Datapath Error Detection Using Hybrid Detection Approach for High-Performance Microprocessors

YUNG-YUAN CHEN, KUEN-LONG LEU† AND KUN-CHUN CHANG
Department of Computer Science and Information Engineering
Chung-Hua University, Hsin-Chu, TAIWAN

†Department of Electrical Engineering
National Central University, Chung-Li, Taiwan

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 and performance 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 performed 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]. Such an influence raises the urgent need to incorporate the fault tolerance into the high-performance microprocessors for safety-critical applications. Concurrent error detection provides an effective approach to detect the errors caused by transient and intermittent faults. 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 [2-3] 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 [3] 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 three schemes with various hardware redundancy, time redundancy and error-detection coverage (EDC). Experimental results are given in Section 3. The conclusions appear in Section 4.
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 [4] (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 [5]. Such a SEU effect could seriously degrade the EDC for self-checking circuits generally designed with a single fault assumption [5]. 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.

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 to fulfill the requirement of zero detection latency. Our hybrid approach is quite comprehensive in that it offers the TMR and comparison approaches of 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 $x$;
- $n_{sc}$: Number of modules equipped with self-checking ability in the $n$ modules of type $x$, $0 \leq n_{sc} \leq n$.
- $s$: Number of spare modules added to type $x$, $s \geq 0$.
- $s_{sc}$: Number of spare modules equipped with self-checking capability in the $s$ spares of type $x$, $0 \leq s_{sc} \leq s$.
- $m$: 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.

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 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. The ETMR/ECMP schemes can be aware of common-mode failures by using the error signals delivered from the self-checking units. It is evident that the EDC of self-checking technique decides the detection capability of the common-mode failures for ETMR/ECMP.

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 output 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:** If the self-checking results show no errors, 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.

**Situation 2:** There are two cases to consider.

- **Case 1:** At least one module out of all the modules having identical outputs provides 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.

- **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 \( n, n_{s-c}, s \) and \( s_{s-c} \), 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 \( n, n_{s-c}, s \) and \( s_{s-c} \), the expression \( 2(m - n_{s-c} - s_{s-c}) \leq (n - n_{s-c}) + (s - s_{s-c}) \), 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. There are two cases to consider in expression (1). The first case is \( n_{s-c} + s_{s-c} = 0 \). Expression (1) becomes \( 2m \leq n + s \). 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 \( n_{s-c} + s_{s-c} \neq 0 \). In this case, if expres-
sion (1) is true, then one possibility is to let \((n_{e-c} + s_{e-c})\) 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:

\[
\text{if } (2(m - n_{e-c} - s_{e-c}) \leq (n - n_{e-c}) + (s - s_{e-c})) \text{ then}
\]

\[
\{m \text{ instructions in present packet can be executed and checked simultaneously.}
\]

\[
\text{else } \{m \text{ instructions in present packet cannot be executed and checked simultaneously.}
\]

The principle of instruction partitioning is described as follows. For two schemes, respectively. The following two equations can be examined by ETMR, ECMP, TMR, comparison and self-checking schemes, respectively. The following two equations are derived from the module resource constraint: \(3m_1 + 2m_2 + 3m_3 + 2m_4 + m_5 \leq n + s\) and \(m_{1-sc} + m_{2-sc} + m_{3-sc} \leq n_{e-c} + s_{e-c}\), where \(m_{1-sc}\) and \(m_{2-sc}\) 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 \((m_1, m_2, m_3, m_4, m_5, m_{1-sc}, m_{2-sc})\). 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 stated earlier is exploited to select a sound solution. The guideline of the selection of an effective solution is presented next.

\[
\text{if } (n_{e-c} + s_{e-c} \neq 0) \text{ then}
\]

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

\[
\text{else } \{\text{We choose a solution whose } m_3 \text{ value is maximal.}
\]

\[
\text{else}
\]

\[
\{\text{Due to lack of the enough module resources, we need to partition the current packet into two or three packets which will be executed sequentially. Such partitions will induce some extra cycles, and therefore, degrade the performance of program execution.}
\]

\[
\text{If } n \text{ is odd with no redundancy added, i.e. } s = n_{e-c} = s_{e-c} = 0, \text{ then the worst partition occurs in a packet containing } n \text{ instructions, which requires partitioning into three packets. It is easy to see that a packet normally requires partitioning into two packets except the worst partition depicted above. As a result, the partition of a packet will normally induce one extra execution cycle except the worst partition, where two extra cycles are needed.}
\]

\[
\text{Next, the principle of instruction partitioning is described as follows. For two packet’s partition, if } m \text{ is even, then we distribute } m \text{ instructions to each packet; else,}
\]

\[
\left\{ \begin{array}{l}
\frac{m}{2} + 1 \\
\frac{m}{2}
\end{array} \right.\text{ instructions to the first and second packets respectively.}
\]

\[
\text{For three packet’s partition, we first distribute } \left\lfloor \frac{m}{3} \right\rfloor \text{ instructions to each packet; next,}
\]

\[
\text{if } (m - \frac{m}{3}) \times 3 = 1, \text{ then the remaining instruction is assigned to the first packet; if } (m - \frac{m}{3}) \times 3 = 2, \text{ then the remaining two instructions are evenly distributed to the first and second packets.}
\]

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 \(n_{e-c}\) and \(s_{e-c}\), 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. We describe three schemes as follows:

Scheme 1: Self-checking technique is not employed in this scheme, i.e. \(n_{e-c} = s_{e-c} = 0\). According to error-detection framework, \((m_1, m_2, m_3, m_4, m_5, m_{1-sc}, m_{2-sc}) = (0, 0, 1, 0, 0, 0, 0)\) is selected as the solution for \(m = 1\). For \(m = 2, 2m\) is equal to \(n + s\), so \((m_1, m_2, m_3, m_4, m_5, m_{1-sc}, m_{2-sc}) = (0, 0, 2, 0, 0, 0)\). 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 instruction is required to complete the execution of three concurrent ALU instructions for concurrent error-detection need.

Scheme 2: This scheme exploits the self-checking methodology to remove the performance degradation as shown in Scheme 1. Let \(n_{e-c} = 1\), and \(s_{e-c} = 1\). Note that an ALU equipped with the self-checking mechanism comprises a self-checking adder/multiplier/logic circuit. For \(m = 1, (1, 0, 0, 0, 0, 2, 0)\) is selected as the final solution. For \(m = 2, (0, 2, 0, 0, 0, 0, 2)\) is the solution. For \(m = 3\), it satisfies the condition \((m - n_{e-c}) \cdot s_{e-c}\) = \((n - n_{e-c}) + (s - s_{e-c})\), and the solution is \((0, 0, 1, 2, 0, 0)\). 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.

**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 the Scheme 2, some of the combinations of the instruction classes in a packet containing three instructions are still required to schedule to two execution packets. For example, three instructions in a packet are all from the same instruction class, such as ‘add’ class. In this case, 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. The EDC of Scheme 3 is slightly lower than Scheme 2.

![Fig. 1: Selection criterion.](image)

### 3 Experimental Results

To evaluate our hybrid approach, Schemes 1, 2 and 3 were implemented in an experimental 32-bit VLIW core respectively. The core consists of five pipeline stages as shown in Fig. 2. This experimental architecture can issue at most three ALU and three load/store instructions per cycle. Fig. 2 shows the architectural implementation of Scheme 2. The architectures of Schemes 1 and 3 are similar to Fig. 2. 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 ‘EXE’ stage was created only for the purpose of the measurement of EDC during the fault injection campaign. The hardware implementations in VHDL and simulated fault injection experiments were performed to measure the design metrics.

The original VLIW core is termed as Scheme 0. Table 1 provides the data of hardware overhead (abbreviated as HO), performance degradation (PD) and EDC for Schemes 1, 2 and 3. The implementation technology used here is UMC 0.18μm process. The area excludes the instruction memory as well as the ‘Error Analysis’ block. 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 original VLIW core. This implies that the control task of our hybrid technique is simple and easy to implement. Eight benchmark programs including heap-sort, quick-sort, four queens, 5 × 5 matrix multiplication, FFT and IDCT (8×8), were developed to measure the performance degradation resulting from the error detection for Schemes 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.

We have conducted a huge amount of fault injection campaigns to calculate the EDC under various fault scenarios. 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 tar-
gets cover the entire ‘EXE’ stage except the ‘load/store’ and ‘Error Analysis’ units, as shown in Fig. 2. To inject the faults into the inside of the adders and multipliers, those components were implemented at the gate level; 3) fault duration = 5 clock cycles. The EDC results shown in Table 1 have 95% confidence interval of ±0.14% to ±0.98%. Five different fault scenarios with various degrees of fault severity were generated to validate the capability of the Schemes 1, 2 and 3. It is evident that the increase of the occurring probability of multiple faults will lower the EDC. As a result, EDC decreases as the fault scenario becomes worse. The rank of scheme’s EDC is Scheme 2 > Scheme 3 > Scheme 1. 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 error-detection methodology even in a very severe fault environment.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>HO</th>
<th>PD</th>
<th>EDC (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0%</td>
<td>0%</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>14.9%</td>
<td>0.6% - 34.3%</td>
<td>97.51-99.31</td>
</tr>
<tr>
<td>2</td>
<td>30.4%</td>
<td>0%</td>
<td>99.2-99.8</td>
</tr>
<tr>
<td>3</td>
<td>24.8%</td>
<td>0.01% - 15%</td>
<td>98.05-99.49</td>
</tr>
</tbody>
</table>

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

4 Conclusions
This paper presents a new, hybrid error-detection approach with no detection latency for high-performance microprocessors. 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 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. The effectiveness of our mechanism even in a very severe fault scenario is justified from the experimental results.

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

References: