Fault-Tolerant VLIW Processor Design and Error Coverage Analysis

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

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

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

1 Introduction

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

The previous researches in reliable microprocessor design are mainly based on the concept of time redundancy approach [3-9] that uses the instruction replication and recomputation to detect the errors by comparing the results of regular and duplicate instructions. The instruction replication, recomputation schedule and result comparison of regular and duplicate instructions can be accomplished either in software level – source code compilation phase to generate redundant code for fault detection [4], [7], [8] or in hardware level [3], [5], [6], [9]. In [7], [8], the authors adopted software techniques for detecting the errors in superscalar and VLIW processors respectively. The compiler-based software redundancy schemes have the advantage of no hardware modifications required, but the performance degradation

and code growth increase significantly as pointed out in [3], [5]. The hardware redundancy approach requires extra hardware and architectural modification to manage the instruction replication, recomputation and comparison to detect the errors.

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

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

2 Fault-Tolerant Data Path Design

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

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

2.1 Concurrent Error Detection and Real-Time Error Recovery

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

CED Scheme. The following notations are developed:

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

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

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

Case 2: $m \times 2 < n+s$. We can divide the instructions into two groups: G(1) and G(2). There are $m1$ instructions and $m2$ instructions in G(1) and G(2) respectively, where $m1+m2=m$, $m1, m2 \geq 0$. Each instruction in G(1) and G(2) can be examined by the triple modular redundancy (TMR) scheme and duplication with comparison, henceforth referred to as comparison, scheme respectively. It is worth noting that to deal with the correlated transient faults, which may cause the multiple module failures, the TMR scheme is enhanced to have the ability to detect the multiple module errors. The following equations and criterion are used to decide $m1$ and $m2$. The equations are $m1 \times 3 + m2 \times 2 \leq n+s$; $m1+m2=m$; $m1, m2 \geq 0$. There may have several solutions derived from the equations. Since TMR can tolerate and locate one faulty module compared to the comparison, the criterion employed is to choose a solution which has the maximal value of m_1 among the feasible solutions. In other

words, TMR has the benefit to avoid activating the procedure of error recovery while only one faulty module occurs. In contrast to TMR, comparison scheme needs to spend time for error recovery. The concern here is again the consideration of real-time applications.

Case 3: $m \times 2 > n + s$. Due to limited resources, m instructions cannot be all checked at the same cycle by TMR and/or comparison schemes. Therefore, we need to partition m instructions into several sequential execution packets such that the instructions in each packet can be examined concurrently. However, some extra cycles are required to guarantee that each instruction can be verified while it is executed. This implies that the performance of program execution will be degraded. The degree of performance degradation depends on the occurring frequency of the Case 3 during the program execution. The compromise between hardware overhead and performance degradation can be accomplished by choosing a proper s.

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

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

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

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

in the design of selection policy for TMR retry is how to compromise between the implementation complexity and the number of faulty modules being tolerated. A sound selection policy for TMR retry is presented next.

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

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

2.2 Reliable Data Path Design: Case Study

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

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

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

```
while (not end of program)
  {switch (m)
    {case '1':
```

```
        TMR_ALU(1, 2, 3); if (TMR_ALU detects more than one
        ALU failure) then the "Error-recovery process" is
        activated to recover the failed instruction.
    case '2':
        the execution packet contains two instructions:
        I1 and I2.
        I1: CMP_ALU(1, 2);   I2: CMP_ALU(3, 4);
        if (I1 fails) then the "Error-recovery process" is
        activated to recover I1.
        if (I2 fails) then the "Error-recovery process" is
        activated to recover I2.
    case '3':
        the packet is divided to two packets and executed
        sequentially.
    }}
```

Error-recovery process:

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

3 Hardware Implementation and Performance Evaluation

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

A fault-tolerant VLIW processor based on the architecture of Figure 1 and the features mentioned previously was realized in VHDL. The implementation data by UMC 0.18μm process are shown in Table 1. The area does not include the instruction memory as well as the 'Error Analysis' block. For performance consideration, we require that the clock frequency of the fault-tolerant VLIW processor must retain the

same as that of non fault-tolerant one. It is worth noting that the overhead of 'ALU_Control' unit is only 0.26 percent compared to the area of the non fault-tolerant VLIW core. This implies that the control task of our scheme is simple and easy to implement. The performance degradation caused from the CED demand is between 0.6% and 34.3% for eight benchmark programs, including heapsort, quicksort, FFT, 5×5 matrix multiplication and IDCT (8×8) etc..

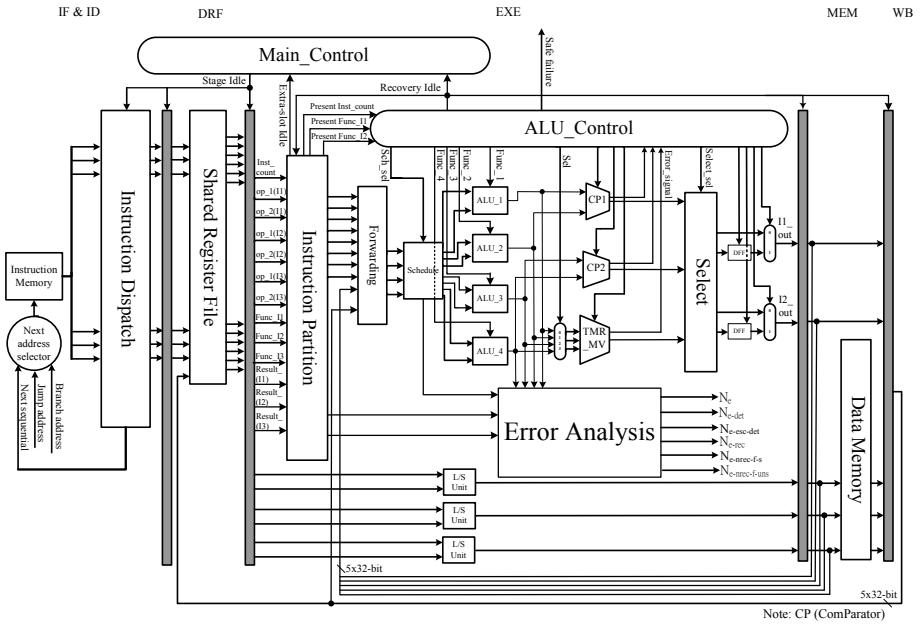


Fig. 1. Fault-tolerant VLIW architecture.

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

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

4 Error Coverage Analysis

In this section, the error coverage analysis based on the fault injection [13] is conducted to validate our scheme. A comprehensive fault tolerance verification platform comprising a simulated fault injection tool, ModelSim VHDL simulator and data analyzer has been built. It offers the capability to effectively handle the operations of fault injection, simulation and error coverage analysis. The core of the verification platform is the fault injection tool that can inject the transient and permanent faults into VHDL models of digital systems at chip, RTL and gate levels during the design phase. The tool adopts the built-in commands of VHDL simulators

to inject the faults into VHDL simulation models. Injection tool can inject the following classes of faults: '0' and '1' stuck-at faults, 'Z': high-impedance and 'X': unknown faults. Weibull fault distribution is employed to decide the time instant of fault injection.

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

4.1 Fault-Tolerant Design Metrics

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

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

- Pf_{-uns} : Probability of system entering the fail-unsafe state;
- $Ce_{-\det}$: Error-detection coverage, i.e. probability of errors detected;
- Ce_{-rec} : Error-recovery coverage, i.e. probability of errors recovered given errors detected;
- Ce : Error coverage, i.e. probability of errors detected and recovered;
- Pf_{-s} : Probability of system entering the fail- safe state;
- $Pt_{-\det-f-s}$: State transition probability from 'detected' state to 'fail-safe' state.
- $Pt_{-\det-f-uns}$: State transition probability from 'detected' state to 'fail-unsafe' state.
- $Pf_{-uns-\det}$: Probability of system entering the fail-unsafe state due to the detection defects stated earlier;
- $Pf_{-uns-rec}$: Probability of system entering the fail-unsafe state due to the recovery defects stated earlier;

The parameters Ne , $Ne_{-\det}$, $Ne_{-esc-\det}$, Ne_{-rec} , $Ne_{-nrec-f-s}$ and $Ne_{-nrec-f-uns}$ (called the error-related parameters) represent the total number

of errors occurred, the number of errors detected, the number of errors escape being detected, the number of errors recovered, the number of errors not recovered and system enters the 'fail-safe' state and the number of errors not recovered and system enters the 'fail-unsafe' state, respectively. The design metrics can be expressed as follows:

$$Pf-uns-\det=\frac{Ne-esc-\det}{Ne};Ce-\det=\frac{Ne-\det}{Ne}=1-Pf-uns-\det;Ce-rec=\frac{Ne-rec}{Ne-\det}. \quad (1)$$

$$Pt-\det-f-s=\frac{Ne-nrec-f-s}{Ne-\det};Pt-\det-f-uns=\frac{Ne-nrec-f-uns}{Ne-\det}; \quad (2)$$
$$Pf-uns-rec=Ce-\det\times Pt-\det-f-uns.$$

$$Pf-s=Ce-\det\times Pt-\det-f-s;Pf-uns=Pf-uns-\det+Pf-uns-rec;$$
$$Ce=Ce-\det\times Ce-rec;Ne=Ne-\det+Ne-esc-\det; \quad (3)$$
$$Ne-\det=Ne-rec+Ne-nrec-f-s+Ne-nrec-f-uns.$$

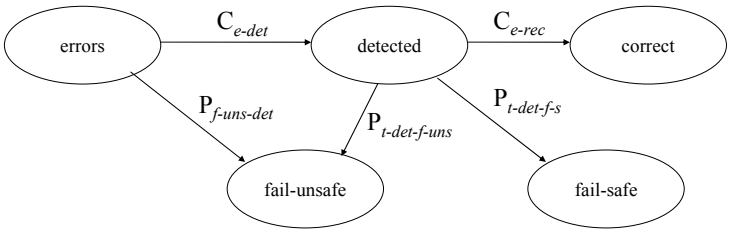


Fig. 2. Predicate graph of fault-tolerant mechanism.

4.2 Simulation Results and Discussion

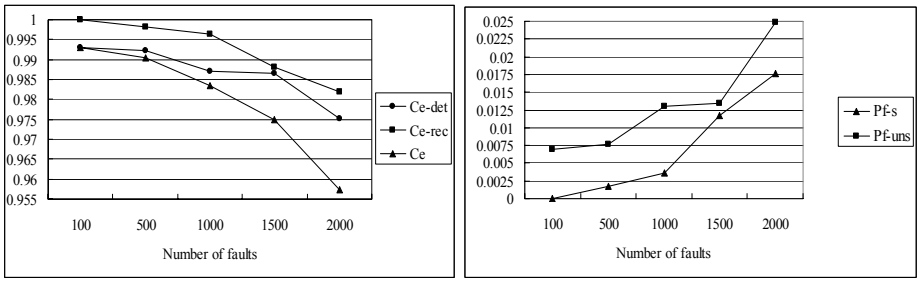
We have conducted a huge amount of fault injection campaigns to validate the proposed fault-tolerant VLIW scheme under various fault situations. We performed a comprehensive experiment to explore a particular fault-related parameter, namely fault-occurring frequency, to see its impact on the fault-tolerant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

Fault-Occurring Frequency. The goal of this experiment is to observe the effect of the fault-occurring frequency on the design metrics depicted in Section 4.1. In this experiment, we copy each of the following benchmark programs: '*N! (N=10)*', '*5×5 matrix multiplication*', '$2\sum_{i=1}^{5} Ai\times Bi$', four times and then the twelve programs are

combined in random sequence to form a workload for the fault simulation. The length of workload is equal to 4384 (clocks) ×30 (ns/clock).

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

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



(a) (b)

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

5 Conclusions

This paper presents a new fault-tolerant framework for VLIW processors that focuses mainly on the reliable data path design. Based on a more rigid fault model, a CED and real-time error recovery scheme is proposed to enhance the reliability of the data paths. Our approach provides the design compromise between hardware overhead, performance degradation and fault tolerance capability. This framework is quite useful in that it can give the designers an opportunity to choose an appropriate

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

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

References

1. Huck, J. et al.: Introducing the IA-64 Architecture. *IEEE Micro*, Vol. 20, issue: 5, pp. 12-23, Sep.-Oct. 2000.
2. Karnik, T., Hazucha, P., Patel, J.: Characterization of Soft Errors Caused by Single Event Upsets in CMOS Processes. *IEEE Trans. on Dependable and Secure Computing*, Vol. 1, issue: 2, pp. 128-143, April-June 2004.
3. Nickle, J. B., Somani, A. K.: REESE: A Method of Soft Error Detection in Microprocessors. *DSN'01*, pp. 401-410, 2001.
4. Holm, J. G., Banerjee, P.: Low Cost Concurrent Error Detection in A VLIW Architecture Using Replicated Instructions. *Intl. Conf. on Parallel Processing*, pp. 192-195, 1992.
5. Franklin, M.: A Study of Time Redundant Fault Tolerance Techniques for Superscalar Processors. *IEEE Intl. Workshop on Defect and Fault Tolerance in VLSI Systems (DFT'95)*, pp. 207-215, 1995.
6. Kim, S., Somani, A. K.: SSD: An Affordable Fault Tolerant Architecture for Superscalar Processors. *Pacific Rim Intl. Symposium. On Dependable Computing*, pp. 27-34, 2001.
7. Oh, N., Shirvani, P. P., McCluskey, E. J.: Error Detection by Duplicated Instructions in Super-Scalar Processors. *IEEE Trans. on Reliability*, Vol. 51, (1), pp. 63-75, March 2002.
8. Bolchini, C.: A Software Methodology for Detecting Hardware Faults in VLIW Data Paths. *IEEE Trans. on Reliability*, Vol. 52, (4), pp. 458-468, December 2003.
9. Qureshi, M. K., Mutlu, O., Patt, Y. N.: Microarchitecture-Based Introspection: A Technique for Transient-Fault Tolerance in Microprocessors. *DSN'05*, pp. 434 – 443, June-July 2005.
10. Mitra, S., Saxena, N. R., McCluskey, E. J.: Common-Mode Failures in Redundant VLSI Systems: A Survey. *IEEE Trans. on Reliability,* Vol. 49, (3), pp. 285 – 295, Sept. 2000.
11. Avizienis, A., Laprie, J.-C., Randell, B., Landwehr, C.: Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. on Dependable and Secure Computing*, Vol. 1, issue: 1, pp. 11-33, Jan.-March 2004.
12. Dugan, J. B., Trivedi, K. S.: Coverage Modeling for Dependability Analysis of Fault-Tolerant Systems. *IEEE Trans. on Computers*, Vol. 38, (6), pp. 775-787, June 1989.
13. Clark, J., Pradhan, D.: Fault Injection: A Method for Validating Computer-System Dependability. *IEEE Computer*, 28(6), pp. 47-56, June 1995.