Abstract

In this study, a fault-tolerant design framework of VLIW processor is proposed. Specifically, this work concentrates on the issue of dependable data path design. We first use three identical functional modules in the data paths to demonstrate our fault-tolerant technique. Basically, we add one spare module in this illustration and refine on the concepts of triple modular redundancy and comparison to achieve fault detection, fault location and error recovery. A real-time error recovery process is developed to conquer the faults. Hardware architecture and its implementation in VHDL are presented. We show that the proposed scheme can be easily extended to data paths, which contains more than three identical functional modules. In addition, for a specific number of identical modules, the fault-tolerant framework provides a design choice among several feasible solutions in terms of hardware overhead, performance degradation and dependability requirements.

1. Introduction

Nowadays, VLIW processor is a major architecture approach for high-performance computing systems. A typical example of VLIW is Intel and HP IA-64 [1]. As processor chips become more and more complicated, and contain large number of transistors, the processors have a limited operational reliability due to the increased likelihood of faults especially when the chip fabrication enters the deep submicron technology [2]. 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 [3-5].

The previous researches in reliable microprocessor design are mainly based on the concept of time redundancy approach [3-5] that uses the instruction replication and recomputation to detect the errors by comparing the results of regular and duplicate instructions. Most of the papers adopt the hardware approach to manage the instruction replication, recomputation and comparison to detect the errors.

The deficiency in previous research is summarized as follows. First, most of the studies do not perform the evaluation of hardware overhead and do not pay their attentions to the part of error recovery. Secondly, the performance degradation due to the addition of fault detection capability into the system is significant during program execution even in fault-free situation and if the error recovery time is counted, the performance will become worse. Moreover, the performance analysis only considers the performance degradation that is resulted from the fault detection under fault-free condition. They are short of the analysis of error recovery time needed to overcome the transient faults.

2 Fault-Tolerant Design Framework

Before we begin introducing our approach, a fault model adopted in this study is described below.

Fault model: a fault inside functional module can affect either single bit or multiple bit signals simultaneously, and multiple faults may exist and become active at the same time. A fault could be permanent or transient types. We assume that at most a fault-free module can be intruded upon the faults at a time, and independent faults in various functional modules won't produce identical erroneous outputs. The assumptions are reasonable in that the probability of violating the assumptions is expected to be very low. Moreover, the output errors may occur simultaneously at multiple modules.

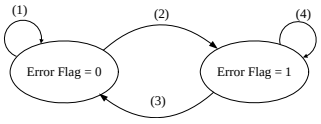
Besides the fault model, we define a hidden fault as the fault stays inside a module and it is not detected, because the operations performed in the module cannot trigger this fault. Consequently, the fault hides inside the module

2.1 Reliable Data Path Design: A Case Study

In this illustration, three ALU modules plus one spare ALU are offered in the VLIW processor core. The four ALU modules are termed ALU_A to ALU_D. At most three ALU instructions can be issued concurrently. The fault-tolerant approach is presented as follows.

Fault-tolerant approach:

Each ALU module uses an error flag to record its faulty status. If error flag is zero, it indicates that this module contains either no faults or hidden faults. Once the faults are detected, the error flag is set to one. This module could be retested later and if faults disappear or faults still exist but not detected by this retest, the error flag is reset to zero again. The error flag will remain at one, if faults do not vanish and detected from the retest step or faults disappear but no retest applies. Figure 1 shows the states of an error flag.



(1): No fault or hidden faults; (2): Faults detected;
(3): Faults disappear or faults not detected by the retest;
(4): Faults detected or faults vanish and no retest applies since faults go away.

Figure 1. States of an error flag.

The state of data paths, ALU_state, could be
S_0: all error flags of ALUs are zero; S_1: an error flag is one and the rest are zero; S_2: two error flags among four are one and the other two are zero; S_3: fail-safe state.

Number of instructions executed concurrently in ALU modules can be 0/1/2/3, respectively. The basic idea of our approach is to recover the execution errors promptly for each instruction run. In other words, 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 inside

the ALUs. For one or two instructions executed at the same cycle, we develop the fault-tolerant schemes based on the refined TMR and comparison mechanisms to achieve fault detection, fault location and error recovery. Three concurrent ALU instructions need to be scheduled to two sequential execution slots where a slot contains two instructions and the other slot contains the rest one; and therefore one extra ALU cycle is needed to complete it under the fault-free condition.

Even though the probability is expected to be quite low in that the hidden faults exist in some modules plus new faults occur in a faulty-free module, and the faults become effective errors concurrently at several modular outputs, we still take the situations into account for stringent dependability and safety consideration. Also, it is difficult to distinguish the permanent faults from transient faults. For performance consideration, when control unit of data paths receives an abnormal signal from ALU modules to indicate that at least an instruction is not correctly executed in ALU modules, the control unit will try to recover the errors by using other same kind of functional modules to assist the system in generating the correct results in the subsequent one or two ALUs' cycles. In other words, while some of ALUs could have transient or permanent faults inside, the top priority of control unit is to utilize the ALU resources as much as possible and to aid the system in producing the right outputs as quick as possible. Thus, the program execution can carry on without lengthy error recovery process. A retest can be applied to a module detected faulty before and if faults disappear or the present operation cannot stir up the faults, this previous faulty module is restored to its operation. Our approach provides the capabilities of on-line fault diagnosis and real-time error recovery. Its complete algorithm is described in the following.

- ALU_instruction_number: number of instructions issued to ALU modules simultaneously;

{Case ALU_state in

S_0: if (ALU_instruction_number = 1)

then { TMR(ALU_A, ALU_B, ALU_C);

if (a faulty module is discovered) then ALU_state is changed to S_1;

if (TMR output is not correct) then control unit is responsible for invoking an error recovery procedure termed retry(ALU_A, ALU_B, ALU_C, ALU_D); }

else /* this is the case of two ALU instructions run at the same cycle. */
Basically, we assign each instruction to two ALU modules and adopt a comparator to detect the errors. There are four possible outcomes for the executions.
1. Two comparator results are correct;
2. One of comparator detects an error and informs control unit about this error. The control unit discards the incorrect result first, and sends a recovery signal out. In addition, the control unit activates an error recovery by reassigning the failed instruction to the other two modules where the instruction is executed correctly at last run. The rerun may succeed or not. If rerun is successful, the correct output is utilized to identify the faulty modules in the previous run and the data path state changes to S_1 or S_2 that depends on the comparison outcomes. The rerun may fail due to the hidden faults and/or new faults occur during the recovery process. Under the circumstances, control unit is responsible for invoking an error recovery procedure called retry(ALU_A, ALU_B, ALU_C, ALU_D);
3. Two comparator results are incorrect; the control unit will try to conquer this situation by rerunning both instructions sequentially and each instruction is recomputed by retry(ALU_A, ALU_B, ALU_C, ALU_D).

S_1: if (ALU_instruction_number = 1)

then { TMR(using three modules having zero in their error flag);

if (a faulty module is discovered) then ALU_state transits to S_2.

if (TMR operation fails) then control unit is responsible for invoking an error recovery procedure called retry(ALU_A, ALU_B, ALU_C, ALU_D); /*TMR operation could fail because two or three modules become faulty. In this situation, it is worth to rerun the instruction by all ALUs, since the module whose error flag is one does not retest in this run and its faults may vanish already or the current instruction won't stir up the faults in this module. Thus, the system still has an opportunity to overcome this crisis. Moreover, most of faults are transient kind, so it is expected that a faulty module has a high probability to be restored later. The control unit will update the ALU_state according to the results of retry(ALU_A, ALU_B, ALU_C, ALU_D). */}

else Basically, we assign each instruction to two ALU modules and adopt a comparator to detect the errors. There are four possible outcomes for the executions.
4. Two comparator results are correct; it means that a module whose error flag is one is retested and the retest does not discover an error at the modular output; so, its corresponding error flag is reset to zero and the state changes to S_0. One thing should be pointed out that this retest result is only based on the current instruction executed in that module. Obviously, it is not thorough test for the module and therefore we cannot assert whether the faults disappear or they become hidden faults from this retest point of view.
5. The same as item 2 except ALU_state may transit to S_0 or S_2.
6. Two comparator results are incorrect: the same as 3.
S_2: This part is similar to **S_1**;
S_3: Fail-safe state;
Case End}

TMR(ALU_X, ALU_Y, ALU_Z): The concept of triple module redundancy (TMR) is employed to mask a single modular failure. In addition, we enhance the TMR methodology by supplementing the following functions.

The output of TMR compares with each ALU output. If an ALU output disagrees with TMR output, the faulty module can be identified and its error flag is set to one. Error recovery is not required under the circumstances. While more than one ALU output disagree with TMR output, at least two modules have encountered the faults, because we assume that the independent faults in various ALUs won't produce identical faulty outputs. It should be emphasized that due to the phenomenon of hidden faults, there is a probability in which the hidden faults exist in some modules and/or new faults intrude into a fault-free module that could cause more than one ALU producing the erroneous outputs. Consequently, TMR approach cannot cope with this situation. Instead, TMR will inform the control unit of data paths about the situation.

retry(ALU_A, ALU_B, ALU_C, ALU_D): Control unit sends out a recovery signal to previous stages and discards the TMR output; invoke the instruction retry to deal with this faulty condition. Under the existing conditions, multiple ALU modules fail to operate correctly at last execution. Therefore, the failed instruction is recomputed by all ALUs, and all execution results are fed into a selection circuit, which can produce a correct output if at least two of four inputs are correct; the correct output is used to locate the faulty modules if any, and reflect present faulty status by updating the state to S_0 or S_1 or S_2 if necessary; otherwise, the selection circuit generates an invalid signal to control unit; once the control unit receives this kind of information, it will force the system into fail-safe state, S_3.

End /* end of our approach illustration */

2.2 General Framework

In last section, we use three identical modules to demonstrate our approach. The number of identical modules could be various in data paths of VLIW processors. Hence, to be generic, we show how to apply the proposed technique to 4 to 6 identical modules and briefly discuss the design consideration in terms of hardware overhead, performance degradation and fault-tolerance capability. Table 1 lists the data of design parameters for various numbers of identical ALUs.

Table 1. Data of design parameters for various numbers of identical ALUs.

No. of ALUs	No. of Spares	Hardware Overhead	Instruction Rescheduling	Fault-Tolerant Scheme
3	1	33.3%	3→ (2, 1)	1: TMR; 2: Comparator
4	0	0%	3→ (2, 1); 4→ (2, 2)	1: TMR; 2: Comparator
4	1	25%	3→ (2, 1); 4→ (2, 2)	1: TMR; 2: (1: TMR;1: Comparator)
4	2	50%	4→ (2, 2)	1: TMR; 2: TMR; 3: Comparator
5	0	0%	3→ (2, 1); 4→ (2, 2); 5→ (2, 2, 1)	1: TMR; 2: (1: TMR, 1: Comparator)
5	1	20%	4→ (2, 2); 5→ (2, 3);	1: TMR;2:TMR; 3: Comparator
6	0	0%	4→ (2, 2); 5→ (2, 3); 6→ (3, 3);	1: TMR;2 TMR; 3: Comparator

The ratio of hardware overhead in Table 1 simply counts the part of ALU itself and does not include other circuit portion resulted from the demand of fault tolerance. The notations shown in fault-tolerant scheme column represent the methodologies used to detect/locate/recover the faults/errors. For instance, 2: (1: TMR, 1: Comparator) represents an execution slot of two instructions where an instruction is executed in TMR technology and the other one is run in comparison scheme.

It is evident that the performance degradation due to fault detection is caused from the partition of instructions into several sequential execution slots. For example, in the case of three ALUs, due to limited resources, we need to schedule an execution slot of three instructions into two slots where a slot contains two instructions and the other has one instruction. We use the notation 3→ (2, 1) to represent the partition, which will induce an ALU cycle's performance degradation. Another source of performance degradation results from the error recovery that needs one or two ALU cycles to perform the recovery. Clearly, the error recovery is very effective from the performance point of view. The performance degradation depends on the frequency of instruction rescheduling plus the error recovery time. Since the faults occur infrequently and most of them are transient type, so we should emphasize that the performance degradation comes from mainly for the purpose of fault detection. The performance degradation caused from the error recovery is insignificant compared with the degradation due to fault detection.

As can be seen from Table 1, our approach offers several design options for a specific number of identical modules. The option is based on the design metrics of hardware overhead, performance degradation and fault-tolerance capability. Therefore, it is quite interesting to analyze the trade-offs among those design metrics. Without loss of generality, we use three ALUs and four ALUs as examples to explain the design trade-offs. First, a spare ALU is required for the case of three identical ALUs in order to prevent severe performance degradation. We know that an execution slot containing two instructions is very common and if no spare added, one of instructions cannot perform the comparison to check its result at once. So, the slot needs to be partitioned into two slots and it will induce an extra ALU cycle to achieve the fault detection for each instruction execution. Hence, a spare cost is paid to lower the performance degradation. Second, in four identical ALUs, there is no difference for performance degradation caused from fault detection between design option of no spare and one spare. But, one spare design enjoys less error recovery time than the design of no spare added. To make this clear, we assume some faults intrude into a module. For no spare option, it always requires an ALU cycle to recover the errors while an execution slot has two instructions. Contrarily, one spare design seldom

needs a cycle to overcome the errors because TMR mechanism is employed here. Generally speaking, more spares result in higher hardware overhead but lower performance degradation and better dependability.

It is well known that the average number of instructions issued simultaneously is restricted to 2-3. Consequently, if number of identical modules increases, more idle modules can be utilized per cycle to perform the error detection and error recovery to tolerate the faults. The necessity to add spares decreases. For instance, we allow the instruction rescheduling to occur for execution slots exceeding three instructions. Under this constraint, from Table 1, we can see that two spares (50% hardware overhead), one spare (20%) and zero spare (0%) redundancy are required in the design of 4, 5 and 6 ALUs respectively. It is also apparent that the frequency of instruction rescheduling affects the degree of performance degradation and is the major contribution to performance degradation in our fault-tolerant scheme. In a word, hardware overhead and performance degradation of our approach decreases while the number of identical modules increases in the data path of VLIW processors.

3. Hardware Architecture and Performance Analysis

An experimental structure of the proposed fault-tolerant VLIW processor is illustrated in Figure 2. The fault-tolerant VLIW processor based on the architecture of Figure 2 and the features mentioned previously are realized in VHDL.

Table 2 lists the hardware overhead of the implementation shown in Figure 2. The areas do not include the instruction and data memory.

Table 2: Comparison of fault-tolerant and non fault-tolerant VLIW core.

	Area (gate count)	Overhead	System clk (MHz)	Degradation
Non fault tolerant VLIW	82852		55.6	
Our approach	100727	21.6%	55.6	0 %

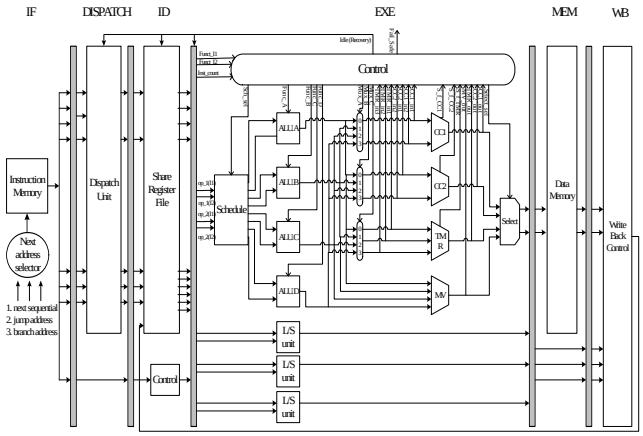


Figure 2. Fault-tolerant VLIW architecture.

Table 3 shows the results of degradation for several programs.

Table 3. The results of performance degradation for several programs.

	Performance Degradation
summation of product	6.1%
matrix multiplication	15%
factorial n	8.7%

4. CONCLUSIONS

This work presents a new fault-tolerant framework for VLIW processors that focuses mainly on the reliable data path design. Our dependable approach is able to provide a couple of design options, which offer the compromise between hardware overhead, performance degradation and fault-tolerance capability. A significant contribution of this study is to integrate the error detection and error recovery into a complete fault-tolerant VLIW system with less hardware overhead and performance degradation. The preliminary results show the effectiveness of our mechanism.

References:

[1] J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder and R. Zahir, “Introducing the IA-64 Architecture,” *IEEE Micro*, vol. 20, issue: 5, pp. 12-23, Sep.-Oct. 2000.

[2] C. Constantinescu, “Impact of Deep Submicron Techonology on Dependability of VLSI Circuits,” *Proc. Of the Int’l Conf. On Dependable Systems and Networks*, pp. 205-209, 2002.

[3] Manoj Franklin, “A Study of Time Redundant Fault Tolerance Techniques for Superscalar Processors,” *Proc. Int. Workshop on Defect and Fault Tolerance in VLSI Systems*, pp. 207-215,1995.

[4] J. B. Nickle and A. K. Somani, “REESE: A Method of Soft Error Detection in Microprocessors,” *Int. Conf. On Dependable Systems and Networks*, pp. 401-410, 2001.

[5] S. Kim and A. K. Somani, “SSD: An Affordable Fault Tolerant Architecture for Superscalar Processors,” *Pacific Rim Int’l Symp. On Dependable Computing*, pp. 27-34, 2001.