based software redundancy schemes [15], [16] are able to perform the error detection without hardware modification and overhead, but have worse performance degradation and program space overhead than our method. In addition, the software redundancy schemes increase the compiler complexity as well. Further, most of the methods in the literature only address the issue of error detection and do not provide enough data, such as hardware overhead and error coverage, to allow us to compare the various schemes fairly. Besides that, the comparisons should also be based on the same or at least similar conditions, like which fault model considered. Consequently, we adopt the qualitative approach to compare our scheme with others.

The comparisons are summarized as follows: 1. our method is more complete in that it offers an effective error-handling process comprising the error detection and error recovery. In the design of fault-tolerant systems, the adopted error-detection scheme determines the error-detection latency and the length of latency could affect the implementation complexity and time efficiency of the error-recovery process. Our error-detection mechanism enjoys no detection latency such that we can utilize a very simple instruction-retry method to recover the errors in real-time manner. Other schemes with variable error-detection latency would pay a higher hardware cost to implement the error-recovery process and more execution time to correct the errors. For example, the error-detection scheme presented in [18] holds 692 cycles of detection latency on average, and 36183 cycles for the worst case. Such a lengthy latency requires more time for the error recovery. That will degrade the performance significantly once the errors occur. However, the scheme in [18] provides the detection coverage of the transient faults for the entire pipeline, whereas our scheme only covers the data paths, but can conquer the transient and permanent faults. 2. This work offers the more complete and solid results including hardware overhead, performance degradation, and fault-tolerant design metrics. Based on such results, our method can be thoroughly validated with confidence. Those data are valuable and important for us to justify whether the scheme is workable or not. Apart from that, we introduce a concept of fault scenario to imitate various degrees of fault's severity. Therefore, we can examine how strong our scheme can achieve in different fault scenarios. Through this investigation, we can gain an insight into the impact of the fault environment on the capability of our scheme.

5 Conclusions

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. A general error-detection framework based on our hybrid approach with a more rigid fault model is developed and then three representative schemes generated from the framework are used to demonstrate the spirit of our hybrid concept. A thorough evaluation of design metrics for those three demonstrated schemes was performed to characterize the effect of various complexities of hybrid designs on the hardware overhead, performance degradation and error-detection capability. The results presented exhibit several design options that our error-detection framework could furnish. Therefore, the framework provides an opportunity for the designers to choose an efficient design solution to best meet the design requirements from the possible design options offered by the framework. We should point out that a huge amount of fault injection campaigns were conducted to estimate the EDC of the schemes under a variety of fault scenarios so as to investigate the capability of our hybrid approach in different fault scenarios. Such experiments can give us more

realistic and comprehensive simulation results. The effectiveness of our mechanism even in a very severe fault scenario is justified from the experimental results. 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.

Acknowledgements. The authors acknowledge the support of the National Science Council, Republic of China, under Contract No. NSC 96-2221-E-216-006.

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] Short, M., Schwarz M. and Boercsoek J.: 'Efficient Implementation of Fault-Tolerant Data Structures in Embedded Control Software', *WSEAS TRANSACTIONS on ELECTRONICS*, 5, (1), January 2008, pp. 12-24.

[5] Hahanov, V., Hahanova, A., Chumachenko, S. and Galagan, S.: 'Diagnosis and Repair Method of SoC Memory', *WSEAS TRANSACTIONS on CIRCUITS AND SYSTEMS*, 7, (7), July 2008, pp. 698-707.

[6] Hahanov, V., Obrizan, V., Litvinova, E. and Man, K. L.: 'Algebra-Logical Diagnosis Model for SoC F-IP', *WSEAS TRANSACTIONS on CIRCUITS AND SYSTEMS*, 7, (7), July 2008, pp. 708-717.

[7] Sohi, G. S., Franklin, M. and Saluja, K. K.: 'A Study of Time-Redundant Fault Tolerance Techniques for High-Performance Pipelined Processors,' *19th IEEE FTCS*, 1989, pp.436-443.

[8] Holm, J. G. and Banerjee, P.: 'Low Cost Concurrent Error Detection in A VLIW Architecture Using Replicated Instructions', *Intl. Conf. on Parallel Processing*, 1992, pp. 192-195.

[9] 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)*, 1995, pp. 207-215.

[10] Chen, Y. Y.: 'Concurrent Detection of Control Flow Errors by Hybrid Signature Monitoring', *IEEE Trans. on Computers*, 54, (10), October 2005, pp. 1298-1313.

[11] Rotenberg, E.: 'AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors', *29th IEEE FTCS*, 1999, pp.84-91.

[12] Rebaudengo, M. et al.: 'Soft-Error Detection through Software Fault-Tolerance Techniques', *DFT'99*, 1999, pp. 210-218.

[13] 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.

[14] Kim, S. and Somani, A. K.: 'SSD: An Affordable Fault Tolerant Architecture for Superscalar Processors', *Pacific Rim Intl. Symposium. On Dependable Computing*, 2001, pp. 27-34.

[15] Oh, N., Shirvani, P. P. and McCluskey, E. J.: 'Error Detection by Duplicated Instructions in Super-Scalar Processors', *IEEE Trans. on Reliability*, 51, (1), March 2002, pp. 63-75.

[16] Bolchini, C.: 'A Software Methodology for Detecting Hardware Faults in VLIW Data Paths', *IEEE Trans. on Reliability*, **52**, (4), 2003, pp. 458-468.

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

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

[19] Kwak, S. W. and Kim, B. K.: 'Task-Scheduling Strategies for Reliable TMR Controllers Using Task Grouping and Assignment', *IEEE Trans. on Reliability*, 49, (4), December 2000, pp. 355-362.

[20] Avizienis, A., Laprie, J.-C., Randell, B. and Landwehr, C.: 'Basic Concepts and Taxonomy of Dependable and Secure Computing', *IEEE Trans. on Dependable and Secure Computing*, 1,

(1), Jan.-March 2004, pp. 11-33.

[21] 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.

[22] Lala, P. K.: 'Self-Checking and Fault-Tolerant Digital Design' (*MORGAN KAUFMANN*, 2001).

[23] Johnson, B. W.: 'Design and Analysis of Fault Tolerant Digital Systems', Addison Wesley, 1989.

[24] Mitra, S., Saxena, N. R. and McCluskey, E. J.: 'Common-Mode Failures in Redundant VLSI Systems: A Survey', *IEEE Trans. on Reliability,* 49, (3), Sept. 2000, pp. 285 – 295.

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

[26] Hsueh, M. C., Tsai, T. K. and Iyer, R. K.: 'Fault Injection Techniques and Tools', *IEEE Computer*, 30, (4), April 1997, pp. 75-82.

[27] Constantinescu, C.: 'Experimental Evaluation of Error-Detection Mechanisms', *IEEE Trans. on Reliability*, 52, (1), March 2003, pp. 53-57.

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

[29] Gracia, J. et al.: 'Comparison and Application of Different VHDL-Based Fault Injection Techniques', *DFT'01*, 2001, pp. 233-241.

[30] 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.

[31] Bose, B., and Lin, D. J.: 'Systematic Unidirectional Error Detecting Codes', *IEEE Trans. on Computers*, **34**, (11), 1985, pp. 1026-1032.

[32] Nicolaidis, M. et al.: 'Fault-Secure Parity Prediction Arithmetic Operators', IEEE Design & Test of Computers, 14, (2), 1997, pp. 60-71.