FAULT TOLERANCE
IN
ADAPTIVE REAL-TIME COMPUTING SYSTEMS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSIPHY

Shu-Yi Yu

December 2001

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

_____

Edward J. McCluskey (Principal Advisor)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

_____

Michael J. Flynn

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

_____

Nirmal R. Saxena

Approved for the University Committee on Graduate Studies:

_____

ABSTRACT

Real-time computing systems (e.g. fly-by-wire and navigation systems) often have stringent reliability and performance requirements. Failures in real-time systems can result in data corruption and lower performance, leading to catastrophic failures. Previous researchers demonstrated the use of reconfigurable hardware (e.g. Field-Programmable Gate Arrays, or FPGAs) for implementing cost effective fault tolerance techniques in general applications. In this dissertation, we developed techniques to design reliable real-time systems on FPGAs.

To demonstrate the effectiveness of our design techniques, we implemented a robot control algorithm on FPGAs. Various fault tolerance features are implemented in the robot controller to ensure reliability. Our implementation results show that the performance of the FPGA-based controller with triple modular redundancy (TMR) is comparable to that of a software-implemented control algorithm (with TMR) in a microprocessor, while providing comparable degrees of fault tolerance.

We developed a roll-forward technique for transient error recovery in TMR-based real-time systems. Our technique eliminates the need for any re-computation or rollback and therefore significantly reduces timing overhead associated with conventional recovery techniques. Markov analysis results show that our recovery scheme can significantly improve reliability of TMR systems compared to conventional approaches. Our implementation results in the robot controller design demonstrate that the roll-forward recovery scheme introduces very small area overhead.

The conventional approach to permanent fault repair in FPGAs is to reconfigure the design so that the faulty part is avoided. However, for TMR systems with high area utilization or long mission times, this approach may not be applicable due to non-availability of additional hardware resources. In such circumstances, our new permanent fault repair scheme reconfigures the original TMR-based design into another fault tolerant design of smaller area so that the faulty elements are avoided. However, unlike TMR systems, extra delays during transient error recovery may occur. Three new design techniques for this repair scheme are presented. Analytical results show that these techniques can provide high reliability while minimizing the delay overhead due to rollbacks. The effectiveness of our permanent fault repair approach is demonstrated using our FPGA-based robot controller design. A repair scheme is also designed for FPGA interconnects. Unlike conventional schemes that use redundant buses, our scheme only needs a spare wire. Our scheme can repair systems from failures caused by single faulty wire connecting FPGAs.

ACKNOWLEDGEMENTS

I express my deepest gratefulness to my advisor, Prof. Edward J. McCluskey, for his constant guidance, support, and encouragement during my years at Stanford. He embodies many of the great qualities and characteristics to which I aspire. I have been inspired by his enthusiasm for both work and life. It has always been an honor and pleasure to work with this world-class professor. I would also like to thank Mrs. Lois Thornhill McCluskey for all her help.

I would like to thank Prof. Michael J. Flynn for being my associate advisor and on my dissertation reading committee. I would also like to thank Dr. Nirmal R. Saxena for sitting on my orals and reading committees, and for his advice on my research. I would like to thank Prof. John Gill and Prof. Arogyaswami Paulraj for being on my orals committee.

I would like to thank my colleagues (RATS) and the Center for Reliable Computing for helpful discussions, critical reviews of papers, and companionship through the years: Almad Al-Yamani, Mehmet Apaydin, Dr. LaNae Avra, Dr. Jonathan T.-Y. Chang, Ray Chen, Eddie Yu-Che Cheng, Kan-Yuan Cheng, Dr. Wei-Je Huang, Sam Kavusi, Chatchai Khunpitiluck, Moon-Jung Kim, Chien-Mo Li, Dr. Siyad Ma, Dr. Samy Makar, Dr. Subhasish Mitra, Dr. Nahmsuk Oh, Dr. Philip Shirvani, Mehdi Tahoori, Dr. Nur Touba, Chao-Wen Tseng, Sungroh Yoon, Yoonjin Yoon, and Steve Zeng. I want to especially thank Dr. Santiago Fernandez-Gomez for his encouragement and technical support. I am greatly indebted to Siegrid Munda for her excellent administrative support.

I wish to thank my professors in National Taiwan University, and my uncle, Wen-Yaw Chung, for their advices and encouragement in my career and studying.

I would like to thank all my friends for their support and encouragement. Jen-Tsan Chi was a great help to me when I first came to US. Jun-Dar Su gave me constant encouragement and support during my Ph.D. study. And many thanks to Esther Chiang, Pei-Jen Hsu, Ta-Chien Huang, Jean Lee, Linda Kuo, Shi-Chin Ou Yang, Danielle Yesavage, and Jerome Yesavage for making my life more enjoyable. I want to especially thank Danielle for her proof reading of this dissertation. I also want to thank Stanford Taiwanese Student Association (STSA) for making our lives more colorful.

Finally, I wish to thank my parents and sister for their support, love, and many prayers. They have always believed in me and always been there for me. I dedicate this dissertation to my dear parents.

My research work at Stanford was supported by Defense Advanced Research Projects Agency (DARPA) under Contract No. DABT63-97-C-0024.

To my parents,

For their love, encouragement, and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

Chapter 1
Introduction

1.1 Motivation

Real-time computing systems are extensively used in our daily lives. Cars, telephone networks, and patient care systems, all contain real-time computing systems. A *real-time computing system* is a computing system in which timeliness is as important as correctness of its outputs [Laplante 97]. In real-time systems, a delayed output is not acceptable even if it has a correct value. For example, a delay in a telephone network can make human voice unrecognizable. In some applications, a delayed output is not only unacceptable but may also cause severe consequences. For instance, a delayed braking signal in a cruise control system of a car may cause a car accident, and a delayed output in an anesthesia control system may give incorrect dosage of medicine to a patient. Real-time systems are not necessarily fast systems, but their outputs have to meet strict time constraints. Hence, predictable performance is needed and is very important in these systems.

Many real-time computing systems are used under hazardous or remote circumstances, such as in nuclear power plants [Kim 96], aircraft [Girard 00], and spacecraft [Wu 93]. In these environments, computing systems are highly susceptible to errors due to radiation. Maintenance and repair is usually very expensive and time consuming for these applications. Hence, besides performance, *fault tolerance*, which is the ability of a system to operate correctly in spite of the occurrence of faults [Siewiorek 00], becomes a very important issue.

Most real-time systems are based either on Application Specific Integrated Circuits (ASICs) [Kocsis 92] or on general-purpose computers, such as microprocessors [Khatib 95], mainframe [Turver 93], and microcontrollers [Cooling 97]. ASIC-based systems usually have better performance than designs on general-purpose computers because ASIC designs can be optimized according to application requirements. However, the design cost of ASICs is usually high. In addition, once manufactured, ASIC devices can never be updated, upgraded, or corrected in any way. In mission-critical applications, any physical damage to ASIC devices cannot be repaired unless the entire device

replaced by a new one. For systems based on general-purpose computers, designs can be updated or upgraded by uploading a new software program. However, for physical chip failures, general-purpose computers share the same problem as ASICs in that the only repair solution is through device replacement. Device replacement is generally expensive. Hence, new economical techniques for providing fault tolerance in real-time applications need to be developed.

The invention of reconfigurable hardware provides solution for bringing fault tolerance into mission-critical real-time computing systems. Reconfigurable hardware, such as Programmable Logic Arrays (PLAs) or Field-Programmable Gate Arrays (FPGAs), can be reprogrammed into different devices by downloading different configuration data to the chip. Hence, by reconfiguration, design errors can be corrected even after implementation, and physically damaged parts of a chip can be avoided by downloading a new configuration that does not use the damaged parts. PLAs are capable of implementing two-level sum-of-product functions, and FPGAs are capable of implementing multi-level functions. In this dissertation, our discussion is limited to FPGAs, since they have a larger capacity compared to other types of reconfigurable hardware.

The key to the reconfiguration technology is memory cells in reconfigurable devices. In reconfigurable hardware, a circuit configuration is stored in memory cells. By downloading a new configuration to the memory cells, the system function can be altered. Reconfigurable hardware is available in many different memory technologies, including Erasable Programmable Read Only Memory (EPROM), Electrical-Erasable Programmable Read Only Memory (EEPROM), and Static Random Access Memory (SRAM) [Hauck 98]. The advantage of EPROM and EEPROM is that the configuration is nonvolatile; hence, system functionality is retained even when the power is off. However, the high voltage needed for reprogramming makes these devices less popular than SRAM-based FPGAs. SRAM cells make the reconfiguration process much easier than other types of memory cells. In an SRAM-based FPGA, we can easily modify downloaded circuits by reprogramming the contents of SRAM cells, where the configuration of circuits is stored.

Different techniques are developed for FPGAs to make use of their reconfiguration ability and make FPGAs more suitable for reliable computing in mission-critical applications. FPGA vendors have built new features that make them more immune to Single Event Upsets (SEUs), which are transient errors caused by radiation [Shirvani 01], and reduce the time needed for reconfiguration [Xilinx 01]. Researchers have developed different techniques to repair FPGA-based designs from permanent faults [Huang 01b]. Moreover, new fabrication technology and new architectures have greatly improved FPGA capacity and performance [Altera 01]. These improvements are described in detail in the following paragraphs.

SRAM-based FPGAs are more susceptible to SEUs than ASICs and general-purpose computers since one SEU can alter the logic function implemented on FPGAs [Hauck 98]. However, a new feature of configuration readout in FPGAs helps to solve the problem. For example, in [Conde 99], a controller reads the FPGA configuration out automatically and compares it with the original configuration in a flash memory. If a mismatch is detected, the original configuration is loaded into the FPGA. In [Alfke 98], the FPGA configuration is read out from three FPGAs and the voted configuration result is written back to FPGAs. In addition to the readout technique, many radiation-hardened FPGAs are also available to protect FPGAs from SEU [Xilinx 01] [Actel 01]. To reduce reconfiguration time, the partial reconfiguration technique in Xilinx Virtex FPGAs enables reconfiguration of only part of an FPGA instead of the whole chip [Xilinx 01]. Moreover, new types of FPGA can be reconfigured without affecting system operation on the chip. When an SEU occurs and an error is detected in FPGA configuration, system operation needs not be stopped and a correct configuration can be downloaded into the affected part of an FPGA quickly.

Much research has been done to exploit the available reconfiguration techniques to provide fault tolerance in FPGA-based computing systems. They utilize reconfiguration to remove permanent faults from designs on FPGA. One approach is to generate a new configuration after permanent faults are detected in computing systems. CAD tools and algorithms for fast FPGA re-routing and re-mapping are described in [Emmert 98] [Dutt 99]. Another approach is to generate pre-compiled alternative FPGA configurations and store the configuration bit maps in non-volatile memory, so that when

permanent faults are present, a new configuration can be chosen without the delay of re-routing and re-mapping [Lach 99] [Huang 01b]. With these repair schemes, permanent faults in an FPGA can be repaired as long as there are sufficient routable fault-free elements on the chip so that designs can avoid using faulty elements. These permanent fault repair schemes can be combined with transient error recovery techniques to successfully recover an FPGA-based design from either transient errors or permanent faults. These repair schemes have been utilized in different dependable computing architectures to provide fault tolerance [Emmert 00] [Saxena 00] [Mitra 00a]. Among these architectures, Stanford's Dual-FPGA architecture uses only reconfigurable hardware and memory devices [Mitra 00a]. Repair controllers are built in each of the FPGAs to perform reconfiguration tasks. Because no non-reconfigurable hardware is used, the need of device replacements for repair is eliminated in this architecture.

Recently, the rapid development of process technology has brought significant computation power to reconfigurable hardware. Current commercially available FPGAs contain up to 10 million gates per chip and can be operated at several hundred MHz [Xilinx 01] [Altera 01]. Because of the increased capacity, performance, and low implementation cost, FPGAs have been extensively used in many commercial applications, such as logic emulation [Quickturn 01], fast chip-to-chip communication [Motorola 01], optical network [AMCC 01], and image processing [Actuality 01]. Many companies have designed high-performance general-purpose processor cores for FPGA implementation [WindRiver 01]. Moreover, FPGAs are even used in space applications, such as electronic devices in cameras in NASA Mars Pathfinder [Reynolds 01].

In summary, FPGAs are commercially available, and can provide large capacity, high performance, and permanent fault repair in computing systems. Therefore, it is possible to use FPGAs as an alternative solution to fault tolerant real-time systems. We want to construct real-time computing systems on FPGAs and provide fault tolerance in computing systems and retain performance so that deadlines can still be met. We concentrate on the Dual-FPGA architecture because it comprises only FPGAs and memory, and avoids the need for device replacement.

To construct reliable real-time computing systems using reconfigurable hardware, the following research is needed and is presented in this dissertation:

- Feasibility of constructing real-time computing systems on reconfigurable hardware. Feasibility demonstration is needed to show that reconfigurable hardware can bring fault tolerance to a real-time computing system without loss of performance compared with conventional non-reconfigurable hardware.
- Transient error recovery with small performance impact. When transient errors occur in real-time computing systems, these systems need to be recovered with very small time delay so that transaction deadlines are not missed.
- Permanent fault repair with small performance impact when the available chip area is reduced because of hardware damage. When permanent faults occur in a computing system, some part of a chip becomes permanently faulty; hence, the actual usable area of a chip is decreased. With limited available area, system performance and availability will be degraded. A robust permanent fault repair scheme needs to be developed for real-time computing systems so that even with reduced available chip area, the impact on system performance and availability can be mitigated.
- Economical permanent fault repair for FPGA interconnects. A conventional approach for permanent fault repair is to use redundant buses for communication. However, interconnect resources of chips are limited. Therefore, a repair scheme that uses small interconnect resources needs to be developed.

1.2 Contributions

We have demonstrated that reconfigurable hardware can be used to provide both performance and fault tolerance in real-time computing systems. The techniques developed include a transient error recovery scheme and a permanent fault repair technique, and a fault tolerant communication scheme for FPGA repair controllers. The following summarizes the contributions of this dissertation:

- We showed that an Adaptive Computing System (ACS), which uses FPGAs for computation, can be used to provide both high performance and

good dependability in real-time computing systems. An example of an FPGA-implemented dependable control algorithm is presented. The flexibility of ACS is exploited by choosing the best precision for our application. Results showed that the FPGA-implemented control algorithm has comparable performance with the software-implemented control algorithm in a microprocessor. Different voting schemes are used in conjunction with multi-threading and combinational redundancy to add fault tolerance in the robot controller. We are the first to use multi-threading for fault tolerance in a real-time application. Fault-injection experiments are used to verify fault tolerance capabilities of the robot controller.

- For failures caused by transient faults, we have designed a new recovery scheme for Triple Modular Redundancy (TMR) systems so that their mission time can be increased. The scheme can effectively recover computing systems from single transient faults without introducing any re-computation delay; hence, it is suitable for real-time applications. A robot controller is used as a case study. Theoretical analysis and implementation results show that, with very small hardware overhead for recovery logic, the proposed scheme can improve the system reliability and lengthen its lifetime by an order of magnitude.
- We present a new scheme for permanent fault repair scheme when no extra routable fault-free elements are available for conventional repair schemes. Our scheme configures the original design into another fault tolerant design that has smaller area, so the damaged element can be avoided. Three new systems that fully utilize available fault-free area and provide graceful degradation on availability are presented. Analytical results show that our schemes have an improvement on availability compared to a module removal approach. The effectiveness of area reduction is demonstrated in an FPGA-based robot controller as a case study.
- We present a new fault tolerant communication scheme between FPGAs. Our scheme concentrates on the communication between repair controllers

in the Dual-FPGA architecture. The scheme can detect and repair single faults in wires connecting the two repair controllers. Unlike conventional schemes that use redundant buses, only a spare wire is needed. Parity checks and timeout are used for error detection. A command bounce strategy is used to locate a faulty wire in the communication channel. Repair is accomplished by reconfiguring both FPGAs to avoid the faulty wire.

1.3 Outline

This dissertation summarizes my work in developing techniques to provide fault tolerance in real-time computing systems using reconfigurable hardware. Detailed descriptions of results are found in the appendices, which are extended versions of published or submitted papers.

Chapter 2 gives an overview of previous research. It briefly introduces different computing system architectures using reconfigurable hardware and describes a general flow of error recovery steps for FPGA-based systems. Different transient error recovery schemes and permanent fault repair schemes are also introduced in this chapter.

Chapter 3 describes our demonstration of feasibility for constructing fault tolerant real-time systems on reconfigurable hardware. We describe our case study, a robot control system, in this chapter. The fault tolerance techniques implemented in the control system are described. Implementation of the system in both general-purpose processors and reconfigurable hardware are presented. Results of emulation of the control system are also shown and compared. The implemented fault tolerance techniques are compared using fault injection experiment results.

Chapter 4 describes a transient error recovery technique without re-computation delay. A reliability analysis of the forward recovery system is presented in this chapter. Reliability of a system with the recovery scheme is calculated and compared with reliability of systems without recovery. Implementation of the recovery scheme in our case study is described, and the hardware overhead introduced by the recovery scheme is shown and discussed in this chapter.

Chapter 5 presents a permanent fault repair technique for FPGA-based systems with limited redundant area. Three design alternatives for permanent fault repair with graceful degradation of system availability are presented. A metric that is used to evaluate the designs is defined. Analytic results are shown to compare these design candidates depending on application requirements. Hardware implementations of the three designs in our case study are described. Synthesis results are compared and discussed.

Chapter 6 describes a fault tolerant communication scheme between repair controllers in the Dual-FPGA architecture. The repair process in the architecture is explained, and the roles of the repair controllers are described. Error detection, fault diagnosis, and repair strategies for faulty wires in the controller communication channel are described in this chapter.

Chapter 7 concludes the dissertation.

Chapter 8 lists the references.

Chapter 2
Background and Previous Work

Adaptive Computing Systems (ACS) augment the traditional Von Neumann processor-memory computation model by a fabric of reconfigurable hardware. A typical example is the NAPA processor [Rupp 98], which comprises a general-purpose processor, a reconfigurable co-processor, memory, and I/O devices. Because of their flexibility and massive parallelism in the reconfigurable hardware, ACS designs are used in many different applications, such as image processing [Moorman 99], data compression [Schmalz 98], and signal processing [Fiore 98]. Moreover, ACS architectures are also used in space applications, for their high computation throughput with lower volume, mass, power, and expense compared with conventional spacecraft computers [Donohoe 99] [Brodrick 01]. Because of the dependability need in these applications, different dependable ACS architectures and various fault tolerance techniques for reconfigurable hardware have been developed. In this chapter, we give a brief review on techniques developed for dependable ACS.

2.1 Dependable Adaptive Computing System (ACS) Architectures

A general model for adaptive fault-tolerant systems can be found in [Hiltunen 94]. Several dependable ACS architectures are presented in the literature [Kwiat 97] [Psomoulis 99] [Saxena 00] [Emmert 00] [Mitra 00a]. Reconfigurable hardware plays different roles in different architectures. [Kwait 97] and [Psomoulis 99] use processors for task execution, and FPGAs are used to support fault tolerance for processors. [Saxena 00] and [Emmert 00] use FPGAs for part or all of computation tasks, and their reconfiguration tasks are supported by microprocessors. [Mitra 00a] presents a dual-FPGA architecture, which uses only FPGA to provide fault tolerance. These architectures are described in the following paragraphs.

In [Kwait 97], redundant processors are responsible for task execution, and FPGAs are responsible for voting and error detection for processors. In [Psomoulis 99], computation tasks are partitioned and executed by two processors. FPGAs are used as

processor interface. When one of the processors fails and is taken away, the FPGA-based interface is reconfigured to support the single-processor configuration.

In [Saxena 00], computation tasks are executed by a multi-threaded processor and a reconfigurable coprocessor. Fault tolerance in the processor is achieved through voting among redundant computation threads. In the coprocessor, different fault tolerance techniques can be implemented for different applications. Permanent damage to the processor need to be repaired by chip replacement, and damage to the reconfigurable coprocessor can be repaired through reconfiguration. In [Emmert 00], an FPGA is used to perform computation tasks, and on-line testing is performed in FPGA to diagnose faulty elements. An embedded microprocessor with storage medium is used to control the test, diagnosis, and reconfiguration.

All of the architectures mentioned above need microprocessors for task execution and FPGA reconfiguration. However, permanent damage in microprocessors needs to be repaired by chip replacement, which is quite expensive. [Mitra 00a] presented a Dual-FPGA architecture where no processors are used. As shown in Fig. 2-1, two FPGAs are used for task execution, and they monitor each other. Controllers are embedded in each of the FPGAs to perform monitoring, fault diagnosis, and reconfiguration tasks. When a permanent fault occurs to one of the FPGAs, the controller in the other FPGA detects the fault and evokes fault diagnosis for reconfiguration. Hence, no chip replacement is needed for permanent fault repair.

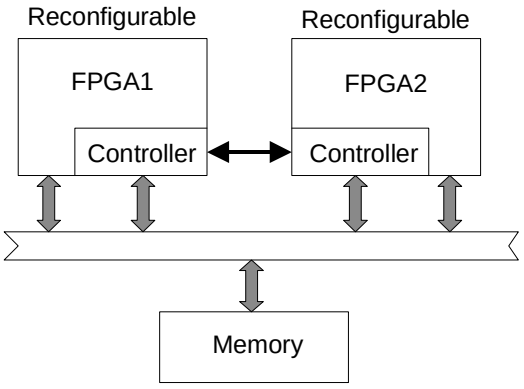


Figure 2-1: Dual-FPGA ACS Architecture.

2.2 Fault Tolerance Techniques for FPGAs

Many techniques have been developed to provide fault tolerance in FPGAs. Figure 2-2 illustrates a general flow of error recovery for FPGA-based systems. First, concurrent error detection (CED) mechanisms detect an error. If an error happens for the first time, it is treated as a transient error; if the error persists, it is treated as a permanent error. When a transient error occurs, the system recovers from corrupt states and resumes normal operation. When a permanent fault occurs, fault diagnosis is initiated to determine the location of the damaged resource, and a suitable configuration is chosen according to the available area. Then computation is resumed.

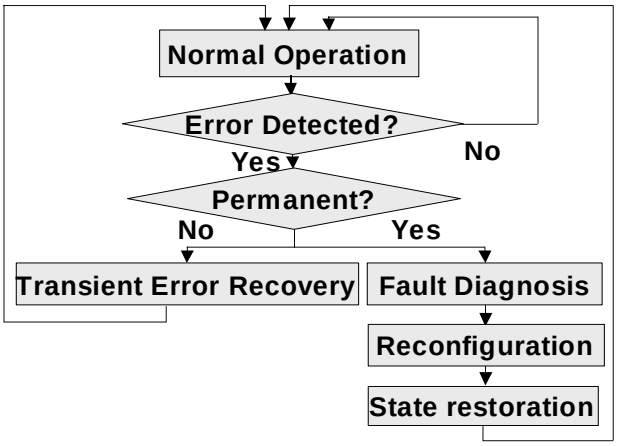


Figure 2-2: Error recovery in FPGAs.

As shown in Fig. 2-2, error recovery introduces timing overhead. For real-time computing systems, the timing overhead needs to be minimized. We define the timing overhead as follows:

$Overhead_{CED}$: The timing overhead introduced by concurrent error detection.

$Overhead_{T}$: The timing overhead introduced by transient error recovery.

$Overhead_{P}$: The timing overhead introduced by permanent fault repair. It includes the delay of fault diagnosis, reconfiguration, and state restoration.

In this section, we describe different fault tolerance techniques that are used in FPGA error recovery.

2.2.1 Concurrent Error Detection (CED)

Various concurrent error detection techniques for general computing systems can be found in literature [Siewiorek 00]. Examples include parity check [Tang 69], duplication [Bartlett 78], design diversity [Mitra 00b], and many application specific schemes, such as arithmetic codes [Sparmann 96] or inverse comparison [Huang 00]. Unlike non-reconfigurable computing systems, an FPGA-based system may have errors in configuration circuitry, such as SRAM cells that store configuration data. The errors in configuration circuitry can appear as a permanent fault since the design configuration is changed. Hence, these errors can also be detected by conventional CED schemes. In addition to conventional CED schemes, different techniques have been developed to detect errors in configuration circuitry. One approach is to build error detection circuitry within SRAM cells [Lala 99]. Another approach is to read configuration data back and to compare it with reference configuration data, which is stored in other places [Conde 99] [Li 99].

2.2.2 Transient Error Recovery

When a transient error occurs in a computing system, data recovery is needed to continue system operation. Depending on the effect of errors, transient faults in reconfigurable hardware can be classified into two complementary categories: *system transients* and *configuration transients* [Huang 01a]. *System transients* cause errors in user application data and do not alter the configuration of reconfigurable hardware. *Configuration transients* cause errors in configuration memory and alter the configuration of the reconfigurable hardware.

Possible recovery schemes for system transients include *rollback* [Chandy 72] [Hunt 87] [Huang 00b] and *roll-forward* recovery [Adams 89] [Pradhan 92] [Long 90] [Xu 96] [Nett 97]. These recovery schemes include both software-assisted recovery such as recovery subroutine in general-purpose processors, and in hardware-assisted recovery. Rollback recovery restores system state through re-computation. In rollback recovery, the system state is backed up to some point in its processing, which is called a *checkpoint*. When an error occurs and is detected during operation, the system will restore its state back to the previous checkpoint and re-compute. However, re-computation has time overhead. Roll-forward recovery obtains correct data through redundancy. When an error

occurs and is detected during operation, the system will recover the corrupt data by copying correct data from fault-free redundant modules to the faulty module. Hence, re-computation delay is avoided. Roll-forward recovery introduces smaller $overhead_T$ than rollback recovery.

Recovery schemes for configuration transients include built-in error correcting circuitry [Wang 99] and using configuration writeback to correct configuration data [Alfke 98] [Carmichael 99]. These recovery schemes can be performed at the same time as computation processes are executed in the FPGA. Therefore, user functions do not need to stop during recovery, and very small $overhead_T$ is introduced. For some types of FPGAs, in which part of configuration memory cells can also be used as user memory, a memory coherence issue may be raised during recovery. The reason is that under some fault scenarios, both user and recovery process may access a memory cell at the same time. A technique is presented in [Huang 01a] to solve the memory coherence problem.

2.2.3 Permanent Fault Repair

The basic idea of permanent fault repair in FPGAs is to remove permanent faults from designs through reconfiguration. A new configuration is generated based upon the information of permanent fault location, so that permanent damaged elements are avoided.

There are many fault diagnosis schemes for FPGAs to support permanent fault repair. Computation resources of FPGAs include logic cells and interconnect. Different schemes are developed for fault diagnosis of logic cells [Lu 99a] and interconnects [Lombardi 96] [Culbertson 97] [Renovell 97]. These schemes reconfigure FPGAs into different configurations for testing, and test patterns are applied externally. Some other schemes reconfigure FPGAs into Built-In Self Test circuitry; therefore, no external test stimuli are needed [Lu 99b] [Hamilton 99] [Abramovici 01]. Most of the diagnosis schemes are configuration-independent, that is, they test all elements in FPGAs whether or not an element is used by current configuration. Configuration-independent diagnostic tests have drawbacks in that test development and testing itself are quite time consuming [Culbertson 97]. For FPGA repair purpose, we only need to locate faulty elements that are currently used, instead of locating all faulty elements. Hence, several configuration-dependent diagnostic schemes are developed to reduce testing time [Das 99] [Huang 01c].

As described in Sec. 1.1, two major approaches are used to provide new configurations for permanent fault repair. One is to generate a new configuration after permanent faults are detected. The other is to generate pre-compiled alternative configurations. For new configuration generation after fault detection, different CAD tools and algorithms are developed for fast rerouting and re-mapping. In some algorithms, spare computation elements are allocated before downloading designs [Hanchek 98] [Dutt 99] [Mahapatra 99]. In other algorithms, they use the inherent redundancy in FPGA designs as spares for permanent fault repair [Mathur 96] [Emmert 97], hence, no spare allocation is needed before initial routing and mapping. However, without spare pre-allocation, a design may not always be routable under presence of faults [Roy 95].

To reduce time overhead of producing new configurations, pre-compiled configurations are also used for permanent fault repair schemes in [Lach 00] and [Huang 01b] [Emmert 00]. These schemes partition an FPGA into small blocks, and generate alternative configurations where each of the configurations does not utilize one or multiple blocks of the FPGA. Hence, depending on the location of faulty blocks, a new configuration can be selected for fault repair. To reduce overhead of memory that store configurations, compression is used in [Huang 01b] to decrease the size of configuration files.

Chapter 3
A Fault Tolerant Real-Time System on Reconfigurable Hardware

To utilize reconfigurable hardware in mission-critical real-time applications, the first thing we need to know is whether a real-time computing system can meet all application requirements when it is constructed on reconfigurable hardware. In this chapter we describe an implementation of a robot controller, which executes a control algorithm that is popularly used in industry, on an ACS platform. Details can be found in Appendix A.

3.1 Case Study: A Robot Control System

We choose a robot control system as our case study, since it is a real-time system and is frequently used in life-critical applications such as handlers in nuclear power plants. Figure 3-1 shows a block diagram of a general robot system [Craig 89]. It consists of a robot unit and a control unit. The robot unit represents the mechanical part, and the control unit represents the electronic part. The interface between the robot unit and the control unit includes sensors and A/D, D/A converters. The sensors sample the current position of the mechanical part and transform it to an analog signal (voltage or current). The A/D converter converts the signal and sends it to the control unit. The desired trajectory and other control parameters are supplied by the user to the control unit. The control parameters describe the characteristics of the robot unit. The control unit collects control parameters, desired position, and current position of robot, and calculates the needed force. The calculated value is converted to an analog signal, and that signal drives the motor to move the robot.

Our focus is on the control unit, which is an electronic controller that executes a control algorithm to control the robot. Since the performance of the controller affects the performance and responsiveness of the whole robot system, the controller needs to be designed with proper performance to meet application requirements. In addition, proper

precision is needed to meet the resolution and accuracy requirements. These design issues will be described in later sections.

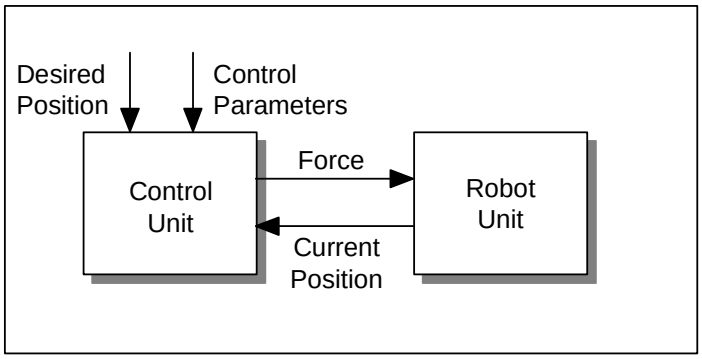


Figure 3-1: Block diagram of a general robot system.

3.2 Previous Work on Reliable Robot Systems

Much research has been done seeking ways of providing an effective fault tolerant robot system. For example, different control schemes are presented in [Ting 94][Yang 98]. Redundancy and voting in joint and actuator levels are suggested in [Tosunoglu 93] [Toye 90] [Thayer 76]. [Hamilton 92] and [Hamilton 94] use multiple processors for control purpose. These processors can be either used for parallel control that can speed up computation of control algorithms, or used as redundant processors for fault tolerance. [Visinsky 95] presents a fault tolerant framework for robots. In the framework, the controller contains special control algorithms that are used when faults are present. [Toye 94] presents different voting methods for robot applications. The voting schemes includes a dynamic democratic voting scheme which takes error history into consideration, and another scheme which uses force compensation to vote among actuators. Metrics to evaluate risk, cost, fault tolerance, performance, and benefit in robot systems have also been investigated [Hamilton 96a] [Hamilton 96b].

With the exception of [Hamilton 92] and [Hamilton 94], most of the cited research work concentrates in either the control or the mechanical field. Not much work has been done in fault tolerance of the electronic hardware that instruments control algorithms other than replicating processors. In this chapter, we use FPGAs that can be used in ACS framework to construct a fault tolerant robot controller and show that it can obtain both high dependability and performance.

3.3 Fault Tolerance Techniques

Fault tolerance techniques are needed to improve the dependability of the control system. To determine suitable fault tolerance techniques for our ACS robot application, two techniques were implemented and compared. One is *multi-threading*, and the other is *combinational redundancy*.

The concept of using multi-threading has been described in [Thorton 64] and [Smith 78]. The idea of using multi-threading for fault tolerance was described in [Saxena 98]. This is illustrated in Fig. 3-2. Figure 3-2 (a) shows an algorithm executed in multiple stages in limited hardware. Some resources are under-utilized (idle) due to data dependencies and memory latency. By scheduling multiple independent threads most of the idle resources can be reclaimed. To obtain fault tolerance, multiple copies of the same algorithm are executed as multiple threads in the hardware. The final output is obtained by voting the results from different threads. As shown in Fig. 3-2 (b), the replicated instructions of the redundant thread can be inserted into the idle stages to minimize timing overhead. In a fault-free scenario, identical outputs are produced from all threads. When faults are present, the redundant threads provide a means for fault detection and fault masking.

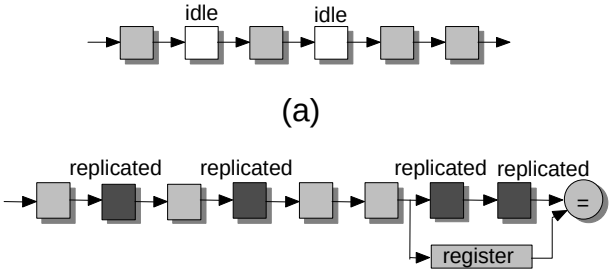


Figure 3-2: Multi-threading:(a) an algorithm in multiple stages, and (b) multi-threading.

In combinational redundancy, the robot controller is implemented in combinational hardware, and the hardware is replicated into multiple copies. The input is connected to all of the modules. In case of two copies (duplex), a comparator is used to detect errors. For three or more copies, a voter is used to determine the final output among the outputs from the modules. The replicated copies provide fault detection and possibly fault masking capabilities.

There exist many possible voting schemes. We implement two voters in our design and compare their behavior. One is the majority voter, which is widely used, and the other is the median voter, which is reported to have lowest error probability in [Bass 97]. The majority voter selects the bit-wise output agreed by the majority among the modules. The median voter selects the middle value among the outputs from different modules. Comparators are added to detect disagreements between the voted results and each of the three outputs. This technique has been used in previous literature such as hybrid redundancy [Siewiorek 73] to remove a faulty module from the system. Comparators are also used in the word voter [Mitra 00c] for disagreement detection.

3.4 Emulation of the Control Algorithm

We emulated the second-order linear control algorithm in ACS hardware, using the Quickturn™ System Realizer [Quickturn 01] and WILDFORCE® board [Annapolis 01]. For the purposes of comparison, the algorithm was also implemented in C programs running on general-purpose processors. To investigate the area overhead and the performance degradation of designs with fault tolerant features, different fault tolerance techniques are implemented in the control system for a comparative evaluation. In addition, the controllers are designed using different bit-widths for precision, and an optimum bit-width is chosen.

Section 3.4.1 describes control systems in ACS hardware. Section 3.4.2 explains the implementation of the control algorithm in general-purpose processors and shows performance data. Section 3.4.3 discusses our results.

3.4.1 Robot Control System in ACS

The controller was initially emulated in the Quickturn's System Realizer to verify its functionality. In the system realizer, Xilinx XLA4013 FPGAs are used. To get more accurate performance and area data, it was implemented on the WILDFORCE board, which is used in adaptive computing applications [Walters 98]. Xilinx 4036LCA FPGAs are used in the board. A detailed description of the emulation platforms can be found in the appendix A.

In the control unit design, we use two architectures to implement fault tolerance techniques. Operation of the control algorithm consists of addition and multiplications.

To implement multi-threading, the computation is partitioned into multiple stages, and the arithmetic computation resources, adders and multipliers, are reused. The computation is replicated as different threads. The threads share the same hardware resources. To implement combinational redundancy, the control unit is designed as a combinational circuit and replicated.

To select a suitable precision for our control system, several controllers were designed using different precisions. The robot characteristics, such as the response time, the steady-state error, and the time to stabilization were used as evaluation criteria. Among all controllers that met the application requirements, the lowest-precision design was selected to minimize the hardware overhead and to optimize the performance.

Table 3-1: Simulated square-wave response for systems with various precisions.

Number of bits	32	24	16	12
Steady state error(%)	0.006	0.1	2	---
Time to stabilization (<0.1%) (cycles)	135	133	145	
Response time (cycles)	1	1	1	

Table 3-1 lists the simulated response data of systems with different precision. We use *steady-state error*, *time to stabilization*, and *response time* to evaluate the controller. *Steady-state error* is the difference between the desired position and the robot position when the robot finally rests at a stable position after stimulation. *Time to stabilization* is the time needed for a robot to reach its final stable position. *Response time* is the time needed to move a robot 90% to its desired position. The precision used was 32, 24, 16, and 12 bits. With the given test trajectory, the systems work well in all precisions except 12 bits. In the 12-bit system, arithmetic overflow occurs resulting in overshoot of the output waveform. As shown in Table 3-1, the response time and the time to stabilization of systems with 32, 24, and 16-bit numbers are almost the same. The major difference among the systems is accuracy: a higher-precision system has less steady-state error. Therefore, in terms of the response of the system, as long as the requested data range does not cause arithmetic overflow in the system and the steady state error is tolerable, a low-precision system is sufficient for the application while taking less area and shorter execution time than a higher-precision design. From the simulation results, the 16-bit system was the implemented choice on the WILDFORCE board.

Table 3-2 lists the designs emulated on the Quickturn's System Realizer. The designs use multi-threading as the fault tolerance technique. For the three-thread design, both 32-bit and 16-bit numbers are implemented. These designs are verified to have correct functionality through emulation. Tables 3-3 and 3-4 show the emulation results on the WILDFORCE board. The area data are obtained from report files of a Xilinx placement-and-route tool. The frequencies are obtained from emulation. In the tables, the combinational hardware area is represented by the number of Look Up Tables (LUTs), and the sequential area is represented by registers. The results are discussed in Sec. 3.4.3.

Table 3-2: Emulation of multi-threaded designs on Quickturn's System Realizer.

Threads	1	2	3	
Number of bits	32	32	32	16
Number of FPGAs	29	29	103[1]	20
Stages	7	9	11	11

Table 3-3: Emulation results on WILDFORCE board – combinational redundancy.

# of Modules	1	2	3	3
Voter Type	---	---	Median	Majority
LUTs (total 3888)	404	819	1284	1257
Registers (total 2592)	115	230	345	349
Frequency (MHz)	24	26	22	24
Latency (μs)	0.042	0.038	0.045	0.042

Table 3-4: Emulation results on WILDFORCE board – multi-threading.

# of Threads	1	2	3	3
Voter Type	--	---	Median	Majority
LUTs (total 3888)	409	814	1248	1251
Registers (total 2592)	362	636	945	945
# of Stages	7	8	11	11
Frequency (MHz)	50	50	50	50
Latency (μs)	0.14	0.16	0.22	0.22

[1]This design was emulated in Quickturn M3000 System Realizer, which is larger than M250. Other designs shown on the table were emulated in Quickturn M250 System Realizer.

3.4.2 Control Algorithm on General-Purpose Processors

We also implemented the control algorithm in C programs and ran them on Ultra Sparc II and Pentium II Xeon processors. Two data types are used: single precision floating-point numbers (32 bits), and fixed-point numbers (32 bits). Redundancy is achieved by software redundancy, that is, duplicating the code inside the program. Table 3-5 shows the latencies of the C programs on general-purpose processors. The latencies are roughly proportional to the number of copies. These results are compared with the performance of the ACS implementation in the next section.

Table 3-5: Performance of the control algorithm on general-purpose processors.

Latency (μs)				
# of Copies	Ultra SPARC II (300MHz)		Pentium II Xeon (450MHz)	
	Float	Fixed	Float	Fixed
1	0.186	0.044	3.866	0.033
2	0.288	0.078	7.741	0.074
3	0.569	0.122	11.689	0.105

3.4.3 Discussion

In this section, the results from the emulated designs on the WILDFORCE board are compared with the results obtained from general-purpose processors.

As shown in Table 3-3, the designs with combinational redundancy have latencies ranging from 0.038μs to 0.045μs. As shown in Table 3-4, all designs with multi-threading are running at the maximum frequency of the board, which is 50MHz. Table 3-6 lists the expected latency of multi-threaded design from synthesis results of Synplify®, and the latencies are much higher than those obtained by emulation. Higher emulation performance may be obtained in multi-threaded designs if the maximum board frequency can be improved. The latencies of multi-threaded designs from emulation range from 0.14μs to 0.22μs, and their expected latencies range from 0.075μs to 0.154μs. The designs with combinational redundancy have shorter latencies than the multi-threaded designs because they are fully combinational.

Table 3-6: Expected latency on WILDFORCE board – multi-threading.

# of Threads	1	2	3	3
Voter Type	--	---	Median	Majority
# of Stages	7	8	11	11
Frequency (MHz)	92.6	92.6	71.4	71.4
Latency (μs)	0.075	0.086	0.154	0.154

As the area results show, the area overhead is roughly proportional to the number of modules or threads. Multi-threaded designs require more registers because hardware is reused and intermediate results need to be stored. For combinational redundancy, area results are proportional to number of copies. However, in the designs with multi-threading, even when resources are reused, the area overhead is not significantly reduced. The reason is that in the implemented algorithm, the area of the reused hardware resources is small compared to the additional registers and the interconnection resources for those registers, which are proportional to the number of threads. In systems that use large and complicated hardware to process data, multi-threading may take advantage of reusing resources and may reduce hardware overhead. In our case, the designs with combinational redundancy have better performance and less area overhead than those with multi-threading. However, for a more complicated control algorithm, combinational redundancy may not always be better than multi-threading. The reason is that a fully combinational circuit for a complicated algorithm can take much area and may not fit in an FPGA. To fit a large algorithm in an FPGA, resource reuse might be needed. In this case, multi-threading can be a better choice to provide redundancy for fault tolerance.

Figure 3-3 shows the performance for the one-copy (simplex) and three-copy (TMR or software redundancy) designs of the control algorithm in both general-purpose processors and reconfigurable hardware. The TMR latency for general-purpose processors are assumed to be the same as that of a simplex design because voting latency is generally small compared with total latency. The Pentium II Xeon processor is in 0.25μm technology, and both the Ultra Sparc II processor and Xilinx FPGA chips used in the WILDFORCE board are based on 0.35μm technology. The simplex results show that even when older generation FPGAs were used, the performance obtained in

reconfigurable hardware is comparable to that of the later generation general-purpose microprocessors. For the TMR design in WILDFORCE board, the performance is not degraded because it takes advantage of parallelism in hardware. We could also use multiple general-purpose processors when implementing TMR to achieve similar performance as simplex designs. However, the cost of multiple processors would be fairly high comparing to a single FPGA. In our case, if we want to implement fault tolerance techniques into designs without much additional cost, a TMR design in a FPGA would be a better choice than a design with software redundancy in microprocessors. For a more complicated control algorithm, a FPGA can still be an appealing choice because parallism can be used to achieve high performance/small latency for a FPGA-based TMR design.

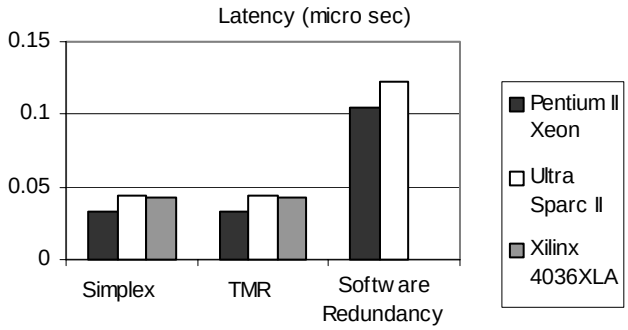


Figure 3-3: Controller performance on general-purpose processors and ACS.

3.5 Comparison of Fault Tolerance Techniques

Comparison is needed to determine the suitable fault tolerance technique for the implemented system in reconfigurable hardware. We consider dependability, area overhead, and performance degradation as the criteria to evaluate our implementation.

In order to evaluate dependability, faults were injected into the control unit of the system by modifying configuration files, and fault coverage was measured. We injected stuck-at-0 and stuck-at-1 faults at the output of LUTs. In each design of the control system, all possible single stuck-at faults are examined. To compare the effectiveness of the voters in the presence of multiple faults, double stuck-at faults are injected in designs with majority voters and median voters. For each examined design, five thousand stuck-at fault pairs are chosen randomly. Output vectors of fault-injected designs are compared

with those of fault-free designs. An output is considered *corrupted* if its value is different from the output of fault-free systems. Results from comparators which compare voted results and each of the three outputs are used to indicate error. Tables 3-7 and 3-8 list the results of single stuck-at fault injection experiments. Table 3-9 lists the results of double stuck-at fault injection experiments. In the tables, we define *corruption without detection* as the case when an output is corrupted but no detection signal is generated from comparators; *corruption with detection* is defined as the case when an output is corrupted and comparators detect the disagreement; *detection without corruption* means that comparators detect a disagreement, but the output is the same as the output from a fault-free design; and *no detection or corruption* is defined as the case when an output is the same as the fault-free output, and comparators do not detect any disagreement.

Table 3-7: Single stuck-at fault injection in designs with multi-threading.

# of Threads	1	2	3	3
Voter Type	---	---	Median	Majority
Injected Faults	936	1843	2803	2830
Corruption w/o Detection	806	18	39	7
Corruption w/ Detection	0	788	121	46
Detection w/o Corruption	0	785	2284	2404
No Detection or Corruption	130	252	359	373

Table 3-8: Single stuck-at fault injection in designs with combinational redundancy.

# of Modules	1	2	3	3
Voter Type	---	---	Median	Majority
Injected Faults	952	1932	2932	2812
Corruption w/o Detection	825	1	4	1
Corruption w/ Detection	0	835	102	32
Detection w/o Corruption	0	57	2442	2428
No Detection or Corruption	127	239	384	351

Tables 3-7 and 3-8 show that in the single-threaded and single-modular designs, which do not have any fault tolerance or error-detecting scheme, almost all of the injected faults result in corruption. For the designs with fault tolerance techniques, the undetected corruption caused by single faults only occur when the fault is injected in the comparator

(error detector), I/O logic, or the common logic part, such as the scheduler. Therefore the undetected corruption is less likely to appear.

Table 3-9: Double stuck-at fault injection in designs with different voters (5000 fault pairs are randomly chosen).

Fault Tolerance Technique	Multi-Threading (3 threads)		Combinational redundancy (3 copies)	
Voter Type	Median	Majority	Median	Majority
Corruption w/o Detection	24	7	0	0
Corruption w/ Detection	1754	1437	1656	1796
Detection w/o Corruption	3139	3464	3250	3125
No Detection or Corruption	83	92	94	79

The two-threaded and two-modular designs only have error detecting but not fault masking mechanism. In these designs, the number of detections with corruption is roughly the same as the number of detections without corruption. The reason is because the output of the designs is from one of the two identical threads or modules, and the probability of fault occurrence in the output thread or module is roughly 50%. The reliability can be improved if diversity is used between modules or threads [Mitra 99].

From the single stuck-at fault results, the three-threaded and three-modular designs, which have fault-masking mechanism, have much less corruption than other designs. The results of both single and double stuck-at faults show that the designs with a majority voter produce smaller numbers of undetected corruption than the designs with a median voter. It means that the median voter is more vulnerable to stuck-at faults than the majority voter, and is less likely to detect corruption. The reason is that a median voter requires subtracters to select middle value among the three. In FPGAs, special carry chain is designed to speed up addition/subtraction. A stuck-at fault in a median voter may affect several output bits at the same time due to the error propagation through the carry chain. On the other hand, in a majority voter, voting is done bit-by-bit. Therefore, a stuck-at fault will affect, in the worst case, one bit of the outputs of the majority voter.

The numbers for corruptions without detection in designs with combinational redundancy are much smaller than those of designs with multi-threading. The reason is that, compared with combinational redundancy, multi-threaded designs have more

hardware that is commonly shared by different threads, such as schedulers and control logic. Therefore, faults are more likely to affect resources that are shared by several threads, and consequently, an error detection mechanism will not detect them. On the contrary, the designs with combinational redundancy replicate the whole module, hence a single fault can affect only one module, and corruption occurs only if the fault is in the voter.

3.6 Summary

We have demonstrated that an ACS is a good platform for implementing robot control algorithms. As a case study, a robots control algorithm has been implemented on reconfigurable hardware. Results obtained from hardware emulation on the WILDFORCE platform have demonstrated that the performance of running the algorithm on reconfigurable hardware is comparable with that on a general-purpose processor. The ability of implementing systems with different precision to customize the design has demonstrated the advantage of an adaptive computing system. In conjunction with different voting schemes, fault tolerance techniques have been used in implementing the algorithm. Fault injection experiments verified fault tolerance capabilities of the robot controller on reconfigurable hardware.

Chapter 4
A Roll-Forward Recovery Scheme for TMR Systems in Real-Time Applications

TMR systems are extensively used in real-time applications because of their fault masking ability [Siewiorek 00]. However, they are only effective for short mission times. Different methods have been presented to recover TMR from permanent faults [Siewiorek 73] and transient faults [Chande 89] so that they can be used in longer missions. However, transient error recovery in TMR is accomplished by re-computation [Chande 89]. Re-computation may not be allowed in real-time applications because of strict deadlines. Roll-forward recovery schemes, which obtain correct states from fault-free redundant modules, avoid re-computation delays. These schemes were used in duplex systems with spares [Pradhan 92] [Xu 96] and quadraplex systems [Adams 89]. However, duplex systems with spares do not provide sufficient redundancy, so that roll-forward in these systems still has potential to cause re-computation. The quadraplex-based roll-forward scheme keeps the fault-free modules running while a faulty module is restored [Adams 89]. When a module is faulty, three modules are left in a quadraplex system for execution, so error masking is still provided. However, this scheme is not suitable for TMR systems since only two modules are left for execution under presence of a single fault. To meet real-time application requirements, we present a roll-forward recovery scheme for TMR systems to lengthen their mission times. In our scheme, no re-computation is needed for recovery; therefore, it is very suitable for real-time applications.

4.1 Roll-Forward Recovery Scheme

4.1.1 Problem Definition

Consider a computing module that has combinational circuitry and storage elements. The computing process in the computing module is a real-time task and each transaction has to be finished before its deadline. TMR is used to provide error masking for the module. During the module's operation, transient faults may occur in the

computing module. An error may be latched into storage elements and the system may end up being in an incorrect state producing erroneous outputs in successive cycles. Our purpose is to correct the erroneous data residing in storage elements through recovery, so that the correct operation of a module is not stopped by a transient fault. In addition, the performance impact caused by the recovery process has to be minimized, so that transactions' deadlines could still be met.

Our scheme is suitable for recovery from single transient faults in computing modules. The scheme can be used in conjunction with permanent fault repair schemes such as self-purging redundancy [Losq 76] to repair systems from permanent faults.

4.1.2 TMR Schemes

Our roll-forward recovery scheme is designed for TMR systems. Two forms of TMR are discussed in this chapter. One is *hardware redundancy*, and the other is *multi-threading* [Saxena 98] [Yu 00]. In hardware redundancy, the hardware is replicated with two copies. A voter is used to determine the final output among the three modules, and voted outputs are compared with module outputs. When there is a disagreement between any module outputs and the voted outputs, an error indication is generated and the faulty module is identified. Disagreement detectors have been used in previous schemes such as hybrid redundancy [Siewiorek 73] to remove a faulty module from the system.

Multi-threading is illustrated in Fig. 4-1. Figure 4-1 (a) shows an algorithm executed in multiple stages reusing hardware. An example of hardware reuse is arithmetic logic units (ALUs) in microprocessors. Different arithmetic operations may use the same ALU at different times for computation. For algorithms executed in multiple stages, some resources may be under-utilized (idle) because of data dependencies and memory latency. To obtain fault tolerance by multi-threading, three copies of the same algorithm are executed as different threads in the hardware. As shown in Fig. 4-1 (b), the replicated instructions of redundant threads use hardware resources, which are originally, idle. The final output is obtained by voting among the results from the three threads. An error detection mechanism is added to the voters. Outputs of each thread are compared with the voted output, and if they are different, error signals will be generated so that the faulty thread is identified.

Multi-threading is not suitable for systems in which no resource is idle at anytime. In these systems, we cannot take advantage of idle resources for redundant computation. Therefore, another form of TMR needs to be used.

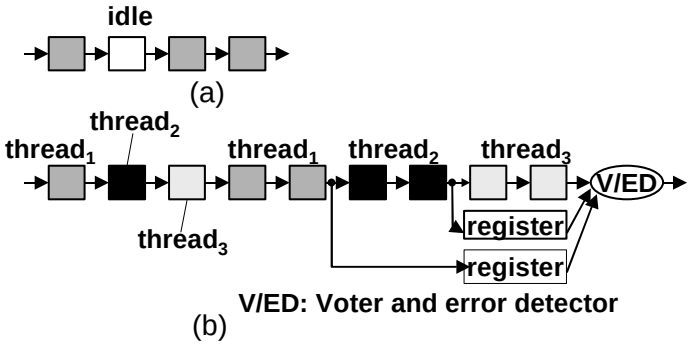


Figure 4-1: Multi-threading: (a) one-threaded computation, and (b) three threads for fault tolerance.

Hardware redundancy has small time overhead. In addition to transient faults, it is capable of tolerating permanent faults. However, it is expensive in terms of hardware overhead. Redundancy obtained by multi-threading is as effective as hardware redundancy for tolerating transient faults, where multi-threading might have smaller hardware overhead since hardware is reused. However, computation latency in multi-threading may be increased. The amount of time overhead in multi-threading depends on the resource usage and data dependency, and it varies among applications. In a computation that has high hardware utilization, a recovery scheme with multi-threading is not necessarily suitable to replace rollback recovery, since a three-threaded design may cause delay longer than simple re-computation after each checkpoint at which there is an error. Since our recovery method is not restricted to a particular redundancy technique, throughout this chapter we will use the term *copy* to describe both original and replicated modules or threads. For hardware redundancy, a copy means a replicated hardware *module*, and for multi-threading, a copy means a replicated computation *thread*. In the following discussion, we use the notation *HR* for hardware redundancy, and *MT* for multi-threading.

4.1.3 Recovery Scheme

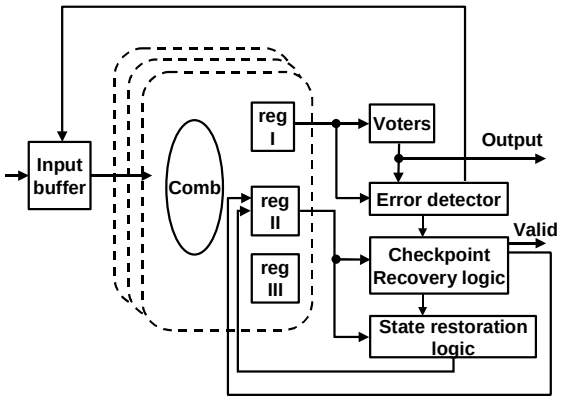


Figure 4-2: The block diagram of a TMR system with the roll-forward recovery scheme.

To determine if there is need for recovery, three computation copies are compared and the results are examined at pre-scheduled computation points. We define these computation points as *checkpoints*. At checkpoints, if comparators detect disagreement among three copies, a recovery process will be initiated. Inputs will be buffered and be processed when the recovery process finishes.

Figure 4-2 shows a block diagram of a TMR system with recovery mechanism. The system consists of three hardware copies or three computation threads, an input buffer, a voter for the output, an error detector, a checkpoint logic/recovery control block, and a state restoration block. In each copy, registers can be divided into three different sets. Register I is the set of registers that contain output data. As shown in the diagram, their outputs are connected to the voter. Register II is the set that contains data that are still needed for computation after checkpoints. Examples for Register II are program counters in microcontrollers and microprocessors. Therefore, data stored in Register II need to be recovered at a checkpoint when a fault happens. As shown in the diagram, these registers are connected to the state restoration logic block and the checkpoint/recovery block. Register III is the set of registers that store intermediate results that are not needed after the checkpoints. They do not need to be recovered when faults are present. Examples for Register III are registers that store intermediate results for subfunction calls. After the subfunction finishes and returns to a main function, these data are not needed anymore. In some applications, Register I and Register II may have a

portion that overlaps. That is, some registers that contribute to the output also need to be recovered during recovery.

There are two major reasons to initiate recovery processes at preset checkpoints instead of as soon as a fault happens. One reason is that with pre-scheduled checkpoints and recovery processes, a fault can be recovered after real-time transactions are finished, hence, transactions will not be interrupted or delayed by the recovery process. The other reason is in some applications, checkpoints can be inserted in a way that the amount of registers in which data need to be recovered is minimized. For example, during a computation task there can be many intermediate values that only contribute to the current task and are not needed for the next task. If we insert checkpoints at completion of tasks, there is no need to recover registers that store these intermediate values since they are not needed anymore. Certainly, if a recovery process is not initiated immediately when a fault happens, extra faults might happen before the recovery process is actually initiated. Because of multiple faults, the system might not be recovered successfully. Longer delay for recovery processes increases the probability of multiple faults. However, a recovery process that can be initiated at any clock cycle is too expensive in terms of both hardware and time overhead. Therefore, recovery processes are scheduled at checkpoints.

During the system's normal operation, all copies execute their function normally. System outputs are obtained by voting. At checkpoints, the checkpoint/recovery controller checks the error detection result. Figure 4-3 shows the logic of checkpoint and recovery. Checkpoints can be inserted by adding an additional checkpoint state in the finite state machine or by counting cycles to pre-defined period. As shown in Fig. 4-3 (a), when the state machine transits to the checkpoint state or when the checkpoint period is reached, checkpoint process is initiated and error indication signals from comparators are examined. The logic that initiates a recovery process is shown in Fig. 4-3 (b). If errors are detected, then a recovery process is initiated. If no errors are detected, the normal process will continue until the next checkpoint. During the recovery process, a signal will be raised to indicate that outputs are invalid, and operation is temporarily stopped. The state restoration block provides correct data to recover states stored in Register II. State restoration will be discussed in detail in the following paragraphs. Incoming data are

buffered and will be processed after recovery is completed. When the restoration process is finished, normal operation will continue.

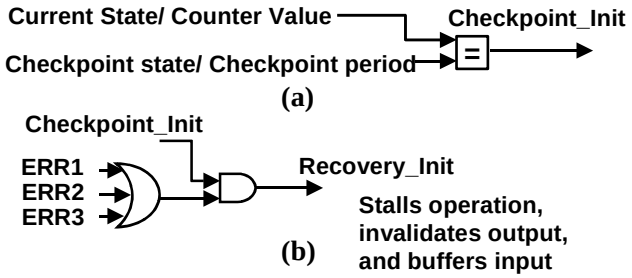


Figure 4-3: Checkpoint and recovery logic used in the recovery scheme: (a) checkpoint logic, and (b) recovery logic.

State restoration is accomplished by copying data from correct copies to the faulty copy by separate controller logic. In [Adams 89], voters are used to obtain correct states for recovery in a quadraplex system. We call the scheme with voter as the *voting scheme (VT)*. A new scheme, the *direct-load scheme (DL)* is presented in this chapter. These schemes are shown in Fig. 4-4 (a) and (b), respectively. The VT scheme obtains correct states by voting among the three copies. The voted results are loaded into all three copies for state restoration. The DL scheme obtains correct states by loading states directly from one of the correct copies into the faulty one. Both schemes are effective for recovering from single faults. The voting scheme can also recover from some multiple faults.

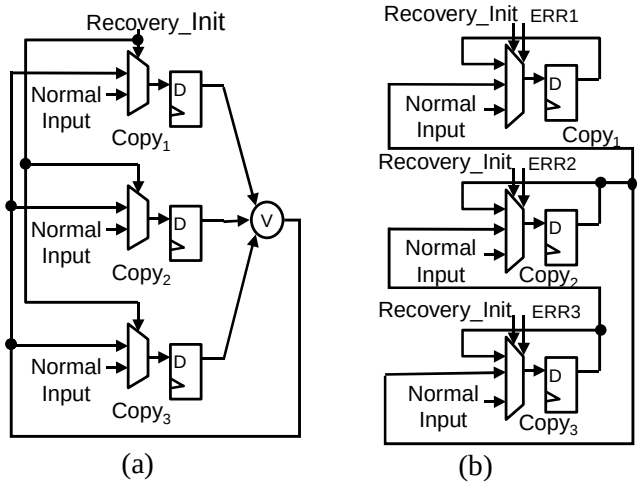


Figure 4-4: The data restoring process: (a) the voting scheme and (b) the direct-load scheme.

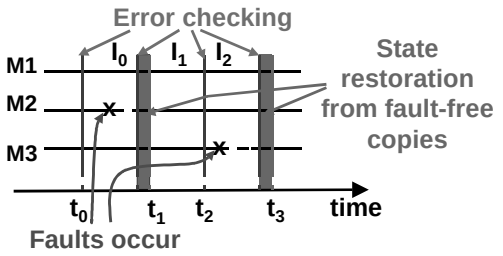


Figure 4-5: A fault scenario for recovery.

Figure 4-5 shows a fault scenario for recovery. As shown in the figure, when a single fault occurs, erroneous outputs are masked by voters. Error detection results are checked at checkpoints by extra hardware as shown in Fig. 4-3. When a faulty copy is identified, the corrupted data is recovered from correct data in fault-free copies. After recovery normal operation is continued again. As shown in the figure, for single copy failures, the fault-free copies provide sufficient information for recovery. Therefore, no rollback is needed. However, failures in common logic such as voters and recovery logic cannot be recovered. To further improve reliability, a fault tolerance mechanism is added to common logic.

To compare overhead among different transient recovery techniques, Table 4-1 lists the timing overhead, $overhead_T$ that was defined in Sec. 2-2, and area overhead of recovery techniques. Rollback recovery does not need redundant modules to provide correct data, but it introduces the most timing overhead among the techniques. To perform rollback recovery, system states need to be backed up at every checkpoint; and upon the presence of a fault, states need to be loaded from the backups and computation needs to be rolled back. For the roll-forward recovery in [Pradhan 92], two modules and spare are needed for the hardware. However, the scheme still needs rollback in some fault scenarios. In addition, checkpointing and state restoration also introduce timing overhead. For the roll-forward recovery in [Adams 89], four modules are used in the system. No rollback is needed in this scheme. State restoration is executed in the background and does not affect normal operations. However, if normal operations write to memory faster than the speed at which states are restored, the state restoration process cannot be converged. Compare with previous schemes, our roll-forward technique eliminates the need of checkpointing and rollback; therefore, timing overhead is significantly reduced. Although we still need to stall normal operation during state restoration for recovery, but

the state restoration process can always converge no matter how the speed of memory writes is. Therefore, our recovery scheme is very suitable for real-time applications.

Table 4-1: Timing and area overhead of transient recovery techniques.

	Rollback [Chandy 72]	Roll-forward in a duplex with spare [Pradhan 92]	Roll-forward in a quadraplex [Adams 89]	**Roll-forward in a TMR**
Timing overhead ($overhead_T$)	State backup + state restoration + re-computation	State backup + state restoration + possible re-computation	State restoration in background, but restoration may not converge	**State restoration**
Area overhead	1 module + CED	2 modules + spare	4 modules	**3 modules**

4.2 Reliability Analysis

Reliability analysis results of the roll-forward recovery scheme are shown in this section. A Markov chain is used to model a TMR system with the recovery scheme. Reliability of the system is calculated and compared with a simplex system, a TMR system, and a TMR-Simplex system. In our analysis, the effects of checkpoint and recovery logic are all taken into consideration. A detailed derivation of the analysis is in the appendix.

We choose the robot controller, described in Sec. 3.1, as our case study to calculate reliability. The robot controller executes the control algorithm, and each computation of the algorithm takes 7 clock cycles. For TMR in hardware redundancy, the whole computation takes 7 clock cycles, which is the same as the simplex design because delay of the voter is small and is not part of the critical path. The three-threaded controller takes 11 clock cycles to complete three computations with the same computation resource as a single-threaded design. In hardware redundancy, checkpoints are set at the completion of each computation. In multi-threading, checkpoints are set at points where all of the three computation threads are completed.The recovery process takes one clock cycle. In a larger design with a larger area and more registers to restore when faults are present, more clock cycles will be needed for recovery. Synthesized data

for reliability analysis were extracted using the Xilinx Virtex FPGA library, since the Virtex FPGA will be used in an adaptive computing system in the ROAR project.

The detailed description of implementation and synthesis is in Sec 4.4. Only the voting scheme for state restoration is used in the calculation. The reason is that the voting scheme has larger area in state restoration logic and larger overhead (which will be shown in Sec. 4.3). We use the larger area to compute the lower bound on reliability for comparison with systems that do not have any recovery mechanism.

Figure 4-6 shows the reliability vs. time, with the time axis normalized to mean time to failure (MTTF) of the simplex design. The result shows that our recovery scheme has much better reliability than Simplex, TMR w/o recovery, and TMR-Simplex. Both hardware redundancy and multi-threading with the recovery scheme have shown this improvement. HR-TMR has reliability slightly better than MT-TMR, since the MT design has larger common logic, such as the thread scheduler, that is shared by the three threads and is not protected by multi-threading. The shared logic can be triplicated to further improve reliability.

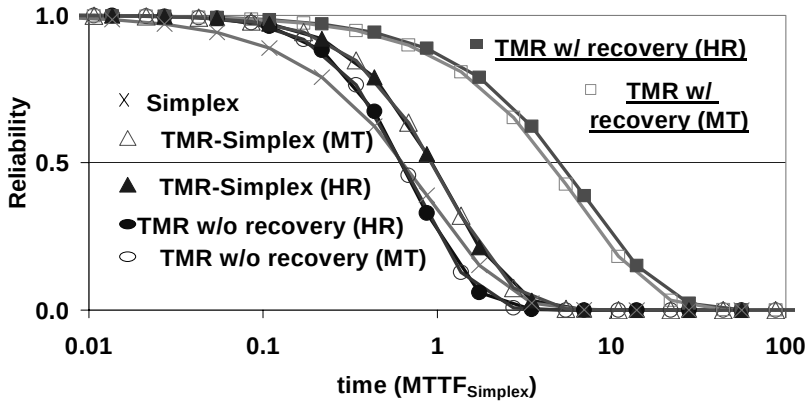


Figure 4-6: Reliability of the robot controller with different fault tolerance techniques.

4.3 Implementation

We implemented the recovery scheme in the robot controller to estimate the overhead of the roll-forward recovery scheme. The designs are synthesized with target libraries including Xilinx Virtex XCV1000 FPGA, and LSI lcbg10p library, using Synplify® and Synopsys® tools, respectively. The results are listed in Table 4-2. The

numbers are normalized, assuming that the area of the simplex design is 1. Sequential logic (registers) and combinational logic (lookup tables -- LUTs) areas are listed.

Table 4-2: Synthesis area of a robot controller with/without recovery mechanism.

	TMR Type	Recovery	State Restore	XCV1000		LSI lcbg10p	
				Regs	LUTs	Non-comb	Comb
Simplex	N/A	No	N/A	402 (1.00)	654 (1.00)	11619 (1.00)	24404 (1.00)
TMR	HR	No	N/A	1206 (3.00)	1979 (3.03)	34857 (3.00)	73399 (3.01)
		Yes	VT	1242 (3.09)	2323 (3.55)	35636 (3.06)	76575 (3.14)
			DL	1242 (3.09)	2262 (3.45)	35636 (3.06)	76143 (3.13)
	MT	No	N/A	924 (2.30)	1514 (2.31)	27273 (2.35)	31432 (1.29)
		Yes	VT	958 (2.38)	1819 (2.78)	28002 (2.41)	34740 (1.42)
			DL	958 (2.38)	1906 (2.76)	27996 (2.41)	34266 (1.40)

From Table 4-2, several observations can be made:

(1) Recovery overhead: The recovery logic has small overhead (6%~52%) compared to triplication (120%~241% for MT and 300% for HR).

(2) Combinational (LUTs) vs. non-combinational (registers) overhead: The overhead of non-combinational logic for recovery is not very large (6%~9%), but the combinational overhead is generally larger and varies among designs and libraries (12%~52%). Combinational overhead will be discussed in the paragraphs (3) through (5).

(3) Comparison between different libraries: The combinational overhead obtained from LSI lcbg10p library is smaller than that from XCV1000 FPGA library. This is because in XCV1000, most combinational logic gates have to be synthesized into 4-input LUTs even when only 2-input or 3-input gates are needed. LSI library has more different logic cells and is more flexible for logic mapping. Although FPGAs lack the flexibility of logic block mapping, they provide the flexibility of reconfiguration so that we can exploit it to further repair a design with permanent fault [Huang 01b].

(4) Hardware redundancy vs. multi-threading: In our case, all multi-threaded designs, with or without recovery, require less hardware overhead than any of the hardware-redundant designs. The reduced sequential overhead comes from the shared registers in common logic blocks. The reduced combinational overhead comes from the shared computation resources such as adders and multipliers. However, the number of clock cycles needed for a three-threaded design is more than clock cycles for hardware

redundant designs. This shows that in applications where hardware resources are limited but performance requirements are not so strict, multi-threading can be an appealing approach. However, this is not necessarily true for all applications. For some designs that do not have idle stages, multi-threading may not be an efficient scheme to implement TMR and can introduce more overhead than hardware redundancy.

(5) State restoration techniques: The direct-load method has smaller combinational overhead than the voted method for state restoration. The two schemes have the same overhead for non-combinational logic. The reduced combinational overhead comes from not using voters in the direct-load method.

In summary, we found that the recovery logic does not introduce much overhead compared to TMR. Among state restoration schemes, the direct-load scheme has less overhead than the voting scheme. The overhead is less in ASICs than in FPGAs, but with the design implemented in FPGAs we could obtain reconfigurability and more flexibility.

4.4 Summary

In this chapter, we have presented a new roll-forward recovery scheme to enhance repliability of TMR systems. Unlike other roll-forward recovery schemes, our recovery scheme exploits the redundancy in TMR so that no re-computation is needed for single-failure recovery. Therefore, it is very suitable for real-time applications. Reliability analysis of a case study has shown that reliability of a TMR design is significantly improved by adding the recovery scheme. We presented a new state restoration scheme, the direct-load scheme, which is capable of recovering faults in a single faulty module. Synthesis results show that the scheme has smaller overhead than the voting scheme, a conventional state restoration approach. However, it cannot recover as many multiple faults as the voting scheme. We have applied the scheme to two forms of TMR: hardware redundancy and multi-threading. Both hardware redundancy and multi-threading have shown the reliability improvement. Synthesis area results show that the recovery scheme introduces small overhead in both the TMR techniques.

Chapter 5

Permanent Fault Repair for FPGAs with Limited Redundant Area

Permanent fault repair in FPGAs is accomplished by reconfiguring the chip so that faulty elements are not used. Many techniques have been presented to provide permanent fault removal through reconfiguration [Emmert 98] [Dutt 99] [Lach 99] [Huang 01b]. With these schemes, permanent faults in an FPGA can be repaired as long as there exist enough routable fault-free elements on the chip so that a design can be changed into a new configuration and faulty elements are not used. However, for designs that use high percentage of FPGA resource, when a permanent fault occurs, there might not be sufficient fault-free routable elements left for permanent fault repair. For designs that have long mission time, fault-free routable elements may be exhausted because of permanent fault repair during its mission. In these cases, further permanent damage can no longer be repaired. To solve the problem, the design has to be reconfigured into a new fault tolerant design with smaller area, so that the permanent faults can be avoided. A traditional way is to remove design elements at the modular level, such as TMR/Simplex or self-purging. However, modular removal may introduce impact on system availability. For example, consider a TMR system with error detection and a duplex system. When a fault occurs in each system, the TMR system can still function, but the duplex system has to be stopped for recovery or repair. Therefore, if we use module removal to remove permanent fault in a TMR system, the resulting duplex system will have lower availability than the original TMR system. In this chapter, we present new methods of permanent fault repair in FPGAs to reduce impact on system availability by reconfiguring the original design into a new fault tolerant design that fully utilizes the available fault-free FPGA area.

5.1 Problem Definition

The problem we are solving here is how to find a configuration for an FPGA-based design when available chip area is reduced because of permanent faults. The new configuration needs to have small availability reduction compared to the module removal

approach. Since TMR is widely used in fault tolerant systems [Siewiorek 00] and can be used for longer missions when recovery mechanisms are implemented [Yu 01], we focus on area reduction in TMR systems.

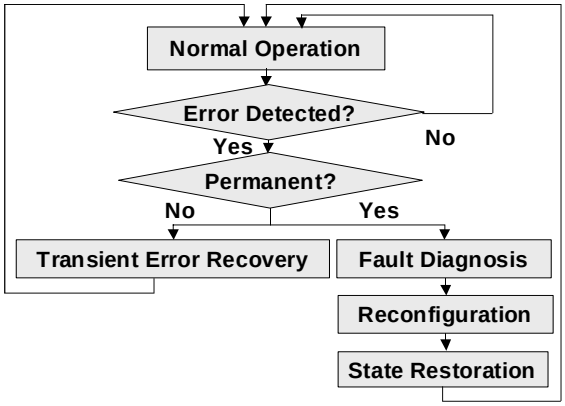


Figure 5-1: Error recovery in FPGAs.

Error recovery for FPGA-based systems is described in Sec. 2.2. Fig. 5-1 is a copy of Fig. 2.2 presented here for the readers' convenience. From the error recovery flow, it is clear that CED and transient error recovery schemes are required for FPGA-based designs. We assume that fault diagnosis and reconfiguration control mechanisms are available off-chip. These mechanisms can be implemented in another FPGA for fault tolerance and recovery purposes [Mitra 00a]. States and data for recovery after permanent fault repair are stored by the other FPGA, too. CED and transient recovery mechanisms are built on-chip within an FPGA-based design. States and data for transient error recovery are handled by the design itself. In this chapter, we present designs that have less area than a TMR but still have error recovery capability.

5.2 Design Techniques

We present three fault tolerant design techniques: (1) duplex with a checking block, (2) hybrid TMR-Simplex-CED, and (3) hybrid Duplex-Simplex-CED. Figure 5-2 (a) shows a TMR system with roll-forward recovery, and Figs. 5-2 (b), (c), and (d) illustrate the design techniques respectively. These designs all contain the features required for error recovery and are able to recover from single faults. We assume that a design can be partitioned into two portions. Unlike conventional fault tolerant designs, these designs are adjusted according to the available FPGA area. When a transient fault

occurs, rollback or roll-forward recovery is used depending on location of faults and design structure. When a permanent fault occurs, an error indication is raised. The choice of rollback or roll-forward will be explained in the following paragraphs.

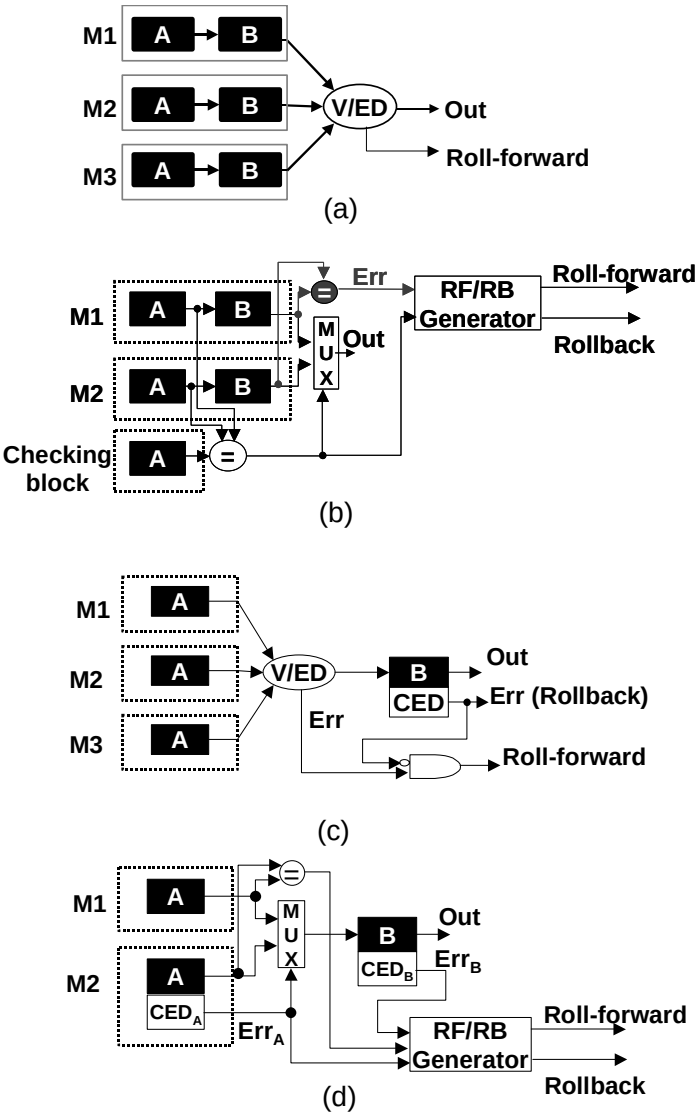


Figure 5-2: Designs with less redundant area than TMR. (a) The original TMR design. (b) A duplex system with a checking block. (c) A hybrid TMR-Simplex-CED design. (d) A hybrid Duplex-Simplex-CED design.

In a duplex system with a checking block, one of the original three modules in TMR is changed to a checking block. The checking block implements a portion of the normal function of a module, and the portion size depends on the available FPGA area. As shown in Fig. 5-2 (b), the original M3 is changed to block A, and the area saved is

approximately the area of block B. The outputs of the checking block are compared with the outputs from block A in M1 and M2. The two modules, M1 and M2, are compared with each other. When there is a mismatch between the duplex modules, the module that agrees with the checking block is selected, and roll-forward is used to restore data in the other one. If there is no mismatch between the two modules but they disagree with the checking block, roll-forward is used to restore data in the checking block. When there is a mismatch between the duplex modules and they both agree with the checking block, rollback is used. This situation happens when faults occur in block B.

In a hybrid TMR-Simplex-CED design, we partition the original simplex design into two parts. As shown in the example of Fig. 5-2 (c), the design is partitioned into blocks A and B. A is triplicated; and CED, such as a parity checker or diversified duplication [Mitra 99], is added to B. A and B can have signals connected to each other. The area saved by the design compared with TMR is approximately the area of block B. The partition depends on the available FPGA area and design constraints. When errors occur in the partition with CED, rollback is used. Otherwise, when errors occur in the partition with TMR, roll-forward is used [Yu 01].

A hybrid Duplex-Simplex-CED design is shown in Fig. 5-2 (d). Block A is duplicated, and their outputs are compared to each other. CED block is built for block A in M2 and block B. The area saved is approximately the area of block B. Recovery approaches depend on CED results and the comparison between the duplicated A blocks. When errors occur in block B with CED, rollback is used for error recovery. Otherwise, when there is a mismatch between the duplex and the CED block in M2 indicates an error, the results from the block A in M1 will be used, and roll-forward is used to restore data in the block A in M2. When there is a mismatch and the CED block in M2 does not indicate an error, the results from the block A in M2 will be used, and roll-forward is used to restore data in the block A in M1.

We are going to compare our designs with a conventional module removal approach. For a TMR system, one approach to module removal for permanent fault repair is to remove the faulty module and change it to a duplex system [Jing 91]. Roll-forward can be used for transient error recovery in TMR, and rollback can be used in a duplex.

We denoted the module removal approach as *TMR-Duplex* throughout the chapter. Figure 5-3 illustrates the idea of TMR-Duplex.

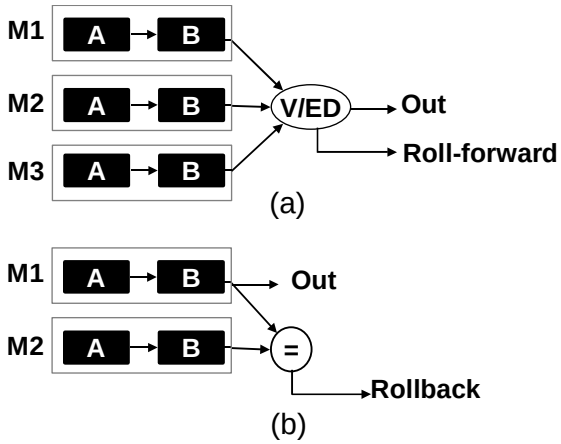


Figure 5-3: Permanent fault repair through a module removal approach, TMR-Duplex: (a) the original TMR design, and (b) duplex.

5.3 Evaluation Metric and Comparison

In this section, we define the metric to evaluate the design techniques, and compare the design techniques with the module removal approach, TMR-Duplex.

Consider a computing system with both rollback and roll-forward recovery capability. When a transient fault occurs, the choice of rollback or roll-forward depends on the location of the fault. Errors are assumed to occur independently in each unit area in unit time. We use *rollback probability* to evaluate a design. It is the probability of initiating rollback recovery when a system is under normal function. It represents the overhead of recovery. For each rollback recovery, the computation process rollbacks to its previous checkpoint. Hence, the average re-computation time for each checkpoint is the product of the rollback probability and the time between two consecutive checkpoints. In a real-time application, a high rollback probability may not be acceptable, since a deadline may be missed because of re-computation. In our analysis, only single faults are considered because the probability of multiple faults is very small. We assume 100% single fault detection coverage for all CED techniques.

Referring to Figs. 5-2 (b) through (d), a rollback recovery will only occur if an error occurs in one of the blocks B. Since we assume errors happen independently in unit

area, rollback probabilities can be computed as probabilities of transient errors happening in the blocks B, that is,

Rollback probability = Transient error probability per unit area unit time× blocks B area.

Rollback probability depends on design structures and area overhead of CED techniques. CED overhead depends on the error detecting mechanism and applications. It can vary from very small to over 100% [Sparmann 96] [Mitra 00b]. We chose 90% of CED overhead as an example. The effect of CED overhead will be described in the next paragraph. The rollback probabilities of the design techniques and TMR-Duplex are plotted in Fig. 5-4. The probabilities are normalized to a duplex system. The area axis is normalized to the area of a simplex system. All designs fully utilize the available area except TMR-Duplex. In TMR-Duplex, when the available area is smaller than the area of a TMR, the design is changed to a duplex and its area is twice the simplex area.

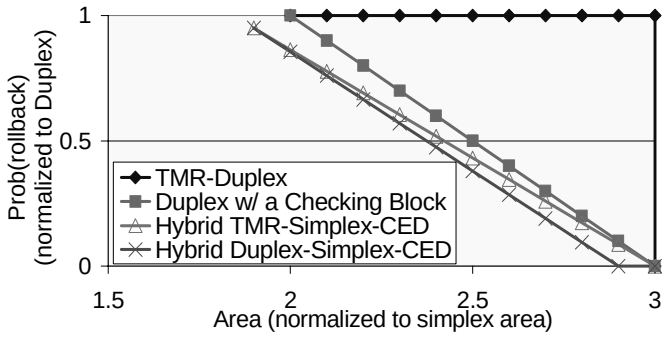


Figure 5-4: Rollback probabilities when CED overhead is 90%

As shown in Fig. 5-4, our approaches all have rollback rate smaller than TMR-Duplex. The relation of the rollback probabilities among the three design techniques and TMR-Duplex is

Hybrid Duplex-Simplex-CED < Hybrid TMR-Simplex-CED
< Duplex with a checking block < TMR-Duplex.

When the CED overhead is less than 100%, the relation should always be the inequality above. The reason is that if a design utilizes more portion of CED, when CED overhead is less than 100%, it can reduce the area overhead of CED and have more area for redundancy, therefore it will have lower rollback probability. From our designs, the hybrid Duplex-Simplex-CED always has the largest portion of CED, and the hybrid TMR-Simplex-CED has the second largest portion of CED. These two designs can

reduce more area overhead than the duplex with a checking block when the CED overhead is smaller than 100%. The lower the CED overhead is, the more the differences among the approaches are. When the CED overhead is 100%, the three design approaches will have the same rollback probabilities since the area they need is the same. One example of 100% CED overhead is duplication, assuming that the area of comparators is negligible.

For CED overhead greater than 100%, the relation of the rollback probabilities among the three design techniques and TMR-Duplex is

Duplex with a checking block < Hybrid TMR-Simplex-CED

< Hybrid Duplex-Simplex-CED < TMR-Duplex.

The rollback probability directly affects the performance of real-time systems. As described in Sec. 2.2, three major components of timing overhead are: $overhead_{CED}$, $overhead_T$, and $overhead_P$. In general, $overhead_{CED}$, is smaller than $overhead_T$ and $overhead_P$. because the gate delay of CED circuitry is usually much smaller than the computation time of normal operations, and the time of fault diagnosis and reconfiguration. For example, if duplication is used for CED, $overhead_{CED}$ will only be the delay overhead of comparators. $Overhead_P$ can be an important factor. It will be discussed in Sec. 6.4. For $overhead_T$, as discussed in Sec. 4.1, when a TMR system is used, we can eliminate the need of rollback. Therefore, in TMR systems, the impact of $overhead_T$ can be minimized. However, if the available FPGA area is reduced and a TMR system does not fit anymore, $overhead_T$ will be increased. This increment is inevitable, as the available area does not provide sufficient redundancy for 100% roll-forward recovery. However, by using the schemes presented in this chapter, $overhead_T$ can be reduced compared with conventional module removal techniques.

5.4 Case Study

In this section, we describe implementation of our permanent fault repair scheme in the robot controller, described in Sec. 3.1, as our case study. The controller consists of registers, a finite state machine, and arithmetic units for computation. For ease of implementation, we choose duplication as our CED technique for the controller. Hybrid-

TMR-Simplex-CED is chosen for area reduction. The module removal scheme, TMR-Duplex, is also implemented for comparison purpose.

We partition the controller in a way to minimize signals connecting blocks A and B. The reason is that additional voters are needed for signals from blocks A to blocks B, and comparators are needed for signals from blocks B to blocks A.

The TMR-Duplex design can be implemented in two ways. We denote them as TMR-Duplex$_1$ and TMR-Duplex$_2$. TMR-Duplex$_1$ uses similar structure as self-purging redundancy [Losq 76], and both roll-forward and rollback recovery mechanism are implemented. The system switches from TMR with roll-forward recovery to Duplex with rollback recovery when a permanent fault occurs and is detected. TMR-Duplex$_2$ makes use of reconfiguration. The system changes from TMR with roll-forward recovery to Duplex with rollback recovery by changing the configuration of the FPGA.

The designs are synthesized with Xilinx XCV300 library. The area results are listed in Table 5-1, including area for Lookup Tables (LUT) and registers (REG). The columns of partition for TMR and Simplex-CED are area results before adding recovery logic. The numbers in parentheses are the area normalized to a simplex controller without recovery logic. The column of total area is the area with all recovery overhead.

Table 5-1: Synthesis area of different designs.

Design		Partition for TMR		Partition for Simplex-CED		Total	
		LUT	REG	LUT	REG	LUT	REG
TMR-Duplex$_1$	TMR with roll-forward	---	---	---	---	2821 (3.24)	1371 (3.56)
	Duplex with rollback	---	---	---	---	1992 (2.20)	1101 (2.63)
TMR-Duplex$_2$		---	---	---	---	3428 (3.79)	1568 (3.74)
Hybrid TMR-Simplex-CED		263 (0.29)	54 (0.13)	642 (0.71)	365 (0.87)	2581 (2.85)	1170 (2.79)

Compared with the original design, TMR with roll-forward, it is shown that the hybrid TMR-Simplex-CED design has smaller area. It means that the new design can

effectively reduce design area. Therefore, it can be used when available FPGA area is limited. Compared with TMR-Duplex$_1$, although the new design has larger area than the Duplex with rollback recovery, but as shown in the previous section, the new design has the advantage of smaller rollback probability. Compared with TMR-Duplex$_2$, the new design has smaller area, and it also has smaller rollback probability.

5.5 Summary

In this chapter we presented three new permanent fault repair schemes for FPGA-based computing systems. When no routable fault-free elements are available for conventional permanent fault repair schemes, our schemes reconfigure the design into another fault tolerant design, which needs smaller area. We have examined three design structures to adapt to limited FPGA area and to provide reliability and availability. These designs are found to have improved availability compared to the module removal approach. Hence, they are more suitable for real-time applications where availability is a critical issue. The choice of designs depends on the CED overhead. One of the repair schemes is implemented in a robot controller as a case study. The synthesis results show that our scheme can effectively reduce design area; therefore, it can be used when available area is limited because of permanent faults.

Chapter 6

Fault Tolerant Communication between Repair Controllers in the Dual-FPGA Architecture

Redundant buses have been used to repair permanent faults in chip interconnects. However, redundant buses can significantly reduce bandwidth in chip interconnects. In this chapter, we present a new scheme to repair single faults in interconnects between two FPGAs. Our scheme eliminates the need of redundant buses and only needs one spare wire. Our scheme is applied in Stanford's Dual-FPGA architecture, which permits permanent fault repair without device replacement. We concentrate on the communication between repair controllers in the architecture.

6.1 Dual-FPGA Architecture

The Dual-FPGA architecture is shown in Fig. 6-1. Two FPGAs are used for task execution, and they monitor each other. Memory devices are used to store configuration files and user data. Repair controllers are embedded in each of the FPGAs to perform error monitoring, fault diagnosis, and reconfiguration tasks. When a permanent fault occurs in one of the FPGAs and is detected, the repair controller in the FPGA reports the fault to the controller in the other FPGA. The other controller evokes fault diagnosis for reconfiguration.

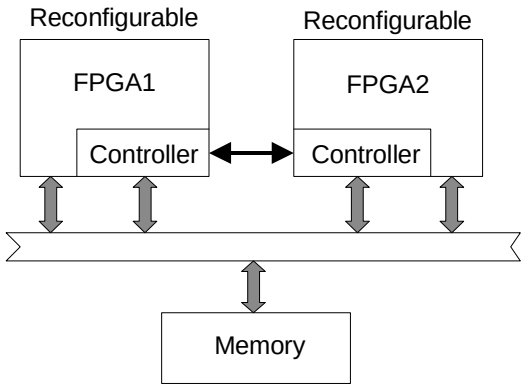


Figure 6-1: Dual-FPGA ACS Architecture.

The Dual-FPGA architecture is capable of repairing and recovering the system from all single faults occurred in the user application circuitry and memory. The system can also be repaired from most of the faults in configuration circuitry. However, there exist single point failures in the architecture. These failures will be discussed in Sec. 6.4. Transient faults in the user application circuits can be recovered by implementing error detection and recovery mechanisms in user application modules. Transient faults in configuration circuitry can be recovered by downloading correct configuration. Permanent faults within FPGAs can be repaired through reconfiguration [Emmert 00] [Huang 01b]. Faults in memory devices are corrected by error correcting codes or masked by using redundant devices. For permanent faults that occur in communication buses and wires between chips, redundant buses have been used in literature [Chan 00].

The use of a redundant bus for fault tolerant communication is generally expensive, however. For communication between the two FPGAs, the reconfiguration capability can be used in conjunction with fault tolerance techniques to reduce the number of redundant pins. In this chapter, we present a communication scheme that provides fault tolerance without the use of redundant buses. We concentrate on communication between repair controllers, because the fault tolerance of the repair process needs to be guaranteed in the Dual-FPGA architecture. Our scheme can also be used for fault tolerant data communication between the two FPGAs.

6.2 Repair in Dual-FPGA Architecture

In this section, we describe the repair process in the Dual-FPGA architecture, and the role the repair controllers play in the repair process.

6.2.1 Block Diagram

Figure 6-2 shows a block diagram of the Dual-FPGA architecture with a detailed internal diagram of one of the FPGAs. The other FPGA has an identical internal structure. This block diagram will be used to illustrate the repair process in the architecture. To simplify our discussion, we divide the repair tasks into different units. Each FPGA contains the following blocks: one or more user function units, a readback/writeback unit, a fault diagnosis unit, a configuration unit, and a repair controller. The communication between the two FPGAs is accomplished through the

repair controller. The repair tasks can also be implemented in the repair controller. The function of the repair controllers is described in Sec. 6.2.2. The details of other blocks are explained in Appendix D.

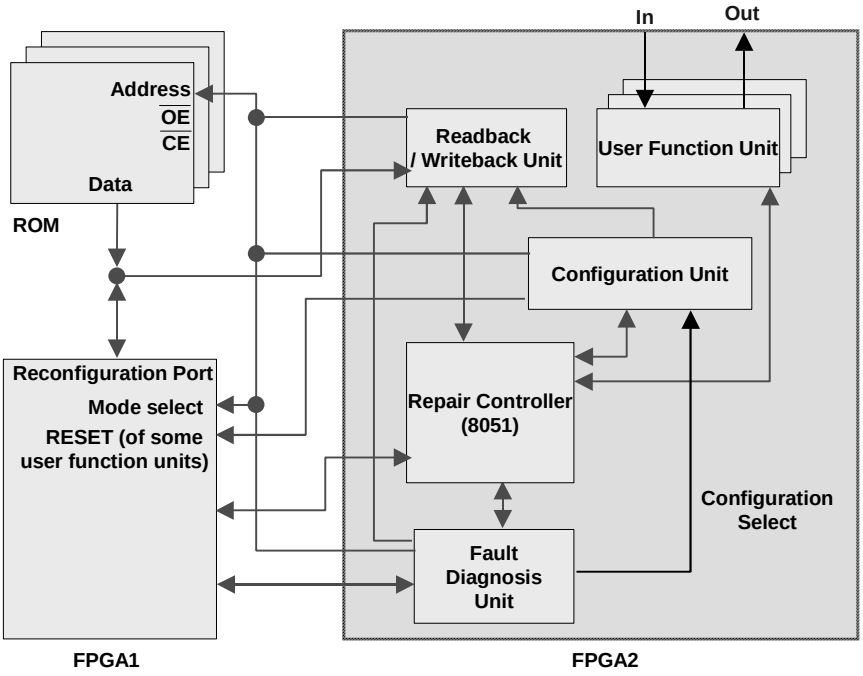


Figure 6-2: Block diagram of the Dual-FPGA architecture.

In the FPGAs, the user function units perform user computation tasks. Error detection mechanisms are implemented in each of the user function units. The readback/writeback unit is responsible for correcting transient errors in configuration cells. To explain the repair process in the Dual-FPGA architecture, suppose a permanent fault occurs in one of the user function units in FPGA1. We use the subscript "1" to indicate a unit in FPGA1, and the subscript "2" to indicate a unit in FPGA2. The fault is detected by the error detection mechanism in the faulty unit, and an indication signal is raised. Repair controller$_1$ collects fault indication signals from all of the user function units, and receives the fault indication from the faulty unit. When repair controller$_2$ asks repair controller$_1$ about its current error status, repair controller$_1$ reports the fault. To repair FPGA1, repair controller$_2$ sends commands to fault diagnosis unit$_2$ to initiate diagnosis. Fault diagnosis unit$_2$ applies test patterns to FPGA1 and collects responses to locate the fault. When a fault is located, fault diagnosis unit$_2$ selects a proper configuration that avoids the fault, and sends the configuration selection to configuration

$unit_2$. Configuration $unit_2$ generates proper address according to the configuration selection information, and reconfigures FPGA1. When FPGA1 is reconfigured, configuration $unit_2$ resets user function units in FPGA1. After reset, the computation processes in FPGA1 can be resumed.

6.2.2 Repair Controller

The repair controller plays a critical role in the repair process. It is responsible for communication with the other FPGA. It is also responsible for controlling other units to perform repair tasks. To provide fault tolerance for the repair controller, it is duplicated and the outputs are compared. If a mismatch occurs, the controller's process is recovered by retry. If the mismatch occurs again after retry, a *perm_ind* signal that indicates a permanent fault will be raised. Figure 6-3 illustrates the flow of the repair functions in the repair controller. These steps are explained as follows.

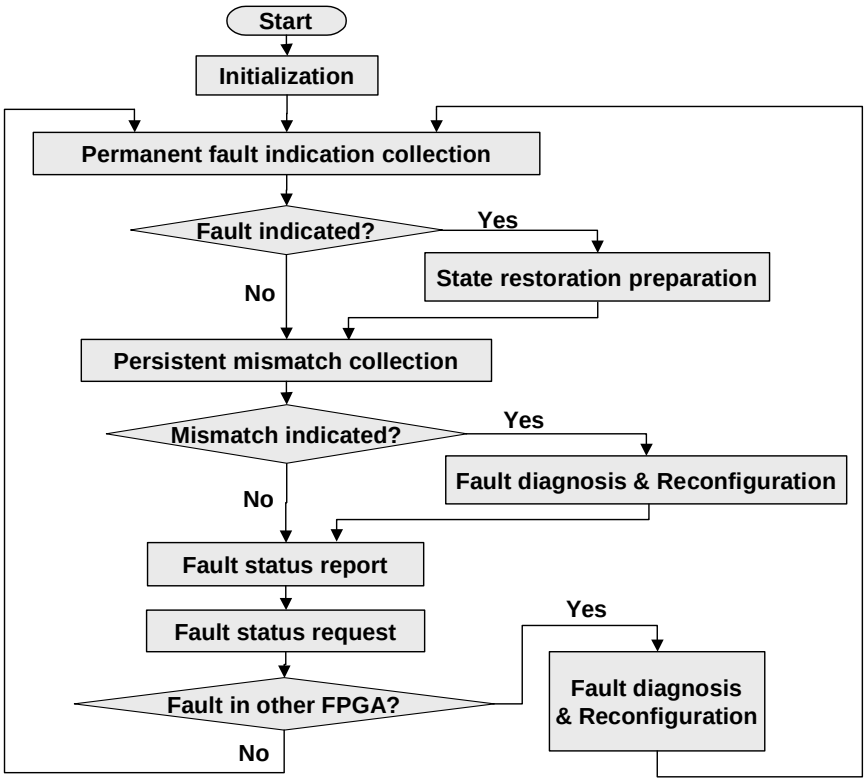


Figure 6-3: Repair function flow of the repair controller.

(1) Initialization.

After the reset of the FPGA is de-asserted, the repair controller initializes its *error status register*. This register stores the current error status of the FPGA. Its value is

initialized to NO_ERR when the FPGA reset is de-asserted. If the controller detects a permanent fault indication among units in its FPGA, it sets the register to ERROR.

After register initialization, the repair controller reads the signals in the communication channel between the controllers, and determines whether the other repair controller has requested reconfiguration. This is used to repair faults in the communication channel between repair controllers. Details of the repair process will be explained in Sec 6.3.3. If a reconfiguration request is received, the repair controller sends a command to the Reconfiguration Unit to initiate reconfiguration.

(2) Permanent fault indication collection.

The repair controller collects the *perm_ind* signals, which indicate permanent faults, from all of the user function units, the readback/writeback unit, the fault diagnosis unit, and the configuration unit from the same FPGA. It also collects *perm_ind* from the controller itself.

The methods for collecting *perm_ind* signals depend on the fault diagnosis schemes for the controller's FPGA. If the diagnosis scheme requires faulty unit identification, the repair controller needs to identify which one of the units indicates a permanent fault. If the diagnosis scheme does not require the faulty unit identification, the repair controller only needs to collect all *perm_ind* signals and does not need to know what unit is faulty.

(3) State restoration preparation.

When a permanent fault occurs, a repair process is needed. After the repair process, computations are resumed. To resume the computations, internal states in user function units need to be restored. Therefore, user function units that need to be reconfigured have to store their internal states in memory devices. After the repair process completes, these internal states are loaded from memory devices so that the computations can be continued. We define the process of storing internal states in memory devices as *state restoration preparation*. For state restoration preparation, a repair controller's tasks depend on FPGA reconfiguration schemes. If whole-chip reconfiguration is used, the repair controller will notify all units for state restoration preparation when it receives a permanent fault indication. If partial reconfiguration is used and only a faulty unit needs to be reconfigured, the repair controller does not need to

notify other units for state restoration preparation. The faulty unit will start its own state restoration preparation process when its *perm_ind* is raised.

(4) Persistent mismatch collection.

A readback/writeback unit reads the current configuration out from the other FPGA and compares it with stored configuration to detect a transient error in configuration memory cells. If a mismatch occurs, it will write the stored configuration back to the FPGA to correct the error. If a mismatch occurs again right after writeback, the mismatch will be recorded. We define the reoccurred mismatch as a *persistent mismatch*. A signal called *persistent_mismatch* that indicates the second mismatch will be raised. The repair controller examines the *persistent_mismatch* signal from the readback/writeback unit. When a persistent mismatch indication is received, the repair controller will initiate fault diagnosis to locate the fault. If the fault is in the configuration port, the repair controller requests both the readback/writeback unit and the configuration unit to switch to the other configuration port. If the fault is located in the configuration memory, a new configuration file will be loaded into the other FPGA and the faulty element will not be used.

(5) Fault status report.

Before explaining the fault status report, we list the basic commands that are used for the communication between the two controllers:

IDLE: a controller is not sending any request or any reply.

REQUEST: the command used for requesting error status from the other controller.

NO_ERR/CNFG: this command has two meanings. It can be used as NO_ERR, a reply to REQUEST when no error is recorded in the *error status register*; or as CNFG, a reconfiguration request. If the command is received during initialization, it is treated as CNFG; otherwise, it is treated as NO_ERR. The use of NO_ERR will be explained in detail in the following paragraphs, and the use of CNFG will be described in Sec. 6.3.3.

ERROR: the reply to REQUEST when an error is recorded in the *error status register*.

RETRY/DIAG: the command used for retry and diagnosis. When an error is detected in the communication channel, the repair controller at the receiving end sends RETRY/DIAG to request the other controller to resend its command. If an error is

detected again after retry, the repair controller at the receiving end sends RETRY/DIAG again to indicate the start of fault diagnosis. The details will be described in Sec. 6.3.1 and Sec. 6.3.2.

Now we are going to explain the fault status report between the repair controllers. To report the fault status, one of the repair controllers first reads the signals in the communication channel and determines whether it is REQUEST. If the received command is REQUEST, the repair controller will reply with its current error status. According to the content of the *error status register*, it will reply NO_ERR/CNFG when the content of the register is NO_ERR, and reply ERROR when the content of the register is ERROR. If faulty unit identification is required, the repair controller will send the faulty unit ID to the other controller following the ERROR reply. Serial transmission for the unit ID can be used to reduce pins needed between the controllers. If the received word is not REQUEST, the repair controller will skip reporting its fault status.

(6) Fault status request.

Before sending a fault status request, the repair controller first checks if the fault diagnosis unit is testing the other FPGA or if the configuration unit is reconfiguring the other FPGA. If the other FPGA is being diagnosed or reconfigured, the repair controller will not send a fault status request. If the other FPGA is neither diagnosed nor reconfigured, the repair controller will send the fault status request command, REQUEST, to the other repair controller. At the moment REQUEST is sent, the other controller may be executing other steps and may not respond immediately. The repair controller will wait for the response from the other controller. A *timer* is used to detect a *timeout*. A *timeout* occurs when the response is not received within a certain period of time, which depends on the control program in the repair controller. It can happen if the transmitted commands are lost during communication, or when a fault occurs in clock circuitry. If a timeout is detected, the repair controller will send REQUEST again. If a timeout is detected again, the repair controller will send a command to the fault diagnosis unit to initiate diagnosis.

When a reply is received from the other controller, the repair controller examines the received command and determines whether it is NO_ERR/CNFG or ERROR. If it is NO_ERR/CNFG, the repair controller sends an IDLE signal to indicate that the

communication channel is idle. If the received command is ERROR, depending on the fault diagnosis scheme, the repair controller may also receive the faulty unit ID from the other controller.

(7) Fault diagnosis and reconfiguration.

When an error report, ERROR, is received from the other repair controller, the repair controller will start the repair process. First, it will send a command to the fault diagnosis unit to initiate fault diagnosis. If faulty unit identification is needed, it also passes the identification to the fault diagnosis unit. Then the repair controller waits until the fault diagnosis and the reconfiguration complete.

6.3 Fault Tolerant Communication Scheme between Repair Controllers

In this section we describe the fault tolerant communication scheme between repair controllers. To provide fault tolerance, three steps are needed (1) error detection in the communication channel, (2) fault diagnosis to find the faulty wire, and (3) reconfiguration to avoid the faulty wire. These steps are described in the following sections. The wires between the controllers are assumed to be bi-directional. Only single faults are considered in our scheme.

6.3.1 Error Detection

In this section, we first explain the codes we chose for the five commands described in Sec. 6.2.2, and then explain the error detection flow in the communication channel between the repair controllers.

Many error detection codes have been developed in literature [Wicker 95]. Examples include Hamming codes, cyclic redundancy check (CRC) codes, and parity checks. In our communication scheme, we use a special code for error detection. Unlike most of the error detection codes, the codewords we use have different *Hamming distances* from one another. *Hamming distance* is the number of different bits between two words [Wicker 95]. Distance 2 provides single error detection capabilities, while distance 3 provides single error correction capabilities.

Figure 6-4 shows the Hamming distance among the five commands described in Sec. 6.2.6. To explain the choice of the Hamming distance, we divide the commands into two sets according to when they are used:

Set_1: {REQUEST, NO_ERR/CNFG, ERROR, IDLE}. Set_1 is the commands that are used during normal fault status requests and fault status reports, when no faults are present in the communication channel.

Set_2: {REPLY/DIAG, NO_ERR/CNFG}. Set_2 is the commands that are used for fault tolerance purposes, when a fault occurs in the communication channel. NO_ERR/CNFG is included in both of the sets because it can be used during both normal functions (NO_ERR) and when a fault occurs (CNFG).

We need to detect single errors in Set_1. Because parity checks are easy to implement, we chose parity checks for the error detection scheme for commands in Set_1. To assign the same parity to every of the four commands, the commands need to have even Hamming distance from one another.

For the commands in Set_2, the code needs to provide not only single error detection capabilities but also single error correction capabilities. The reason is that these two commands need to be distinguished from all other commands even when a fault is present in the communication channel. Otherwise, a fault could corrupt the commands so the commands would not be understood by the other controller. To have single error correction capabilities, these two commands need to have at least Hamming distance 3 from all other commands.

From the discussion in the previous two paragraphs, we assign REQUEST, ERROR, and IDLE to have Hamming distance 2 from one another. RETRY/DIAG is assigned to have distance 3 from any other commands. For NO_ERR/CNFG, because its Hamming distance from other commands needs to be an even number and at least 3, we choose it to have distance 4 from any other commands. Table 6-1 lists an example of the codeword that satisfies the Hamming distance requirement.

Table 6-1: An example of the codeword that satisfies the Hamming distance requirement.

Command	Codeword
REQUEST	11001
NO_ERR/CNFG	00100
ERROR	11010
IDLE	11111
RETRY/DIAG	00011

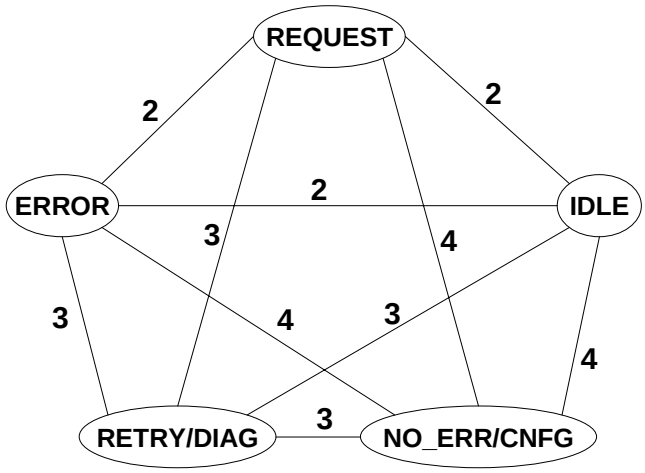


Figure 6-4: Hamming distance among the commands.

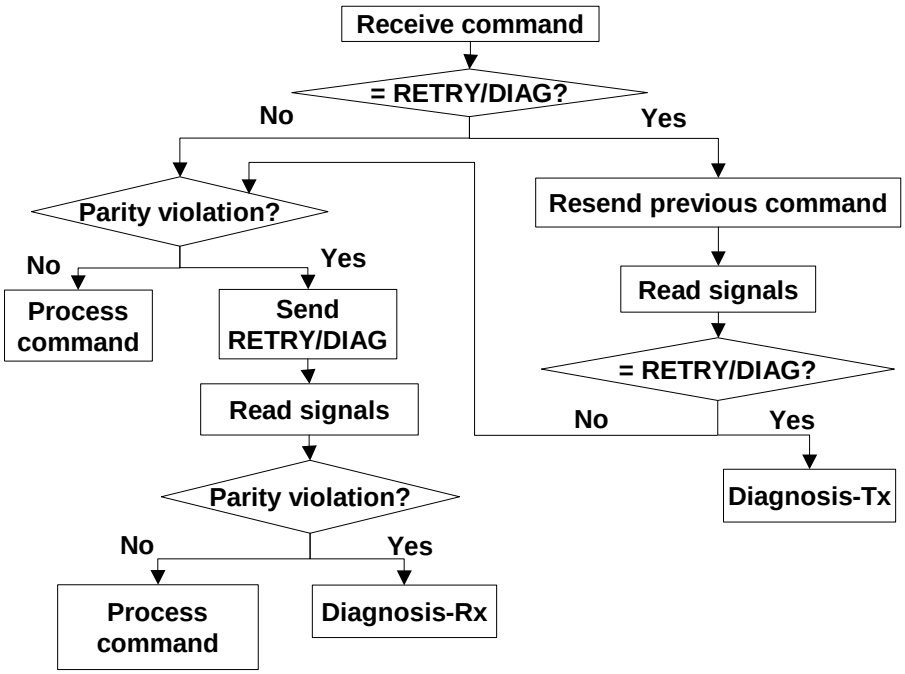


Figure 6-5: Error detection for the communication between the repair controllers.

Figure 6-5 illustrates the flow of error detection for the communication between the repair controllers. When a repair controller receives a command from the other controller, it first determines whether it is RETRY/DIAG. If RETRY/DIAG is received, it indicates that the previous command the controller sends out is corrupt. Therefore, the controller resends the previous command. If RETRY/DIAG is received again, diagnosis will be needed. The box Diagnosis-Tx will be explained in Sec. 6.3.2. If the received

command is not RETRY/DIAG, the repair controller performs parity checks to detect if there is a parity violation. If no violation is detected, the controller will process the received command. If parity violation is detected, the controller sends RETRY/DIAG to the other controller to ask for retry. If parity violation is detected again after retry, diagnosis will be needed. The box Diagnosis-Rx will be explained in Sec. 6.3.2.

6.3.2 Interconnect Diagnosis

Different techniques have been presented to test and diagnose interconnect between chips [Wagner 87] [Wang 89] [Chan 92] [Feng 95] [Hsu 96]. However, all of the schemes mentioned above need more than one test pattern. We present a new diagnosis scheme for the interconnection between the repair controllers in the Dual-FPGA architecture. Instead of testing interconnects off-line and applying multiple patterns, we apply diagnosis on-line and use only one test pattern.

The scheme we present is called *command-bounce diagnosis*. It uses corrupt commands as test patterns for diagnosis. Suppose a permanent fault occurs in the communication channel and is detected as described in Sec. 6.3.1. When a permanent fault is detected, a repair controller can be performing one of two roles: it can be the controller that receives the corrupt command and detects the fault, or the controller that sends the corrupt command. Referring to Fig. 6-5, if the controller receives the corrupt command, it will go to the Diagnosis-Rx step. If the controller sends the corrupt command, it will go to the Diagnosis-Tx step. We will discuss Diagnosis-Rx and Diagnosis-Tx in the following two paragraphs.

For Diagnosis-Rx, the repair controller will resend RETRY/DIAG to the other controller to notify that the diagnosis process is going to start. Following the RETRY/DIAG command, the repair controller bounces (resends) the corrupt command back to the other controller as the test pattern.

For Diagnosis-Tx, the repair controller has received RETRY/DIAG that indicates the start of diagnosis. The repair controller reads one command in the communication channel followed by RETRY/DIAG and compares it with the previous command it sends out. By comparing the original command and the bounced command, the bit that mismatches is identified as the faulty bit. The details of the explanation is in Appendix D.

6.3.3 Reconfiguration

To repair faulty interconnection economically, we use reconfiguration as our repair scheme. Unlike the repair of faulty elements inside an FPGA [Emmert 98] [Huang 01b], the repair process for faulty wires connecting FPGAs requires reconfiguration of multiple FPGAs. Figure 6-6 illustrates the reconfiguration process. The two FPGAs are connected to each other through the communication channel. As shown in Fig. 6-6 (a), pins of FPGA1 are connected to pins of FPGA2 through wires. Spare wires are needed to replace the function of faulty wires. Suppose a permanent fault occurs in a wire connecting the two FPGAs, and is detected and located by the repair controller in FPGA1. To repair the system, we need to reconfigure both FPGAs to avoid the faulty wire. In Fig. 6-6 (b), FPGA1 reconfigures FPGA2, so that the output that is originally connected to the faulty wire is moved to connect a spare wire. After the reconfiguration, in Fig. 6-6 (c), FPGA2 reconfigures FPGA1, so that the output that is originally connected to the faulty wire is moved to connect the spare wire.

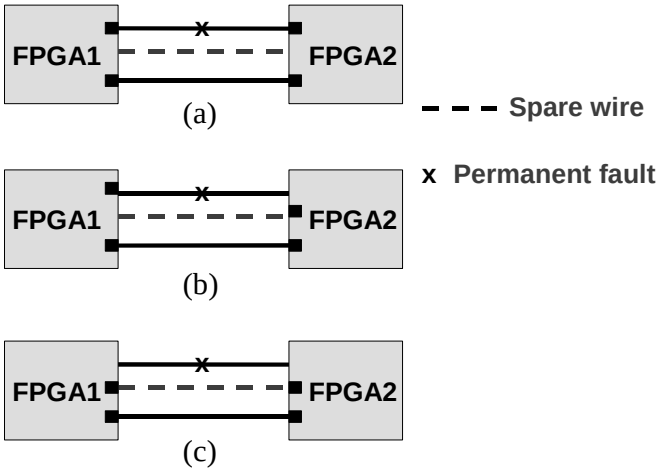


Figure 6-6: Reconfiguration process when a fault occurs in inter-chip communication: (a) original configuration, (b) FPGA2 is reconfigured, and (c) FPGA1 is reconfigured.

To perform reconfiguration, a repair controller can be in two different roles. It may be the repair controller that locates the fault and reconfigures the other FPGA first, such as the repair controller in FPGA1 in the example of Fig. 6-6, or the repair controller that is reconfigured first, such as the repair controller in FPGA2 in the example of Fig. 6-6. We denote the repair controller in FPGA1 as *repair controller*$_1$ and the repair

controller in FPGA2 as *repair controller*$_2$. Figure 6-7 shows the reconfiguration flow for repair controller$_1$. Figure 6-8 shows the reconfiguration flow for repair controller$_2$.

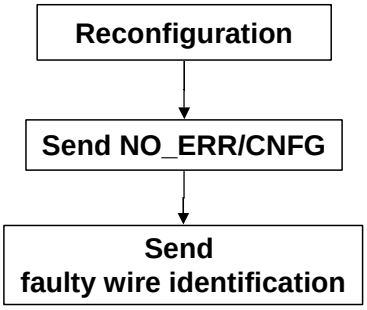


Figure 6-7: Reconfiguration flow for repair controller$_1$.

When repair controller$_1$ locates the interconnect fault, as shown in Fig. 6-7, the first step is to reconfigure FPGA2. Repair controller$_1$ sends configuration commands to its configuration unit to initiate reconfiguration. When the reconfiguration completes, repair controller$_1$ sends the reconfiguration request, NO_ERR/CNFG, to repair controller$_2$. Following NO_ERR/CNFG, it also sends the faulty wire identification to repair controller$_2$ so that repair controller$_2$ can reconfigure FPGA1 according to the identification information.

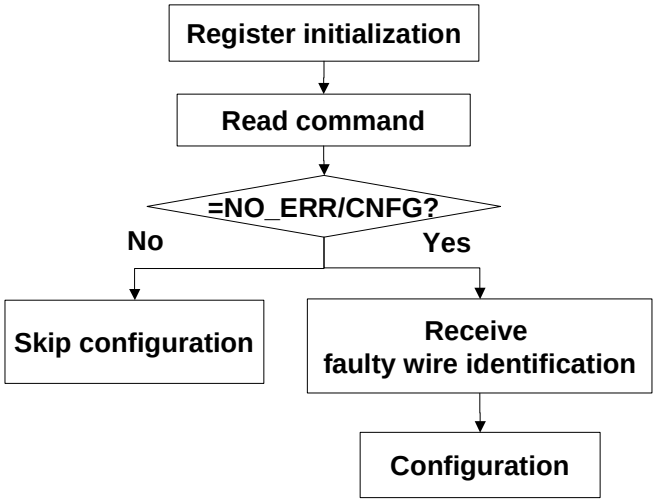


Figure 6-8: Reconfiguration flow for repair controller$_2$.

Repair controller$_2$ does not perform reconfiguration tasks until FPGA2 is reconfigured. As shown in Fig. 6-8, after initializing its registers, repair controller$_2$ reads a command in the communication channel and determines whether it is NO_ERR/CNFG.

If NO_ERR/CNFG is received, repair controller$_2$ reads the faulty wire identification in the communication channel. After that, it sends configuration commands to its configuration unit and waits for the completion of the reconfiguration. If the received command is not NO_ERR/CNFG, repair controller$_2$ will skip the reconfiguration and starts its normal function, as described in Sec. 6.2.2.

The faulty wire identification needs to be coded so that every word can be distinguished from one another even when a wire is mismatched between the two repair controllers. To achieve this, we encode the identification information with the Reduced (6, 3) Hamming code that provides Single-Error-Correcting-Double-Error-Detecting (SEC-DED) capability. Because there are only 5 wires in the communication channel, we replicate the last bit of the Hamming code 4 times to make it two words. Table 2 lists the codeword we use for the faulty wire identification.

Table 6-2: Codeword for the faulty wire identification.

Faulty wire	First word	Second word
1	00000	00000
2	00110	11111
3	01011	00000
4	01101	11111
5	10011	11111

6.3.4 Comparison

In this section, we are going to compare our scheme with the conventional scheme that uses redundant buses for fault tolerance.

Suppose two repair controllers are connected to each other through a communication channel. For a conventional fault tolerant scheme that uses redundant buses, we assume that one parity bit is added for error detection, and the total number of the needed wires is doubled because of redundancy.

For our communication scheme, one additional spare wire is used. In addition, two special commands RETRY/DIAG and CNFG are used. To provide sufficient Hamming distances to the special commands and to the original commands, extra wires are needed. An analysis is done to calculate the required wires in our scheme. The details of the analysis are in Appendix D. Analytical results show that when the total number of

commands is greater than four, not including the special commands RETRY/DIAG and CNFG, our scheme requires fewer wires than the conventional scheme. Moreover, the more commands are needed in the communication, the more wires our scheme can reduce compared to the conventional scheme.

6.4 Limitation and Discussion

Although the Dual-FPGA architecture can be repaired from most of the failures caused by single faults, it has some limitations. In this section, we will discuss the limitation of the architecture and suggest possible solutions.

(1) Memory contention: if the two FPGAs have access to the same memory device, the faulty FPGA may prevent the fault-free FPGA from accessing the memory. To prevent this, one solution is to assign different memory devices to different FPGAs, so that the two FPGAs do not share the same memory chip.

(2) Communication between FPGAs and memory devices: The discussion of our fault tolerant communication scheme focused on the communication between repair controllers. It can also be applied to data communication between the two FPGAs. However, this scheme is not suitable for the communication between an FPGA and memory devices. For the communication between an FPGA and a memory device, redundant memory chips and buses are needed. The reason is that current commercial memory devices do not have any fault tolerance capability to repair faults in input and output pins. Although error-correcting codes are implemented in memory devices, they have not been implemented in input or output pins. In addition, there is no redundant input and output pin for memory devices. Therefore, if one of the pins becomes faulty, the whole memory device needs to be removed from the system. Connecting to redundant memory devices and redundant buses reduces the available pins of an FPGA. One possible solution is to use FPGAs as memory devices and store data in the block RAMs and select RAMs. However, in current technology, the memory capacity of FPGAs is still limited (around the order of mega bits per chip), and it is not practical to use current commercial FPGAs to store configuration files.

(3) Single point failures: There are a few pins in FPGAs reserved for reconfiguration purposes. These pins are not reconfigurable. For most of the special purpose pins, their faults can be tolerated by changing to another reconfiguration mode. However, there are a few pins that are needed in all of the configuration modes. Therefore, when they become faulty, their functions cannot be replaced by other pins. For example, in Xilinx Virtex FPGAs, pins PROGRAM, DONE, and INIT are used in all of the configuration modes. Single point failures will occur if faults occur in one of these pins.

(4) Timing overhead for permanent fault repair: Referring to Fig. 2-2, the repair controller is responsible for controlling the repair tasks including fault diagnosis, reconfiguration, and state restoration. However, the timing overhead introduced by permanent fault repair, $overhead_P$, may have impact on real-time applications. By choosing diagnosis and configuration schemes, we can minimize $overhead_P$ so that the performance impact is reduced. For diagnosis schemes, a configuration-dependent diagnosis scheme can be used to reduce timing overhead. For example, if the diagnosis scheme in [Huang 01c] is used, $overhead_P$ can be as small as the delay of reconfiguring one whole FPGA, which is in the order of milliseconds [Xilinx 01a]. If pre-compiled configurations are used [Lach 00] [Huang 01b], the delay of generating new configurations can be eliminated. Moreover, if partial reconfiguration is used [Xilinx 01b], the configuration time can be significantly reduced. However, for some real-time applications, the delay of partial configuration might still introduce performance impact to the system. One solution is to finish current computation tasks before initiating repair processes when a permanent fault is detected. Hence, the current real-time task can be finished on time. However, incoming tasks need to be postponed until the repair process completes.

6.5 Summary

In this chapter, we have presented a fault tolerant communication scheme between the repair controllers in Stanford's Dual-FPGA Architecture. The scheme is capable of repairing the system from failures caused by single faults in the communication channel.

Unlike conventional fault tolerance schemes for communication, our scheme takes advantage of flexibility of FPGAs and does not need to use redundant buses for permanent fault repair. Therefore, the number of the needed wires is significantly reduced. The repair process includes three steps: (1) error detection, (2) fault diagnosis, and (3) reconfiguration. Parity checks are used for error detection. A diagnosis scheme, the command-bounce diagnosis, was presented in the report. Unlike other diagnosis schemes, command-bounce diagnosis only requires to apply one test pattern to identify the faulty wire. Repair is performed by reconfiguring both of the FPGAs. The limitations of the Dual-FPGA architecture have also been discussed in this chapter.

Chapter 7

Concluding Remarks

This dissertation summarizes my contributions to obtaining fault tolerance in real-time systems using reconfigurable hardware. Because of the large capacity, high performance, and reconfiguration ability, reconfigurable hardware can provide an alternative solution to construct cost effective reliable real-time computing systems. Research for realizing fault tolerant real-time computing systems on reconfigurable hardware has been investigated thoroughly. The feasibility of using reconfigurable hardware in real-time applications is demonstrated, a transient error recovery scheme and a permanent fault repair scheme that are suitable for real-time FPGA-based systems are developed. To guarantee the success of FPGA repair processes, a fault tolerant communication scheme between repair controllers in the Dual-FPGA architecture has been developed.

For demonstration of feasibility, a robot control algorithm has been implemented on reconfigurable hardware as a case study. Results obtained from hardware emulation on the WILDFORCE platform have demonstrated that the performance of running the algorithm on reconfigurable hardware is comparable with that on a general-purpose processor. The ability of implementing systems with different precision to customize the design has demonstrated the advantage of an adaptive computing system. The controller is implemented with different fault tolerance schemes. Fault-injection experiments show that fault tolerance can be achieved for the robot controller constructed on the reconfigurable platform.

For transient recovery in real-time systems, a new roll-forward recovery scheme that eliminates the need of recomputation has been presented. Reliability analysis of a case study has shown that reliability of a TMR design is significantly improved by adding the recovery scheme. The scheme is applied to two forms of TMR: hardware redundancy and multi-threading. Both hardware redundancy and multi-threading have shown the reliability improvement. The recovery mechanism introduces small hardware overhead, and does not need re-computation for single failure recovery. Therefore, it is very suitable for real-time applications.

For permanent fault repair, a new permanent fault repair scheme for FPGA-based computing systems has been presented. This repair scheme can repair FPGA-based systems from permanent faults even when no fault-free elements are available for conventional permanent fault removal schemes. We have examined three design candidates to adapt to limited FPGA area and to provide reliability and availability. These designs have shown to have an improvement on availability compared to the module removal approach. Hence, they are more suitable for real-time applications where availability is a critical issue. The scheme has been implemented in a robot controller as a case study. The results show that by properly partitioning a design, design area can be effectively reduced. Therefore it can be used when available FPGA area is reduced because of permanent faults.

For FPGA repair process, a fault tolerant communication scheme between repair controllers in the Dual-FPGA architecture has been presented. This communication scheme can repair FPGAs from single permanent faults in faulty wires between repair controllers. Unlike conventional schemes that use redundant buses, our scheme only requires a spare wire. Parity checks and time out are used for error detection, and a command bounce strategy is used for interconnect diagnosis. Repair is accomplished by reconfiguring both FPGAs to avoid using the faulty wire. Our repair scheme can be applied to repair for faulty wires in data communication between FPGAs.

Chapter 8

References

[Abramovici 90] Abramovici, M., M. A. Breuer, and A. D. Friedman, *Digital Systems Testing and Testable Design*, Revised Printing, IEEE Press, 1990.

[Abramovici 01] Abramovici, M. and C. E. Stroud, "BIST-based Test and Diagnosis of FPGA Logic Blocks," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, Vol. 9, No. 1, pp. 159-72, Feb. 2001.

[Actuality 01] Actuality Systems, Inc., http://www.actuality-systems.com, 2001.

[Adams 89] Adams, S. J., "Hardware Assisted Recovery from Transient Errors in Redundant Processing Systems," *FTCS 19th Digest of Papers. The 19th Int'l Symp. on Fault-Tolerant Computing*, pp. 512-9, Chicago, IL, USA, 21-23 June 1989.

[Alfke 98] Alfke, P. and R. Padovani, "Radiation Tolerance of High-Density FPGAs," http://www.xilinx.com/products/hirel_qml.htm, 1998.

[Altera 01] Altera Co., http://www.altera.com, 2001.

[AMCC 01] AMCC, http://www.amcc.com, 2001.

[Annapolis 99] Annapolis Micro Systems Inc., http://www.annapmicro.com, 1999.

[Bass 97] Bass, J. M., G. Latif-Shabgahi, and S. Bennett, "Experimental Comparison of Voting Algorithms in Cases of Disagreement," *Proceedings. 23rd Euromicro Conference: New Frontiers of Information Technology*, pp. 516-23, Budapest, Hungary, 1-4 Sept. 1997.

[Bartlett 78] Bartlett, J. F., Tandem Computers Inc., Cupertino, CA, "A Nonstop Operating System," *Proc. of the Eleventh Hawaii International Conf. on System Sciences*, pp. 103-17, Honolulu, HI, USA, 5-6 Jan. 1978.

[Brodrick 01] Brodrick, D., A. Dawood, N. Bergmann, and M. Wark, "Error Detection for Adaptive Computing Architectures in Spacecraft Applications," *Proc. 6th Australasian Computer Systems Architecture Conf.*, pp. 19-26, Gold Coast, Qld., Australia, 29-30 Jan. 2001.

[Carmichael 99] Carmichael, C., E. Fuller, P. Blain, and M. Caffrey, "SEU Mitigation Techniques for Virtex FPGAs in Space Applications," http://www.xilinx.com/products/hirel_qml.htm, 1999.

[Chan 92] Chan, J. C., "An Improved Technique for Circuit Board Interconnect Test," *IEEE Trans. on Inst. and Meas.*, Vol. 41, No. 5, pp. 692-698, 1992.

[Chan 00] Chan, S., H. Luong, W. Charlan, R. Fukuhara, E. Homberg, P. Jones, G. Pixler, "The Implementation of a COTS Based Fault Tolerant Avionics Bus Architecture," *IEEE Aerospace Conf. Proc*, Vol. 7, pp. 297-305, 2000.

[Chandy 72] Chandy, K. M. and C. V. Ramamoorthy "Rollback and Recovery Strategies for Computer Programs," *IEEE Trans. on Computers*, Vol. C-21, No. 6, pp. 546-56, June 1972.

[Cooling 97] Cooling, J. E. and P. Tweedale, "Task Scheduler Co-Processor for Hard Real-Time Systems," *Microprocessors and Microsystems*, Vol. 20, No. 9, pp. 553-66, May 1997.

[Conde 99] Conde, R. F., A. G. Darrin, F. C. Dumont, P. Luers, S. Jurczy, N. Bergmann, and A. Dawood, "Adaptive Instrument Module - A Reconfigurable Processor for Spacecraft Applications," http://www.xilinx.com/products/hirel_qml.htm, 1999

[Craig 89] Craig, J. J., *Introduction to Robots: Mechanics and Control*, Addison-Wesley Publishing Company, 2nd Edition, 1989.

[Culbertson 97] Culbertson, W. B., R. Amerson, R. J. Carter, P. Kuekes, and G. Snider, "Defect Tolerance on the Teramac Custom Computer," *Proc. of the 1997 IEEE Symp. on FPGA's for Custom Computing Machines*, pp. 140-147, Napa Valley, CA, USA, 16-18 April 1997.

[Das 99] Das, D. and N. A. Touba, "A Low Cost Approach for Detecting, Locating, and Avoiding Interconnect Faults in FPGA-based Reconfigurable Systems," *12th Int'l Conf. on VLSI Design*, pp. 266-9, Goa, India, 7-10 Jan. 1999.

[DEC 80] DEC, Intel, and Xerox, "The Ethernet: A Local Area Network Data Link Layer and Physical Layer Specifications," V. 1.0, 1980.

[Donohoe 99] Donohoe, G. W. and J. C. Lyke, "Adaptive Computing for Space," *42nd Midwest Symp. on Circuits and Systems*, Vol. 1, pp. 126-9, Las Cruces, NM, USA, 8-11 Aug. 1999.

[Dutt 99] Dutt, S., V. Shanmugavel, and S. Trimberger, "Efficient Incremental Rerouting for Fault Reconfiguration in Field Programmable Gate Arrays," *Digest of Technical Papers, IEEE/ACM Int. Conf. on Computer-Aided Design*, pp. 173-176, San Jose, CA, USA, 7-11 Nov. 1999.

[Emmert 97] Emmert, J. M., and D. Bhatia, "Partial Reconfiguration of FPGA Mapped Designs with Applications to Fault Tolerance and Yield Enhancement," *Proc. of International Workshop of Field-Programmable Logic*, pp. 141-150, London, UK, 1-3 Sept. 1997.

[Emmert 98] Emmert, J. M., and D. Bhatia, "Incremental Routing in FPGAs," *Proc. of 11th Annual IEEE Int. ASIC Conf.*, pp. 217-221, Rochester, NY, USA, 13-16 Sept. 1998.

[Emmert 00] Emmert, J. M., C.E. Stroud, B. Skaggs, and M. Abramovici, "Dynamic Fault Tolerance in FPGAs via Partial Reconfiguration," *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, pp. 165-174, Napa Valley, CA, USA, 17-19 April 2000.

[Fiore 98] Fiore, P. D., C. S. Myers, J. M. Smith, and E. K. Pauer, "Rapid Implementation of Mathematical and DSP Algorithms in Configurable Computing Devices," *Proc. of the SPIE – The Int'l Society for Optical Engineering, Conf. of Configurable Computing: Technology and Applications*, Vol. 3526, pp. 178-89, Boston, MA, USA, 2-3 Nov. 1998.

[Furutani 89] Furutani, K., K. Arimoto, H. Miyamoto, T. Kobayashi, K. Yasuda, K. Mashiko, "A Built-in Hamming Code ECC Circuit for DRAMs," *IEEE Journal of Solid-State Circuits*, V. 24, I. 1, pp. 50-56, Feb. 1989.

[Girard 00] Girard, M., "F/A-18A Testing of Flight Control System Reversion to Mechanical Backup," *IEEE Aerospace Conference Proceedings*, Vol. 2, pp.21-7, Big Sky, MT, USA, 18-25 March 2000.

[Hamilton 92] Hamilton, D. L., J. K. Bennett, and I. D. Walker, "Parallel Fault-Tolerant Robot Control," *Proceedings of SPIE – The Int. Society for Optical Engineering. Conf. of Cooperative Intelligent Robotics in Space III*, Vol. 1829, pp. 251-61, Boston, MA, USA, 16-18 Nov. 1992.

[Hamilton 94] Hamilton, D. L., M. L. Visinsky, J. K. Bennett, J. R. Cavallaro, and I. D. Walker, "Fault Tolerant Algorithms and Architectures for Robots", *Proceedings of MELECON '94 Mediterranean Electrotechnical Conference*, Vol. 3, pp. 1034-6, Antalya, Turkey, 12-14 April 1994.

[Hamilton 96a] Hamilton, D. L., J. R. Cavallaro, and I. D. Walker, "Risk and Fault Tolerance Analysis for Robots and Manufacturing," *Proceedings of 8th Mediterranean Electrotechnical conference on Industrial Applications in Power Systems, Computer Science and Telecommunications*, Vol. 1, pp. 250-5, Bari, Italy, 13-16 May 1996.

[Hamilton 96b] Hamilton, D. L., I. D. Walker, and J. K Bennett, "Fault Tolerance Versus Performance Metrics for Robot Systems", *Proceedings of IEEE International Conference on Robots and Automation*, Vol. 4, pp. 3073-80, Minneapolis, MN, USA, 22-28 April 1996.

[Hamilton 99] Hamilton, C., G. Gibson, S. Wijesuriya, and C. Stroud, "Enhanced BIST-Based Diagnosis of FPGAs via Boundary Scan Access," *IEEE VLSI Test Symp.*, pp. 413-8, Dana Point, CA, USA, 25-29 April 1999.

[Hansen 90] Hansen, P., "Chapter 7: Taking Advantage of Boundary-Scan in Loaded-Board Testing," *The Test Access Port and Boundary Scan Architecture*, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[Hauck 98] Hauck, S., "The Roles of FPGA's in Reprogrammable Systems," *Proceedings of The IEEE*, Vol. 86, No. 4, pp. 615-638, April 1998.

[Hiltunen 94] Hiltunen, M. A. and R. D. Schlichting, "A Model for Adaptive Fault-Tolerant Systems," *EDCC-1 First European Dependable Computing Conf.*, pp. 3-20, Berlin, Germany, 4-6 Oct. 1994.

[Hsu 96] Hsu; P.-C. and S.-J. Wang, "Testing and diagnosis of board interconnects in microprocessor-based systems," *Proceedings of the Fifth Asian Test Symposium*, pp. 56-61, 1996.

[Huang 00a] Huang, W.-J., N.R. Saxena, and E.J. McCluskey, "A Reliable LZ Data Compressor on Reconfigurable Coprocessors", *FCCM'00 Symposium on Field-Programmable Custom Computing Machines*, pp. 249-258, Napa Valley, CA, USA, 16-19 April 2000.

[Huang 00b] Huang, W.-J., and E. J. McCluskey, "Transient Errors and Rollback Recovery in LZ Compression," *Proc. 2000 Pacific Rim International Symposium on Dependable Computing*, pp. 128-135, Los Angeles, CA, USA, 18-20 Dec. 2000.

[Huang 01a] Huang, W.-J. and E. J. McCluskey, "A Memory Coherence Technique for Online Transient Error Recovery of FPGA Configurations," *9th ACM Int. Symp. on Field-Programmable Gate Arrays (FPGA'01)*, pp. 183-192, Monterey, CA, USA, 11-13 Feb. 2001.

[Huang 01b] Huang, W.-J. and E. J. McCluskey, "Column-Based Precompiled Configuration Techniques for FPGA Fault Tolerance," to appear in *FCCM'01 Symposium on Field-Programmable Custom Computing Machines*, 2001.

[Huang 01c] Huang, W.-J., S. Mitra, and E. J. McCluskey, "Fast Run-Time Fault Location in Dependable FPGAs," *Proc. IEEE Defect and Fault Tolerance*, pp. 191-199, 2001.

[Hunt 87] Hunt, D. B., and P. N. Marinos, "A General Purpose Cache-Aided Rollback Error Recovery (CARER) Technique," *Proc. Int'l Symposium on Fault-Tolerant Computing*, pp. 170-175, Pittsburgh, PA, USA, 6-8 July 1987.

[Khatib 95] Khatib, O., K. Yokoi, K. Chang, D. Ruspini, R. Holmberg, A. Casal, and A. Baader, "Force Strategies for Cooperative Tasks in Multiple Mobile Manipulation Systems", *Robots Research 7, The Seventh International Symposium*, G. Giralt and G. Hirzinger, eds., pp. 333-342, Springer 1996.

[Kim 96] Kim, J. H., J. K. Lee, H. S. Eom, and J. W. Choe, "An Underwater Mobile Robot System for Reactor Vessel Inspection in Nuclear Power Plants," *Proc. of the Sixth International Symposium on Robots and Manufacturing, Recent Trends in Research, Education and Applications*, pp. 351-6, Montpellier, France, 28-30 May 1996.

[Kocsis 92] Kocsis, F., K. Ort, and J. F. Bohme, "ASIC Solution for Real-Time Inverse Kinematic and Jacobian Computations," *Proc. of Sixth European Signal Processing Conference*, Vol. 3, pp. 1545-8, Brussels, Belgium, 24-27 Aug. 1992.

[Kwiat 97] Kwait, K., "Performance Evaluation of a Dynamically Reconfigurable Multiprocessing and Fault-Tolerant Computing Architecture," *1997 Summer Computer Simulation Conf.*, pp. 91-6, Arlington, VA, USA, 13-17 July 1997.

[Lach 99] Lach, J., W. H. Mangione-Smith, and M. Potkonjak, "Algorithms for Efficient Runtime Faulty Recovery on Diverse FPGA Architectures", *Proc. Int'l Symp. on Defect and Fault Tolerance in VLSI Systems*, pp. 386-394, Albuquerque, NM, USA, 1-3 Nov. 1999.

[Lach 00] Lach, J., W. H. Mangione-Smith, and M. Potkonjak, "Enhanced FPGA Reliability through Efficient Run-Time Fault Reconfiguration", *IEEE Trans. on Reliability*, Vol. 49, No. 3, pp. 296-304, Sept. 2000.

[Lala 99] Lala, P. K., A. Singh, and A. Walker, "A CMOS-Based Logic Cell for the Implementation of Self-Checking FPGAs," *Proc. IEEE Int'l Symp. on Defect and Fault Tolerance in VLSI Systems*, pp. 238-46, Albuquerque, NM, USA, 1-3 Nov. 1999.

[Laplante 97] P. A. Laplante, *Real-Time Systems Design and Analysis –An Engineer's Handbook*, 2nd Ed., IEEE Press, 1997.

[Li 99] Li, Y., D. Li, and Z. Wang, "A New Approach to Detect-Mitigate-Correct Radiation-Induced Faults for SRAM-Based FPGAs in Aerospace Applications," *Proc. of IEEE National Aerospace and Electronics Conf.*, pp. 588-94, Dayton, OH, USA, 10-12 Oct. 2000.

[Lin 94] Lin, M.-B. and A. Y. Oruc, "A Fault-Tolerant Permutation Network Modulo Arithmetic Processor," *IEEE Trans. on VLSI Systems*, Vol. 2, No. 3, Sept. 1994.

[Lombardi 96] Lombardi, F., D. Ashen, X. T. Chen, and W. K. Huang, "Diagnosing Programmable Interconnect Systems for FPGAs," *Proc. ACM/SIGDA Int'l Symp. on field Programmable Gate Arrays*, pp. 100-6, Monterey, CA, USA, 11-13 Feb. 1996.

[Long 90] Long, J., W. K. Fuchs, and J. A. Abraham, "A Forward Recovery Using Checkpointing in Parallel Systems," *Proc. of the 1990 International Conf. on Parallel Processing*, Vol. 1, pp. 272-275, Chicago, IL, USA, 13-17 August 1990.

[Losq 76] Losq, J. "A Highly Efficient Redundancy Scheme: Self-Purging Redundancy," *IEEE Trans. Comput.*, C-25, No. 6, pp. 269-78, June 1976.

[Lu 99a] Lu S.-K. and C.-W. Wu, "A Novel Approach to Testing LUT-Based FPGAs," *Proc. of the 1999 IEEE Int'l Symp. on Circuits and Systems. VLSI*, Vol. 1, pp. 173-7, Orlando, FL, USA, 30 May-2 June 1999.

[Lu 99b] Lu, S.-K., J.-S. Shih, and Wu C.-W., "Built-In Self-Test and Fault Diagnosis for Lookup table FPGAs," *Proc. ISCAS IEEE Int'l Symp. on Circuits and Systems. Emerging Technologies for the 21st Century*, Vol. 1, pp. 80-3, Geneva, Switzerland, 28-31 May 2000.

[Mitra 99] Mitra, S., N.R. Saxena, and E.J. McCluskey, "A Design Diversity Metric and Reliability Analysis for Redundant Systems," *Proc. 1999 Int. Test Conf.*, pp. 662-671, Atlantic City, NJ, USA, 28-30 Sep. 1999.

[Mitra 00a] Mitra, S., W.-J. Huang, N.R. Saxena, S.-Y. Yu, and E.J. McCluskey, "Dependable Adaptive Computing Systems, The Stanford CRC ROAR Project," *2000 Pacific Rim International Symposium on Dependable Computing Fast Abstracts (PRDC 2000)*, pp. 15-16, Los Angeles, CA, USA, 18-20 Dec. 2000.

[Mitra 00b] Mitra, S. and E.J. McCluskey, "Which Concurrent Error Detection Scheme to Choose?" *Proc. 2000 Int. Test Conf.*, pp. 985-994, Atlantic City, NJ, 3-5 Oct. 2000.

[Mitra 00c] Mitra, S. and E.J. McCluskey, "Word-Voter: A New Voter Design for Triple Modular Redundant Systems," *18th IEEE VLSI Test Symposium*, pp. 465-70, Montreal, Canada, 30 Apr.-4 May, 2000.

[Moorman 99] Moorman, A. C., D. M. Jr. Cates, "A Complete Development Environment for Image Processing Applications on Adaptive Computing Systems," *Proc. of IEEE Int'l Conf. on Acoustics, Speech, and Signal Processing*, Vol. 4, pp. 2159-2162, Phoenix, AZ, USA, 15-19 March 1999.

[Motorola 01] Motorola, Inc., http://www.motorola.com, 2001.

[Nett 97] Nett, E. and M. Mock, "A Recovery Model for Extended Real-Time Transactions," *Proc. 1997 High-Assurance Engineering Workshop*, pp. 124-7, Washington, DC, USA, 11-12 Aug 1997.

[Ohlsson 98] Ohlsson, M., P. Dyreklev, K. Johansson, and P. Alfke, "Neutron Single Event Upsets in SRAM-Based FPGAs," *IEEE Radiation Effects Data Workshop*, pp. 177-80, Newport Beach, CA, USA, 24 July 1998.

[Peterson 00] Peterson, L. L. and B. S. Davie, *Computer Networks – A Systems Approach*, 2nd Ed, Morgan Kaufmann, 2000.

[Pradhan 92] Pradhan, D. K., and N. Vaidya, "Roll-Forward Checkpointing Scheme: Concurrent Retry with Nondedicated Spares," *IEEE Workshop on Fault Tolerant Parallel and Distributed Systems*, pp. 166-74, Amherst, MA, USA, 6-7 July1992.

[Psomoulis 99] Psomoulis, A. M., N. Cazajus, D. S. Dandouras, H. Barthe, M. Gangloff, and E. T. Sarris, "Development of an Innovative, Two-Processor Data Processing Unit for the Magnetospheric Imaging Instrument onboard the Cassini Mission to Saturn. I. Hardware Architecture," *IEEE Trans. on Geoscience and Remote Sensing*, Vol. 37, No. 4, pp. 1980-96, July 1999.

[Quickturn 01] Quickturn, A Cadence Company, http://www.quickturn.com, 2001.

[Reynolds 01] Reynolds, R. O., P. H. Smith, L. S. Bell, and H. U. Keller, "The Design of Mars Lander Camaras for Mars Pathfinder, Mars Surveyor'98 and Mars Surveyor'01", *IEEE Trans. On Instrumentation and Measurement*, Vol. 50, No. 1, pp. 63-71, Feb. 2001.

[Renovell 97] Renovell, M., J. Figueras, and Y. Zorian, "Test of RAM-Based FPGA: Methodology and Application to the Interconnect," *Proc. IEEE VLST Test Symposium*, pp. 230-7, Monterey, CA, USA, 27 April-1 May 1997.

[Roy 95] Roy, K. and S. Nag, "On Routability for FPGAs under Faulty Conditions," *IEEE Trans. on Computers*, Vol. C-44, No. 11, pp. 1296-1305, Nov. 1995.

[Rupp 98] Rupp, C. R., M. Landguth, T. Garverick, E. Gomersall, H. Hold, J. M. Arnold, and M. Gokhale, "The NAPA Adaptive Processing Architecture," *IEEE Symp. on FPGAs for Custom Computing Machines*, pp. 28-37, Napa Valley, CA, USA, 15-17 April 1998.

[Saxena 98] Saxena, N. R. and E.J. McCluskey, "Dependable Adaptive Computing Systems," *IEEE Systems, Man, and Cybernetics Conf.*, pp. 2172-2177, San Diego, CA, USA, Oct. 11-14, 1998.

[Saxena 00] Saxena, N. R., S. Fernandez-Gomez, W.-J. Huang, S. Mitra, S.-Y. Yu, and E. J. McCluskey, "Dependable Computing and Online Testing in Adaptive and Configurable Systems," *IEEE Design and Test*, Vol. 17, No. 1, pp. 29-41, Jan.-March 2000.

[Schmalz 98] Schmalz, M .S., E. X. Ritter, and F. H. Caimi, "Mathematical Methods for Mapping Image and Data Compression Transforms to Adaptive Computing Systems," *OCEANS '98 Conf. Proc.*, Vol. 3, pp. 1629-1633, Nice, France, 28 Sept.-1 Oct. 1998.

[Shirvani 01] Shirvani, P. P, *Fault Tolerant Computing for Radiation Environments*, PhD Dissertation, Stanford University, 2001.

[Siewiorek 73] Siewiorek, D. P. and E. J. McCluskey, "Switch Complexity in Systems with Hybrid Redundancy," *IEEE Trans. Comp.*, C-22, pp. 276-282, March 1973.

[Siewiorek 00] Siewiorek, D. P. and R. S. Swarz, *Reliable Computer Systems – Design and Evaluation*, 3rd Ed, Digital Press, 2000.

[Smith 78] Smith, B. J., "A Pipeline, Shared Resource MIMD Computer", *Proc. Intl. Conf. on Parallel Processing*, pp. 6-8, Bellaire, MI, USA, 22-25 Aug. 1978.

[Spainhower 9] Spainhower, L. and t. A. Gregg, "S/390 Parallel Enterprise Server G5 Fault Tolerance," *IBM Journal of Research Development*, Vol. 43, pp. 863-873, Sept.-Nov. 1999.

[Sparmann 96] Sparmann, U., "On the Effectiveness of Residue Code Checking for Parallel Two's Complement Multipliers," *IEEE Trans. on VLSI Systems*, Vol. 4, No. 2, pp. 227-39, June 1996.

[Tang 69] Tang, D. T. and R. T. Chien, "Coding for Error Control," *IBM Sys. J. 8*, No. 1, pp. 48-85, 1969.

[Thayer 76] Thayer, W. J., "Redundant Electrohydraulic Servoactuators," *Technical Bulletin 127*, MOOG INC. Controls Division, 1976.

[Thorton 64] Thorton, J. E., "Parallel Operation in the Control Data 660", *AFIPS Conf. Proc., Fall Joint Computer Conf.*, Vol. 26, pp. 33-40, 1964.

[Ting 94] Ting, Y., S. Tosunoglu, and B. Fernandez, "Control Algorithms for Fault-Tolerant Robots," *Proceedings of 1994 IEEE International Conference on Robots and Automation*, Vol. 2, pp. 910-15, San Diego, CA, USA, 8-13 May 1994.

[Tosunoglu 93] Tosunoglu, S., "Fault Tolerance for Modular Robots," *Proceedings of IECON – 19th Annual Conference of IEEE Industrial Electronics*, Vol. 3, pp. 1910-14, Maui, HI, USA, 15-19 Nov. 1993.

[Toye 90] Toye, G., *Management of Non-homogeneous Functional Modular Redundancy for Fault Tolerant Programmable Electro-mechanical Systems*, Ph.D. Dissertation, Stanford University 1990.

[Toye 94] Toye, G. and L. J. Leifer, "Helenic Fault Tolerance for Robots," *Computers & Electrical Engineering*, Vol. 20, No. 6, pp. 479-97, Nov. 1994.

[Turver 93] Turver, K. D, "Batch Processing of Flight Test Data," *International Telemetering Conference (Telemetering – Yesterday, Today, and Tomorrow)*, pp. 815-9, Las Vegas, NV, USA, 25-28 Oct. 1993.

[Visinsky 95] Visinsky, M. L., J. R. Cavallaro, and I. D. Walker, "A Dynamic Fault Tolerance Framework for Remote Robots", *IEEE Transactions on Robots and Automation*, Vol. 11, No. 4, pp. 477-490, August 1995.

[Wagner 87] Wagner, P. T., "Interconnect Testing with Boundary Scan," *Int. Test. Conf.*, pp. 52-57, 1987.

[Walters 98] Walters, A. and P. Athanas, "A Scaleable FIR Filter using 32-bit Floating-point Complex Arithmetic on a Configurable Computing Machine," *IEEE Symp. on FPGAs for Custom Computing Machines*, pp. 333-4, Napa Valley, CA, USA, 15-17 April 1998.

[Wang 89] Wang, L.-T., M. Marhoefer, and E. J. McCluskey, "A Self-Test and Self-Diagnosis Architecture for Boards Using Boundary Scans," *Int. Test Conf.*, 1989.

[Wang 99] Wang, J. J., R. B. Katz, J. S. Sun, B. E. Cronquist, J. L. McCollum, T. M. Speers, and W. C. Plants, "SRAM Based Re-programmable FPGA for Space Application," *IEEE Trans. on Nuclear Science*, Vol. 46, No. 6, Pt. 1, pp. 1728-35, Dec. 1999.

[Wicker 95] Wicker, S. B., *Error Control Systems for Digital Communication and Storage*, Prentice-Hall, Inc., 1995.

[WindRiver 01] Wind River Systems, Inc., http://www.windriver.com, 2001.

[Wu 93] Wu, E. C., J. C. Hwang, and J. T. Chladek, "Fault Tolerant Joint Development for the Space Shuttle Remote Manipulator System: Analysis and Experiment," *IEEE Trans. on Robots and Manufacturing—Recent Trends in Research, Education, and Applications*, Vol. 9, No. 5, pp. 675-84, Oct. 1993.

[Xilinx 00a] Xilinx Application Note 181, "SEU mitigation Design Techniques for XQR4000XL," http://www.xilinx.com/xapp/xapp181.pdf, March 28, 2000.

[Xilinx 00b] Xilinx Application Note 216, "Correcting Single-Event Upsets Through Virtex Partial Configuration," http://www.xilinx.com/xapp/xapp216.pdf, June 1, 2000.

[Xilinx 01a] Xilinx, Inc., http://www.xilinx.com, 2001.

[Xilinx 01b] Xilinx Virtex Datasheet, http://www.xilinx.com/apps/virtexapp.htm#databook, 2001.

[Xu 96] Xu, J. and B. Randell, "Roll-Forward Error Recovery in Embedded Real-Time Systems," *Proc. of 1996 Int'l Conf. On Parallel and Distributed systems*, pp. 414-21, Tokyo, Japan, 3-6 June 1996.

[Yu 00] Yu, S.-Y., N. Saxena, and E. J. McCluskey, "An ACS Robot Control Algorithm with Fault Tolerant Capabilities," *IEEE Symp. on Field-Programmable Custom Computing Machines*, pp. 175-84, Napa Valley, CA, USA, 16-19 April 2000.

[Yu 01] Yu, S.-Y. and E. J. McCluskey, "On-line Testing and Recovery in TMR Systems for Real-Time Applications," *Proc. 2001 Int'l Test Conf.*, pp. 240-249, 2001.

[Zeng 99] Zeng, C., N. R. Saxena, and e. J. McCluskey, "Finite State Machine Synthesis with Concurrent Error Detection," *Proc. 1999 Int'l Test Conf.*, pp. 672-680, Atlantic City, NJ, USA, 28-30 Sept. 1999.

Appendix A

An ACS Robot Control Algorithm with Fault Tolerant Capabilities

(An extended version of "An ACS Robot Control Algorithm with Fault Tolerant Capabilities," by Shu-Yi Yu, Nirmal Saxena, and Edward J. McCluskey, that appeared in *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 175-184, 2000.)

An ACS Robot Control Algorithm with Fault Tolerant Capabilities

Abstract

This paper demonstrates that an adaptive computing system (ACS) is a good platform for implementing robot control algorithms. We show that an ACS can be used to provide both good performance and high dependability. An example of an FPGA-implemented dependable control algorithm is presented. The flexibility of ACS is exploited by choosing the best precision for our application. This reduces the amount of required hardware and improves performance. Results obtained from a WILDFORCE emulation platform showed that even using 0.35μm technology, an FPGA-implemented control algorithm has comparable performance with the software-implemented control algorithm in a 0.25μm microprocessor. Different voting schemes are used in conjunction with multi-threading and combinational redundancy to add fault tolerance to the robot controller. Error-injection experiments demonstrate that robot control algorithms with fault tolerance techniques are orders of magnitude less vulnerable to faults compared to algorithms without any fault tolerant features.

1. Introduction

Adaptive Computing Systems (ACS) augment the traditional Von Neumann processor-memory computation model by a fabric of reconfigurable hardware. One model of the ACS architecture consists of CPU, I/O, memory, and reconfigurable coprocessors. For example, a compiler [Gokhale 98] can partition and map applications into parts that run well on traditional fixed instruction processors and parts that run well on the reconfigurable coprocessor. Commercial implementations of the ACS model range from a single chip [Gokhale 98], a multi-chip board [Annapolis 99][TelSys 99][Xilinx 99], or a multiple-board system [Quickturn 99]. Figure 1 illustrates an example of an ACS using a Xilinx Virtex series FPGA as a reconfigurable component.

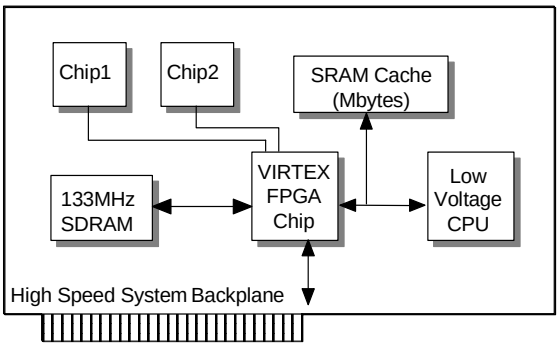


Figure 1. A Xilinx Virtex chip as a system component- an ACS implementation [Xilinx 99].

Applications that are well suited for ACS are: digital signal processing, image processing, control, bit manipulation, compression, and encryption [Saxena 00]. Most of the research and development in the ACS area has been dictated by the performance requirements of target applications. Performance is an important requirement; however, the use of ACS applications in areas like nuclear reactors, fly-by-wire systems, remote (space station) locations, and life-critical systems makes dependability also an important requirement. An advantage of using ACS for dependable computing is that by exploiting the ability of reconfiguration, systems can be repaired when faults are present. With very few exceptions [Culbertson 97][Lach 98], little work has been done in the dependable ACS area. This research is part of the ROAR project [Saxena 98], whose objective is to provide dependable computing solutions in an ACS framework. The main thrust of this paper is the instrumentation of robot control algorithms in ACS environments.

Robots are frequently used under hazardous or remote circumstances, such as in nuclear power plants [Kim 96] or in spacecraft [Wu 93]. Maintenance and repair is usually very expensive and time-consuming for these applications.

Much research has been done seeking ways of providing an effective fault tolerant robot system. For example, different control schemes are presented in [Ting 94][Yang 98]. Redundancy and voting in joint and actuator levels are suggested in [Tosunoglu 93] [Toye 90] [Thayer 76]. [Hamilton 92] and [Hamilton 94] use multiple processors for control purpose. These processors can be either used for parallel control that can speed up computation of control algorithms, or used as redundant processors for fault tolerance. [Visinsky 95] presents a fault tolerant framework for robots. In the framework, the controller contains special control algorithms that are used when faults are present. [Toye

94] presents different voting methods for robot applications. The voting schemes includes a dynamic democratic voting scheme which takes error history into consideration, and another scheme which uses force compensation to vote among actuators. Metrics to evaluate risk, cost, fault tolerance, performance, and benefit in robot systems have also been investigated [Hamilton 96a] [Hamilton 96b].

With the exception of [Hamilton 92] and [Hamilton 94], most of the cited research work concentrates in either the control or the mechanical field. Not much work has been done in fault tolerance of the electronic hardware that instruments control algorithms. A conventional way of controlling a robot is to run the control algorithms on general-purpose computers. Figure 2, for example, shows a mobile Puma 560 Manipulator [Khatib 95], in which a Pentium CPU is used to compute the force strategies. In this paper, we emulate the control algorithm on FPGAs that are used in ACS frameworks [Walters 98] to show the feasibility of ACS in the robot area in terms of high dependability and performance.

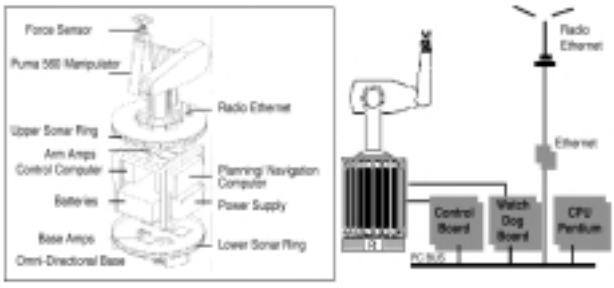


Figure 2. Puma 560 manipulator used in SAMM project [Khatib 95].

In the following sections, we present a fault tolerant control algorithm in ACS environment. Section 2 introduces the robot control system. Section 3 presents the fault tolerance techniques implemented in the control system. Section 4 describes the implementation of the system in both general-purpose processors and reconfigurable hardware. Results of emulation of the control system are also shown in this section. Section 5 describes the comparison of fault tolerance techniques and experimental results. Section 6 concludes this paper.

2. Control System

Figure 3 shows a block diagram of a general robot system [Craig 89]. It consists of a robot unit and a control unit. The robot unit represents the mechanical part, and the control unit represents the electronic part. The interface between the robot unit and the control unit includes sensors and A/D, D/A converters. The sensors sample the current position of the mechanical part and transform it to an analog signal (voltage or current). The A/D converter converts the signal and sends it to the control unit. The desired trajectory and other control parameters are supplied by the user to the control unit. The control parameters describe the characteristics of the robot unit. The control unit collects the current position of the robot and the desired position, and calculates the needed force. The calculated value is converted to an analog signal, and that signal drives the motor to move the robot.

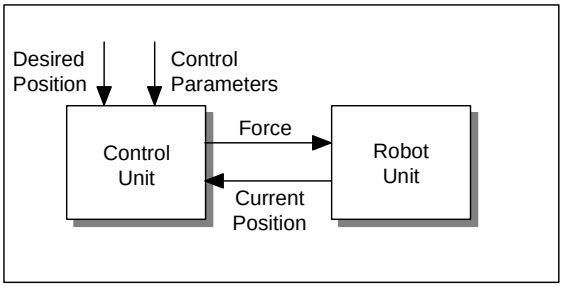


Figure 3. Block diagram of a robot system.

The robot can be modeled as a second-order linear mechanical system. We apply a second-order linear control algorithm to control the robot. A detailed description of the system and the algorithm is in the appendix. The designs are emulated and implemented in ACS test-beds. These test-beds are described in Sec. 4.1.

This paper focuses on the control unit. In the design of the control unit, we use two approaches to implement fault tolerance techniques. In the first approach, the algorithm is implemented as a fully combinational circuit. In the second approach, the algorithm is implemented in multiple stages and hardware is reused. A detailed description of the control unit is in Sec 4.1.

The performance of the controller affects the performance of the whole robot system. The rate of producing a corresponding control force is called the servo rate [Craig 89]. The servo rate needs to be fast enough to keep track of the inputs, to reject disturbance from the outside world, to prevent aliasing, and to suppress the natural

resonance of the mechanical part. In this paper, the performance achieved by different implementations of the control unit is examined and compared.

The controller needs to be designed with proper precision to meet the requirement of the robot application and to accommodate the characteristics of the robot. The tolerable steady-state error differs among applications, and the resolution and accuracy vary among robots. Hence the needed precision of the controller varies among applications. There is a trade-off among precision, performance, and hardware overhead. To implement the control algorithm, we need to choose the precision that meets all the application requirements while taking less hardware overhead and giving the highest performance. An ACS provides the flexibility of varying precision. The design can be optimized by choosing the precision of the system. Section 4.1 discusses experiments on different precision.

3. Fault Tolerance Techniques

Because safety is important in robot applications, fault tolerance techniques are needed to improve the dependability of the control system. To determine suitable fault tolerance techniques for our ACS robot application, two techniques were implemented and compared. One is *multi-threading*, and the other is *combinational redundancy*.

The concept of using multi-threading has been described in [Thorton 64] and [Smith 78]. The idea of using multi-threading for fault tolerance was described in [Saxena 98]. This is illustrated in Fig. 4. Figure 4 (a) shows an algorithm executed in multiple stages in limited hardware. Some resources are under-utilized (idle) due to data dependencies and memory latency. By scheduling multiple independent threads most of the idle resources can be reclaimed. To obtain fault tolerance, multiple copies of the same algorithm are executed as multiple threads in the hardware. The final output is obtained by voting among the results from different threads. As shown in Fig. 4 (b), the replicated instructions of the redundant thread can be inserted into the idle stages to minimize timing overhead. In a fault-free scenario, identical outputs are produced from all threads. When faults are present, the redundant threads provide a means for fault detection and fault masking.

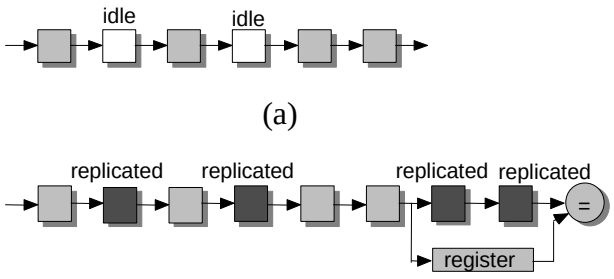


Figure 4. Multi-threading:(a) an algorithm in multiple stages, and (b) multi-threading.

In combinational redundancy, the robot controller is implemented in combinational hardware, and the hardware is replicated into multiple copies. The input is connected to all of the modules. In case of two copies (duplex), a comparator is used to detect errors. For three or more copies, a voter is used to determine the final output among the outputs from the modules. The replicated copies provide fault detection and possibly fault masking capabilities.

Possible voting schemes used in conjunction with fault tolerance techniques include exact (bit-by-bit) majority voting, median voting, weighted average voting, plurality voting, and others. In an NMR (N-Modular Redundancy) system, these voting schemes are indistinguishable when there is only one faulty module. These different voting schemes behave differently in the presence of multiple-module failures. Comparison among various voting schemes is reported in [Bass 97]. In this paper, we implement two voters in our design and compare their behavior. One is the majority voter, which is widely used, and the other is the median voter, which performs the best in Bass' paper. The majority voter selects the bit-wise output agreed by the majority among the modules. The median voter selects the middle value of the outputs from different modules.

Error detection mechanism is added in conjunction with voters to provide error-detecting ability. Figure 5 illustrates a voter with error detection. The inputs are compared with the voted output, and if it is different, error signals will be generated. This technique has been used in previous literature such as hybrid redundancy [Siewiorek 73] to remove a faulty module from the system. Comparators are also used in the word voter [Mitra 00c] for disagreement detection.

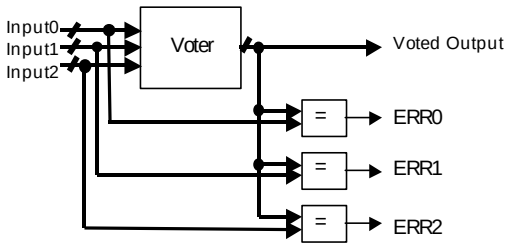


Figure 5. A voter with error detection mechanism.

To compare fault tolerance techniques, we considered dependability, and overhead in terms of area and performance degradation. The area and performance are examined by implementing the system in reconfigurable hardware. The experiments are described in Sec 4. The dependability of the systems is examined through fault injection experiments that yield data on numbers of detected faults and corruption. Fault injection experiments are described in Sec. 5.

4. Emulation of the Control Algorithm

The feasibility of implementing robot algorithms in ACS environment is demonstrated by the emulation of a second-order linear control algorithm in ACS hardware, Quickturn™ System Realizer and WILDFORCE® board. For the purposes of comparison, the algorithm was also implemented in C programs running on general-purpose processors. To investigate the area overhead and the performance degradation of designs with fault tolerant features, different fault tolerance techniques are implemented in the control system for a comparative evaluation. In addition, the controllers are designed using different bit width for different precisions, and an optimum precision is chosen for final implementation.

Section 4.1 describes control systems in ACS hardware. Section 4.2 explains the implementation of the control algorithm in general-purpose processors and shows performance data. Section 4.3 discusses our results.

4.1 Robot Control System in ACS

The controller was initially emulated in the Quickturn's System Realizer to verify its functionality. To get more accurate performance and area data, it was implemented on the WILDFORCE board, which can be used ACS architectures (described in Section 1).

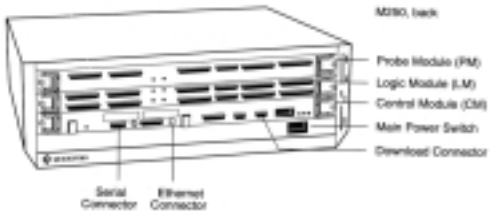
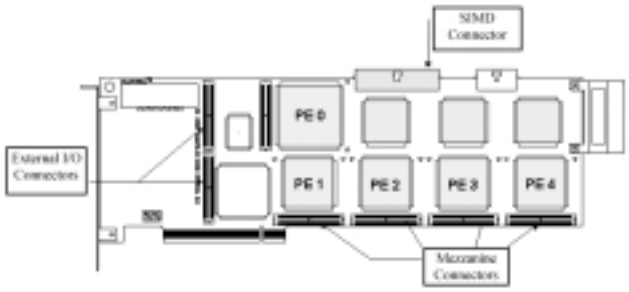
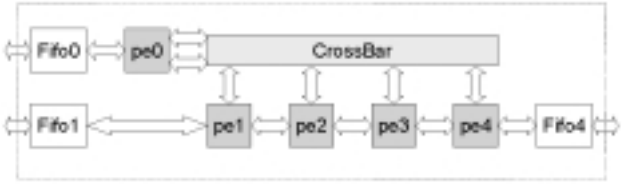


Figure 6. Quickturn's M250 System Realizer [Quickturn 99]



(a)



(b)

Figure 7. The WILDFORCE board [Annapolis 99]: (a) a layout, and (b) a block diagram.

Quickturn's M250 System Realizer is an emulator from Quickturn Design Systems. The System Realizer comprises one Logic Module (LM), one Probe Module (PM), and one Control Module (CM). A LM contains Quickturn MUX chips and 80 Xilinx LCA4013 FPGAs. In each FPGA, up to 13k logic gates can be implemented. The LM performs the logic function of the design. The PM contains an integrated logic

analyzer and stimulus generator. The PM applies test vectors to the design and records output responses. The CM contains an embedded CPU and hard disk drives. It controls the System Realizer and provides network connectivity. Additional Target Interface Modules (TIMs) can be used for the connection of emulation engines and the target system for in-circuit emulation. The I/O cards in TIMs can be replaced with other devices, such as embedded memory or analog circuits.

The WILDFORCE board is a PCI-based custom reconfigurable multi-chip system. A layout and a block diagram of the WILDFORCE board are shown in Fig. 7. The system has five processing elements: one Control/Processing Element (CPE) and four Processing Elements (PE1 through PE4.) The PEs consist of Xilinx 4036XLA FPGAs, each of which can realize up to 36k logic gates. Each PE has a 1Mb of local memory. All of the local memory is accessible to the host processor through the PCI bus. A crossbar is used to provide communication among PEs on the board. Each PE has one 36-bit connection to the crossbar, and CPE has two. The crossbar is composed of three Xilinx XC4006E FPGA devices. The PE array connects to the PCI bus through a set of bi-directional synchronous FIFOs. The maximum frequency of FIFO access is 50MHz. The board is capable of generating PE clock rates ranging from 300kHz to 55MHz.

In the control unit design, we use two architectures to implement fault tolerance techniques. Operation of the control algorithm consists of addition and multiplications. To implement multi-threading, the computation is partitioned into multiple stages, and the arithmetic computation resources, adders and multipliers, are reused. The computation is replicated as different threads. The threads share the same hardware resources. To implement combinational redundancy, the control unit is designed as a combinational circuit and replicated.

To select a suitable precision for our control system, several controllers were designed using different precisions. The robot characteristics, such as the response time, the steady-state error, and the time to stabilization were used as evaluation criteria. Among all controllers that met the application requirements, the lowest-precision design was selected to minimize the hardware overhead and to optimize the performance.

Table 3-1 lists the simulated response data of systems with different precision. We use *steady-state error*, *time to stabilization*, and *response time* to evaluate the

controller. *Steady-state error* is the difference between the desired position and the robot position when the robot finally rests at a stable position after stimulation. *Time to stabilization* is the time needed for a robot to reach its final stable position. *Response time* is the time needed to move a robot 90% to its desired position. The precision used was 32, 24, 16, and 12 bits. With the given test trajectory, the systems work well in all precisions except 12 bits. In the 12-bit system, arithmetic overflow occurs resulting in overshoot of the output waveform. As shown in Table 3-1, the response time and the time to stabilization of systems with 32, 24, and 16-bit numbers are almost the same. The major difference among the systems is accuracy: a higher-precision system has less steady-state error. Therefore, in terms of the response of the system, as long as the requested data range does not cause arithmetic overflow in the system and the steady state error is tolerable, a low-precision system is sufficient for the application while taking less area and shorter execution time than a higher-precision design. From the simulation results, the 16-bit system was the implemented choice on the WILDFORCE board.

Table 1. Simulated square wave response for systems with various precisions.

Number of bits	32	24	16	12
Steady state error(%)	0.006	0.1	2	---
Time to stabilization (<0.1%) (cycles)	135	133	145	
Response time (cycles)	1	1	1	

The designs emulated on Quickturn's emulator were written in Verilog RTL code. The code was synthesized and compiled by Quest software. The compiled data are downloaded to the emulator. Test vectors were applied to the emulated circuit to verify the designs. Table 2 lists the designs emulated on the Quickturn's System Realizer. The designs use multi-threading as the fault tolerance technique. For the three-thread design, both 32-bit and 16-bit numbers are implemented. Emulation verified that these designs have desired function.

Table 2. Emulation of the multi-threaded designs on Quickturn's System Realizer.

Threads	1	2	3	
Number of bits	32	32	32	16
Number of FPGAs	29	29	103[1]	20
# of Stages	7	9	11	11

The designs implemented on the WILDFORCE board were written in VHDL code. The code was synthesized with Synplify®. The placement and routing was done with the software provided by Xilinx. Frequencies are obtained from emulation.

Table 3. Emulation results on WILDFORCE board – combinational redundancy.

# of Modules	1	2	3	3
Voter Type	---	---	Med	Maj
LUTs (total 3888)	404	819	1284	1257
Registers (total 2592)	115	230	345	349
Frequency (MHz)	24	26	22	24
Latency (μs)	0.042	0.038	0.045	0.042

The control system is implemented in two PEs. The control unit is in PE1, and the robot unit is in PE2. The two units are separated in two chips for fault injection purposes. The two units communicate through the bus between PE1 and PE2. Input vectors are applied from the computer through FIFO1. Output values are transmitted to the host computer through FIFO4. Tables 3 and 4 show the emulation results on the WILDFORCE board. In the tables, the hardware area is represented by the number of Look Up Tables (LUTs) and registers used in FPGAs. In the Xilinx XC4000 series FPGAs, the logic function is provided by Configurable Logic Blocks (CLBs). A CLB consists of two four-input LUTs (F LUTs), one three-input LUT (H LUT), and two registers. The number of LUTs is the total number of mapped F LUTs and H LUTs. The

[1] This design was emulated in Quickturn M3000 System Realizer, which is larger than M250. Other designs shown on the table were emulated in Quickturn M250 System Realizer.

data is obtained from report files of a Xilinx placement-and-route tool. These results will be discussed in Sec. 4.3.

Table 4. Emulation results on WILDFORCE board – multi-threading.

# of Threads	1	2	3	3
Voter Type	--	---	Med	Maj
LUTs (total 3888)	409	814	1248	1251
Registers (total 2592)	362	636	945	945
# of Stages	7	8	11	11
Frequency (MHz)	50	50	50	50
Latency (μs)	0.14	0.16	0.22	0.22

4.2 Control Algorithm on General-Purpose Processors

We also implemented the control algorithm in C programs and ran them on Ultra Sparc II and Pentium II Xeon processors. Two data types are used: single precision floating-point numbers (32 bits), and fixed-point numbers (32 bits). Redundancy is achieved by software redundancy, that is, duplicating the code inside the program. Table 5 shows the latencies of the C programs on general-purpose processors. The latencies are roughly proportional to the number of copies. These results are compared with the performance of the ACS implementation in the next section.

Table 5: Performance of the control algorithm on general-purpose processors.

Latency (μs)				
# of Copies	Ultra SPARC II (300MHz)		Pentium II Xeon (450MHz)	
	Float	Fixed	Float	Fixed
1	0.186	0.044	3.866	0.033
2	0.288	0.078	7.741	0.074
3	0.569	0.122	11.689	0.105

4.3 Discussion

In this section, the results from the emulated designs on the WILDFORCE board are compared with the results obtained from general-purpose processors.

As shown in Table 3, the designs with combinational redundancy have latencies ranging from 0.038μs to 0.045μs. As shown in Table 4, all designs with multi-threading are running at the maximum frequency of the board, which is 50MHz. Table 6 lists the expected latency of multi-threaded design from synthesis results of Synplify®, and the latencies are much higher than those obtained by emulation. Higher emulation performance may be obtained in multi-threaded designs if the maximum board frequency can be improved. The latencies of multi-threaded designs from emulation range from 0.14μs to 0.22μs, and their expected latencies range from 0.075μs to 0.154μs. The designs with combinational redundancy have shorter latencies than the multi-threaded designs because they are fully combinational.

Table 6: Expected latency on WILDFORCE board – multi-threading.

# of Threads	1	2	3	3
Voter Type	--	---	Med	Maj
# of Stages	7	8	11	11
Frequency (MHz)	92.6	92.6	71.4	71.4
Latency (μs)	0.075	0.086	0.154	0.154

As the area results show, the area overhead is roughly proportional to the number of modules or threads. It does not differ much between different fault tolerance techniques or different voting schemes except in the number of registers. Multi-threaded designs require more registers because hardware is reused and intermediate results need to be stored. For combinational redundancy, results are proportional to number of redundant copies. However, in the designs with multi-threading, even when resources are reused, the area overhead is not significantly reduced. The reason is that in the implemented algorithm, the area of the reused hardware resources is small compared to the additional registers and the interconnection resources for those registers, which are proportional to the number of threads. In systems that use large and complicated

hardware to process data, multi-threading may take advantage of reusing resources and may reduce hardware overhead. In our case, the designs with combinational redundancy have better performance and less area overhead than those with multi-threading. However, for a more complicated control algorithm, combinational redundancy may not always be better than multi-threading. The reason is that a fully combinational circuit for a complicated algorithm can take much area and may not fit in an FPGA. To fit a large algorithm in an FPGA, resource reuse might be needed. In this case, multi-threading can be a better choice to provide redundancy for fault tolerance.

Figure 8 shows the performance for the one-copy (simplex) and three-copy (TMR or software redundancy) designs of the control algorithm in both general-purpose processors and reconfigurable hardware. The TMR latency for general-purpose processors are assumed to be the same as that of a simplex design because voting latency is generally small compared with total latency. The Pentium II Xeon processor is in 0.25μm technology, and both the Ultra Sparc II processor and Xilinx FPGA chips used in the WILDFORCE board are based on 0.35μm technology. The simplex results show that even when older generation FPGAs were used, the performance obtained in reconfigurable hardware is comparable to that of the later generation general-purpose microprocessors. For the TMR design in WILDFORCE board, the performance is not degraded because it takes advantage of parallelism in hardware. We could also use multiple general-purpose processors when implementing TMR to achieve similar performance as simplex designs. However, the cost of multiple processors would be fairly high comparing to a single FPGA. In our case, if we want to implement fault tolerance techniques into designs without much additional cost, a TMR design in a FPGA would be a better choice than a design with software redundancy in microprocessors. For a more complicated control algorithm, a FPGA can still be an appealing choice because parallism can be used to achieve high performance/small latency for a FPGA-based TMR design.

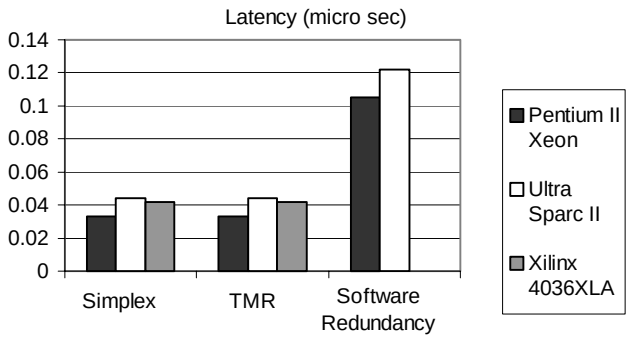


Figure 8. Performance of implementations on general-purpose processors and ACS.

5. Comparison of Fault Tolerance Techniques

Comparison is needed to determine the suitable fault tolerance technique for the implemented system in reconfigurable hardware. We consider dependability, area overhead, and performance degradation as the criteria to evaluate our implementation.

In order to evaluate dependability, faults were injected into the control unit of the system by modifying configuration bit-stream, and fault coverage was measured. A fairly extensive list of fault injection points in an FPGA is: configuration logic bits, CLOCK/RESET logic, output of flip-flops, output of LUTs, and primary inputs and outputs. Stuck-at faults at primary inputs, primary outputs, and CLOCK/RESET logic, will corrupt designs unless redundant I/O and CLOCK/RESET are used. The effect of stuck-at faults at flip-flop outputs can be approximated as stuck-at faults at output of LUTs, which are connected to flip-flop inputs. Therefore in this paper, we injected stuck-at-0 and stuck-at-1 faults at the output of LUTs to evaluate fault tolerance capabilities.

In each design of the control system, all possible single stuck-at faults are examined. To compare the effectiveness of the voters in the presence of multiple faults, double stuck-at faults are injected in designs with majority voters and median voters. For each examined design, five thousand stuck-at fault pairs are chosen randomly. Output vectors of fault-injected designs are compared with those of fault-free designs. An output is considered *corrupted* if its value is different from the output of fault-free systems. Results from comparators which compare voted results and each of the three outputs are used to indicate error. Tables 7 and 8 list the results of single stuck-at fault injection experiments. Table 9 lists the results of double stuck-at fault injection experiments. In the

tables, we define *corruption without detection* as the case when an output is corrupted but no detection signal is generated from comparators; *corruption with detection* is defined as the case when an output is corrupted and comparators detect the disagreement; *detection without corruption* means that comparators detect a disagreement, but the output is the same as the output from a fault-free design; and *no detection or corruption* is defined as the case when an output is the same as the fault-free output, and comparators do not detect any disagreement.

Table 7: Single stuck-at fault injection in designs with multi-threading.

# of Threads	1	2	3	3
Voter Type	---	---	Med	Maj
Injected Faults	936	1843	2803	2830
Corruption w/o Detection	806	18	39	7
Corruption w/ Detection	0	788	121	46
Detection w/o Corruption	0	785	2284	2404
No Detection or Corruption	130	252	359	373

Table 8: Single stuck-at fault injection in designs with combinational redundancy.

# of Modules	1	2	3	3
Voter Type	---	---	Med	Maj
Injected Faults	952	1932	2932	2812
Corruption w/o Detection	825	1	4	1
Corruption w/ Detection	0	835	102	32
Detection w/o Corruption	0	857	2442	2428
No Detection or Corruption	127	239	384	351

Tables 7 and 8 show that in the single-threaded and single-modular designs, which do not have any fault tolerance or error-detecting scheme, almost all of the injected faults result in corruption. For the designs with fault tolerance techniques, the undetected corruption caused by single faults only occur when the fault is injected in the comparator

(error detector), I/O logic, or the common logic part, such as the scheduler. Therefore the undetected corruption is less likely to appear.

Table 9: Double stuck-at fault injection in designs with different voters (5000 fault pairs are randomly chosen).

Fault Tolerance Technique	Multi-Threading (3 threads)		Combinational redundancy (3 copies)	
Voter Type	Median	Majority	Median	Majority
Corruption w/o Detection	24	7	0	0
Corruption w/ Detection	1754	1437	1656	1796
Detection w/o Corruption	3139	3464	3250	3125
No Detection or Corruption	83	92	94	79

The two-threaded and two-modular designs only have error detecting but not fault masking mechanism. In these designs, the number of detections with corruption is roughly the same as the number of detections without corruption. The reason is because the output of the designs is from one of the two identical threads or modules, and the probability of fault occurrence in the output thread or module is roughly 50%. The reliability can be improved if diversity is used between modules or threads [Mitra 99].

From the single stuck-at fault results, the three-threaded and three-modular designs, which have fault-masking mechanism, have much less corruption than other designs. The results of both single and double stuck-at faults show that the designs with a majority voter produce smaller numbers of undetected corruption than the designs with a median voter. It means that the median voter is more vulnerable to stuck-at faults than the majority voter, and is less likely to detect corruption. The reason is that a median voter requires subtracters to select middle value among the three. In FPGAs, special carry chain is designed to speed up addition/subtraction. A stuck-at fault in a median voter may affect several output bits at the same time due to the error propagation through the carry chain. On the other hand, in a majority voter, voting is done bit-by-bit. Therefore, a stuck-at fault will affect, in the worst case, one bit of the outputs of the majority voter.

The numbers for corruptions without detection in designs with combinational redundancy are much smaller than those of designs with multi-threading. This is because multi-threaded designs have more hardware that is commonly shared by different threads, such as schedulers and control logic. Therefore, faults are more likely to affect resources that are shared by several threads, and consequently, an error detection mechanism will not detect them. On the contrary, the designs with combinational redundancy replicate the whole module, hence a single fault can affect only one module, and corruption occurs only if the fault is in the voter.

Recalling the implementation results of Sec. 4, the designs with combinational redundancy have better performance and less area overhead than those with multi-threading. From the fault injection experiments in this section, we can conclude that in our implementation, combinational redundancy provides higher dependability and has lower overhead. Therefore, combinational redundancy is more suitable for our robot application than multi-threading.

6. Conclusion

We have demonstrated that an ACS is a good platform for implementing robot control algorithms. As a case study, a robots control algorithm has been implemented on reconfigurable hardware. Results obtained from hardware emulation on the WILDFORCE platform have demonstrated that the performance of running the algorithm on reconfigurable hardware is comparable with that on a general-purpose processor. The ability of implementing systems with different precision to customize the design has demonstrated the advantage of an adaptive computing system. In conjunction with different voting schemes, fault tolerance techniques have been used in implementing the algorithm. Fault injection experiments show that combinational redundancy is less vulnerable to stuck-at faults.

7. Acknowledgements

The authors would like to thank Dr. Santiago Fernandez-Gomez, Philip Shirvani, Subhasish Mitra, Chao-Wen Tseng, and Siegrid Munda for their useful feedback and suggestions. We would also like to thank Mike Butts and Quickturn Design Systems for

their technical support. This work was supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. DABT63-97-C-0024.

References

[Annapolis 99] Annapolis Micro Systems Inc., www.annapmicro.com, 1999.

[Bass 97] Bass, J. M., G. Latif-Shabgahi, and S. Bennett, "Experimental Comparison of Voting Algorithms in Cases of Disagreement," *Proceedings. 23rd Euromicro Conference: New Frontiers of Information Technology*, pp. 516-23, 1997.

[Craig 89] Craig, J. J., *Introduction to Robots: Mechanics and Control*, Addison-Wesley Publishing Company, 2nd edition, 1989.

[Culbertson 97] Culbertson, W. B., R. Amerson, R. J. Carter, P. Kuekes, and G. Snider, "Defect Tolerance on the Teramac Custom Computer, " *Proc. IEEE Symp. FCCM '97*, pp. 116-123, Apr. 1997.

[Gokhale 98] Gokhale, M. and J. M. Stone, "NAPA C: Compiling for a Hybrid RISC/FPGA Architecture", *Proc. IEEE Symp. FCCM'98*, Apr. 1998.

[Hamilton 92] Hamilton, D. L., J. K. Bennett, and I. D. Walker, "Parallel Fault-Tolerant Robot Control," *Proceedings of SPIE*, Vol. 1829, pp. 251-61, 1992.

[Hamilton 94] Hamilton, D. L., M. L. Visinsky, J. K. Bennett, J. R. Cavallaro, and I. D. Walker, "Fault Tolerant Algorithms and Architectures for Robots", *Proceedings of MELECON '94 Mediterranean Electrotechnical Conference*, Vol. 3, pp. 1034-6, 1994.

[Hamilton 96a] Hamilton, D. L., J. R. Cavallaro, and I. D. Walker, "Risk and Fault Tolerance Analysis for Robots and Manufacturing," *Proceedings of 8th Mediterranean Electrotechnical conference on Industrial Applications in Power Systems, Computer Science and Telecommunications*, Vol. 1, pp. 250-5, 1996.

[Hamilton 96b] Hamilton, D. L., I. D. Walker, and J. K Bennett, "Fault Tolerance Versus Performance Metrics for Robot Systems", *Proceedings of IEEE International Conference on Robots and Automation*, Vol. 4, pp. 3073-80, 1996.

[Khatib 95] Khatib, O., K. Yokoi, K. Chang, D. Ruspini, R. Holmberg, A. Casal, and A. Baader, "Force Strategies for Cooperative Tasks in Multiple Mobile Manipulation Systems", *Robots Research 7, The Seventh International Symposium*, G. Giralt and G. Hirzinger, eds., pp. 333-342, Springer 1996.

[Kim 96] Kim, J. H., J. K. Lee, H. S. Eom, and J. W. Choe, "An Underwater Mobile Robot System for Reactor Vessel Inspection in Nuclear Power Plants," *Proc. of the Sixth International Symposium on Robots and Manufacturing*, pp. 351-6, 1996.

[Lach 98] Lach, J., W. H. Mangione-Smith, and M. Potkonjak, "Efficiently Supporting Fault-Tolerance in FPGAs, *" Proc. ACM/SIGDA Int'l. Symp. on FPGAs*, pp. 105-115, Feb 1998.

[Mitra 99] Mitra, S., N.R. Saxena, and E.J. McCluskey, "Design Diversity for Redundant Systems," *29th International Symposium on Fault-Tolerant Computing (FTCS-29)* Fast Abstracts, Madison, WI, pp. 33-34, June 15-18, 1999.

[Quickturn 99] Quickturn Design Systems (now part of Cadence Design), www.quickturn.com or www.cadence.com, 1999.

[Saxena 98] Saxena, N. R. and E.J. McCluskey, "Dependable Adaptive Computing Systems," *IEEE Systems, Man, and Cybernetics Conf.*, pp. 2172-2177, Oct. 11-14, 1998.

[Saxena 00] Saxena, N. R., S. Fernandez-Gomez, W.-J. Huang, S. Mitra, S.-Y. Yu, and E. J. McCluskey, "Dependable Computing and Online Testing in Adaptive and Configurable Systems," *IEEE Design and Test*, pp. 29-41, 2000.

[Smith 78] Smith, B. J., "A pipeline, Shared Resource MIMD Computer", *Proc. Intl. Conf. on Parallel Processing*, pp. 6-8, 1978.

[TelSys 99] TSI TelSys Inc., www.tsi-telsys.com, 1999.

[Thayer 76] Thayer, W. J., "Redundant Electrohydraulic Servoactuators," *Technical Bulletin 127*, MOOG INC. Controls Division, 1976.

[Thorton 64] Thorton, J. E., "Parallel Operation in the Control Data 660", *AFIPS Conf. Proc., Fall Joint Computer Conf.*, Vol. 26, pp. 33-40, 1964.

[Ting 94] Ting, Y., S. Tosunoglu, and B. Fernandez, "Control Algorithms for Fault-Tolerant Robots," *Proceedings of 1994 IEEE International Conference on Robots and Automation*, Vol. 2, pp. 910-15, 1994.

[Tosunoglu 93] Tosunoglu, S., "Fault Tolerance for Modular Robots," *Proceedings of IECON*, Vol. 3, pp. 1910-14, 1993.

[Toye 90] Toye, G., *Management of Non-homogeneous Functional Modular Redundancy for Fault Tolerant Programmable Electro-mechanical Systems*, Ph.D. thesis, Stanford University 1990.

[Toye 94] Toye, G. and L. J. Leifer, "Helenic Fault Tolerance for Robots," *Computers & Electrical Engineering*, Vol. 20, No. 6, pp. 479-97, 1994.

[Visinsky 95] Visinsky, M. L., J. R. Cavallaro, and I. D. Walker, "A Dynamic Fault Tolerance Framework for Remote Robots", *IEEE Transactions on Robots and Automation*, Vol. 11, No. 4, pp. 477-490, August 1995.

[Wong 96] Wong, K. K., E. C. Tan, and A. Wahab, "A VLSI Median Voter for Fault Tolerant Signal Processing Applications," *3rd International Conference on Signal Processing Proceedings*, Vol. 2, pp. 1574-7, 1996

[Wu 93] Wu, E. C., J. C. Hwang, and J. T. Chladek, "Fault Tolerant Joint Development for the Space Shuttle Remote Manipulator System: Analysis and Experiment," *IEEE Trans. On Robots and Manufacturing—Recent Trends in Research, Education, and Applications*, Vol. 9, No. 5, pp. 675-80, 1993.

[Yang 98] Yang, J.-M. and J.-H. Kim, "Fault-Tolerant Locomotion of the Hexapod Robot," *IEEE Transactions on Systems, Man, and Cybernetics – Part B: Cybernetics*, Vol. 28, No. 1, pp. 109-116, February 1998.

[Xilinx 99] Xilinx, Inc, www.xilinx.com, 1999.

Appendix A: Second-Order Linear Control Algorithm

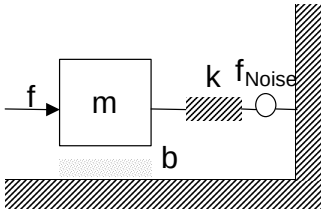


Figure 9. A second-order linear mechanic system.

Consider a second-order linear mechanic system. As shown in Fig. 9, an object of mass *m* is connected to the wall by a spring with coefficient *k*. Assume the frictional force is proportional to the object's velocity, and the friction coefficient is *b*. The position of the object is described by the variable *x*. Assume there exists a noise force f_{Noise}. The origin (x=0) is defined as the resting position of the object. Then the equation of motion is

(eq 1) $$f_{Noise} + m\ddot{x} + b\dot{x} + kx = f$$

We want the object to move along the desired trajectory, $x_d(t)$. To control the motion of the object, we want to apply a force $f(t)$ to the object parallel to x axis. The control law is

(eq 2) $f=m(\ddot{x}_d+K_d\dot{e}+K_p e+K_i\int e dt)+b\dot{x}+kx$

where

$$e=(x-x_d)$$

By equating f of equation 1 and 2, and differentiating both sides, the result is obtained in the form of a third order differential equation in terms of e:

(eq 3) $\dddot{e}+K_d\ddot{e}+K_p\dot{e}+K_i e=\frac{\dot{f}_{Noise}}{m}$

From equation 3, by assigning different values to the control gains K_p, K_i, and K_d, we can make $e(t)$ approach zero quickly. That is, $x(t)$ can approach $x_d(t)$ quickly if we choose proper values for K_p, K_i, and K_d.

A.1 Robot Block

In our design, the behavior of the robot block is emulated using hardware. We are going to solve x in the equation of motion (eq 1) to obtain the current position of the robot.

In a discrete-time system, differentiation and integration are performed as

(eq 4) $\int x(t)dt=\sum x(t)\Delta t=\sum x(n\Delta t)\Delta t=\sum x[n]\Delta t$

(eq 5) $\dot{x}(t)=\frac{x(t)-x(t-\Delta t)}{\Delta t}=\frac{x(n\Delta t)-x(n\Delta t-\Delta t)}{\Delta t}=\frac{x[n]-x[n-1]}{\Delta t}$

Then the original equation of motion (eq 1) becomes

(eq 6) $f_{Noise}[n]+m\left(\frac{x[n]-2x[n-1]+x[n-2]}{(\Delta t)^2}\right)+b\left(\frac{x[n]-x[n-1]}{\Delta t}\right)+kx[n]=f[n]$

We can combine the Δt part into the coefficients. Therefore,

(eq 7) $f_{Noise}[n]+m'(x[n]-2x[n-1]+x[n-2])+b'(x[n]-x[n-1])+kx[n]=f[n]$

where

$$m'=\frac{m}{(\Delta t)^2}$$

$$b'=\frac{b}{\Delta t}$$

Solving equation 7 in discrete time,

(eq 8) $$x[n] = \frac{m'(2x[n-1] - x[n-2]) + b'x[n-1]}{m'+b'+k} + \frac{f[n] - f_{Noise}[n]}{m'+b'+k}$$

The robot block will compute the current position from the above equation. The force $f[n]$ is the output of the control block.

A.2 Control Block

In the control block, we are going to compute the needed force f from the control law:

(eq 2) $$f = m(\ddot{x}_d + K_d\dot{e} + K_p e + K_i \int e dt) + b\dot{x} + kx$$

Similar to the robot block, we still use subtraction and addition to replace differentiation and integration in our discrete-time system. Therefore, the equation becomes

(eq 9) $$f[n] = m'(x_d[n] - 2x_d[n-1] + x_d[n-2]) + K_d'(e[n] - e[n-1]) + K_p e[n] + K_i' \sum e[n] + b'(x[n] - x[n-1]) + kx[n]$$

where

$$m' = \frac{m}{(\Delta t)^2}$$

$$b' = \frac{b}{\Delta t}$$

$$K_d' = \frac{K_d}{\Delta t}$$

$$K_i' = K_i \Delta t$$

In the control block, the inputs are the control gains K_p, K_i', and K_d'– whose values are decided by the user– and the coefficients m', b', and k– which describe the motion of the second-order linear system. In practical, m, b, and k are obtained by measuring the characteristics of the robot; in addition, the sampling interval Δt is decided by the system. Throughout this paper, we have used m'=40, b'=10, k'=10, K_p=0.4, K_i'=0.064, and K_d'=0.4.

Appendix B

On-line Testing and Recovery in TMR Systems for Real-Time Applications

(An extended version of "On-line Testing and Recovery in TMR Systems for Real-Time Applications," by Shu-Yi Yu and Edward J. McCluskey, *Proc. IEEE International Test Conference*, pp. 240-249, 2001.)

On-line Testing and Recovery in TMR Systems for Real-Time Applications

Abstract

Triple Modular Redundancy (TMR) is known to improve reliability in real-time computing systems for short mission times. However, TMR-based systems are not effective for longer missions. For failures caused by transient faults, we have designed a new recovery scheme for TMR systems so that they can be used for long-mission applications. The scheme can effectively recover computing systems from single transient faults without introducing any re-computation delay; hence, it is suitable for real-time applications. A robot controller is used as a case study. Theoretical analysis and implementation results show that, with very small hardware overhead for recovery logic, the proposed scheme can significantly improve system reliability and lengthen its lifetime. A new state restoration scheme for recovery is presented and shown to have lower overhead than a conventional restoration scheme.

1. Introduction

Many real-time computing systems are used under hazardous or remote circumstances, such as in nuclear power plants [Kim 96], aircraft, and spacecraft [Wu 93]. In these environments, computing systems are highly susceptible to errors due to radiation. Maintenance and repair is usually very expensive and time-consuming for these applications. In addition, timing is a critical issue because each transaction has to meet strict deadlines. Hence, fault tolerance with minimum time overhead becomes a requirement in real-time systems.

One approach to provide fault tolerance in real-time systems is through redundancy, such as N-modular redundancy (NMR) or duplex pairs. An NMR system replicates a computing resource into N modules running in parallel and uses voters to mask errors at outputs. Therefore, NMR can successfully tolerate faults as long as they happen in no more than $\lfloor N/2 \rfloor$ modules. A duplex-pair system uses two pairs of duplicated modules, which are four in total, for fault tolerance [Bartlett 78]. A module is duplicated for error detection purpose, and the duplex is used in pairs so that when an

error is detected in one of the duplex modules, the output from the other would be selected as the output and the system could still continue producing correct data. Although NMR and duplex pairs can increase reliability of systems, they are only effective for short mission times unless repair is possible. Several techniques have been developed to lengthen mission times in NMR systems. For permanent faults, self-purging redundancy [Losq 76] and NMR/Simplex [Mathur 71] [Mathur 75] have been proposed to lengthen system lifetimes. Permanent fault repair has also been implemented in duplex pair systems, such as Stratus computers [Webber 91]. For transient faults, the Draper Laboratory has built a real-time computing system with quadruple processors with transient fault recovery mechanism to improve reliability [Lala 86] [Adams 89] [Adams 90]. A quadruple system is good for preventing all single point failures including Byzantine failures[2] in distributed systems. Another possible solution is to use a duplex pair system that can easily be recovered from single failures by copying data from a correct duplex subsystem to the corrupt one. However, for a non-distributed computing system with no Byzantine failures, a TMR system, which has lower area overhead than a quadruple system and a duplex pair system, can be a better choice. Since TMR is extensively used in many applications [Siewiorek 00], there is a need of designing suitable transient error recovery schemes for TMR systems.

Transient error recovery is achieved by restoring correct states to continue operation. One way to restore correct states is through re-computation, which is called *rollback recovery* [Chandy 72]. In rollback recovery, system operation is backed up to some point in its processing, which is called a *checkpoint*. When an error occurs and is detected during operation, the system will restore its state back to the previous checkpoint and re-compute. Re-computation has been used in TMR systems for recovery [Chande 89]. However, re-computation has time overhead. Another way to obtain correct states is called *roll-forward recovery,* which copies correct states from a fault free redundant module or a spare to the faulty module to avoid re-computation. Roll-forward recovery schemes have been used in duplex systems with spares [Pradhan 92] [Xu 96] and quadraplex systems [Adams 89]. However, duplex systems with spares do not provide

[2]A Byzantine failure is a failure where a faulty processor continues execution and gives misleading information [11].

sufficient redundancy and roll-forward in these systems still has potential to cause re-computation even for single faults. A quadraplex system provides enough redundancy so that no single fault will require re-computation. But the cost of a quadraplex system is high. In addition, while states in the faulty module are being restored, other fault-free modules are still operational [Adams 89]. This can be used in a quadraplex system but may not be suitable in a TMR system. The reason is that in a quadraplex system with a faulty module being restored, three modules are left for execution; but in a TMR system when a faulty module is being restored, only two modules are operational and cannot provide error masking.

We present here a new approach to provide recovery for transient errors in TMR systems. Our recovery scheme uses a roll-forward approach to avoid re-computation and to meet strict transaction deadlines. By exploiting redundancy in TMR, no re-computation is needed for single module failures. Hence, it is very suitable for real-time applications. The three modules are synchronized during recovery, so error masking is always preserved during computation.

This paper is organized as follows. Section 2 defines the problem and describes our on-line testing and recovery scheme. Section 3 shows a reliability analysis of the roll-forward recovery system. We calculate reliability of a TMR system with the recovery scheme, and compare it with TMR systems without recovery. Section 4 describes the implementation of the system. Hardware overhead introduced by the recovery scheme is shown in this section. Section 5 concludes this paper.

2. On-line Testing and Roll-Forward Recovery

2.1 Problem Definition

Consider a computing module that has combinational circuitry and storage elements. The computing process in the computing module is a real-time task and each transaction has to be finished before its deadline. TMR is used to provide error masking for the module. During the module's operation, transient faults may occur in the computing module. An error may be latched into storage elements and the system may end up being in an incorrect state producing erroneous outputs in successive cycles. Our purpose is to correct the erroneous data residing in storage elements through recovery, so

that the module lifetime is not stopped by a transient fault. In addition, the performance impact caused by the recovery process has to be minimized, so that transactions' deadlines could still be met.

Our scheme is suitable for recovery from single transient faults in computing modules. The scheme can be used in conjunction with permanent fault repair schemes such as self-purging redundancy [Losq 76] to repair systems from permanent faults.

2.2 On-line Testing for TMR

In this section, we describe on-line testing used for TMR systems, and describe different forms of TMR in which we implement our roll-forward recovery scheme.

Two forms of TMR are discussed in this section. One is *hardware redundancy*, and the other is *multi-threading* [Saxena 98] [Yu 00]. In hardware redundancy, the hardware is replicated with two copies. A voter is used to determine the final output among the three modules. To add an on-line testing capability, voted outputs are compared with module outputs. On-line testing for TMR is shown in Fig. 1. The outputs of each module, noted as Out0, Out1, Out2, are compared with the voted output. If they are different, error signals, ERR0, ERR1, ERR2, are generated so that the faulty module is identified. Disagreement detectors have been used in previous schemes such as hybrid redundancy [Siewiorek 73] to remove a faulty module from the system.

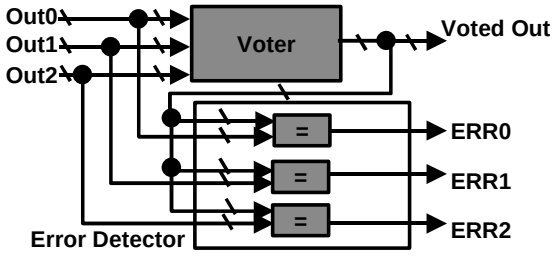


Figure 1. Voter/error detector for on-line testing in TMR.

Multi-threading is illustrated in Fig. 2. Figure 2 (a) shows an algorithm executed in multiple stages reusing hardware. An example of hardware reuse is arithmetic logic units (ALUs) in microprocessors. Different arithmetic operations may use the same ALU at different times for computation. For algorithms executed in multiple stages, some resources may be under-utilized (idle) because of data dependencies and memory latency.

To obtain fault tolerance by multi-threading, three copies of the same algorithm are executed as different threads in the hardware. As shown in Fig. 2 (b), the replicated instructions of redundant threads use hardware resources which are originally idle. The final output is obtained by voting among the results from the three threads. When transient faults are present, the redundant threads provide a means for fault masking. To add on-line testing capability, the voter/error detector shown in Fig. 1 is used.

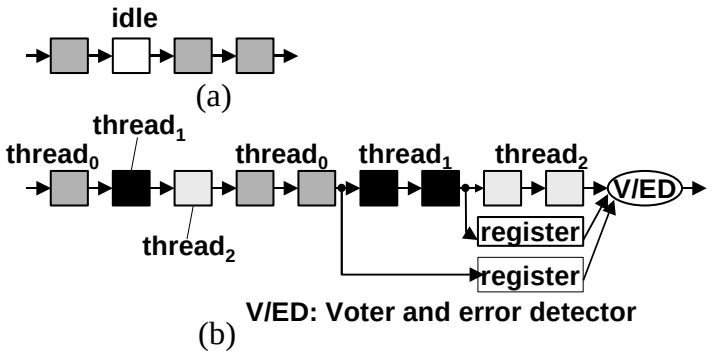


Figure 2. Multi-threading: (a) one-threaded computation, (b) three threads for fault tolerance.

Multi-threading is not suitable for systems in which no resource is idle at anytime. In these systems, we cannot take advantage of idle resources for redundant computation. Therefore, another form of TMR needs to be used.

Hardware redundancy has small time overhead. In addition to transient faults, it is capable of tolerating permanent faults. However, it is expensive in terms of hardware overhead. Redundancy obtained by multi-threading is as effective as hardware redundancy for tolerating transient faults, where multi-threading might have smaller hardware overhead since hardware is reused. However, computation latency in multi-threading may be increased. The amount of time overhead in multi-threading depends on the resource usage and data dependency, and it varies among applications. In a computation that has high hardware utilization, a recovery scheme with multi-threading is not necessarily suitable to replace rollback recovery, since a three-threaded design may cause delay longer than simple re-computation after each checkpoint at which there is an error. Since our recovery method is not restricted to a particular redundancy technique, throughout this paper we will use the term *copy* to describe both original and replicated

modules or threads. For hardware redundancy, a copy means a replicated hardware *module*, and for multi-threading, a copy means a replicated computation *thread*. In the following discussion, we use the notation *HR* for hardware redundancy, and *MT* for multi-threading.

2.3 Recovery Scheme

To determine if there is need for recovery, three computation copies are compared and the results are examined at pre-scheduled computation points. We define these computation points as *checkpoints*. At checkpoints, if comparators detect disagreement among three copies, a recovery process will be initiated. Inputs are buffered and will be processed when the recovery process finishes.

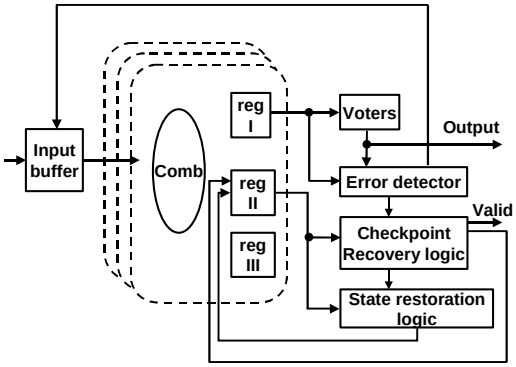


Figure 3. The block diagram of a TMR system with the roll-forward recovery scheme.

Figure 3 shows a block diagram of a TMR system with recovery mechanism. The system consists of three hardware copies or three computation threads, an input buffer, a voter for the output, an error detector, a checkpoint logic/recovery control block, and a state restoration block. In each copy, registers can be divided into three different sets. Register I is the set of registers that contain output data. As shown in the diagram, their outputs are connected to the voter. Register II is the set that contains data that are still needed for computation after checkpoints. Examples for Register II are program counters in microcontrollers and microprocessors. Therefore, data stored in Register II need to be recovered at a checkpoint when a fault happens. As shown in the diagram, these registers are connected to the state restoration logic block and the checkpoint/recovery block. Register III is the set of registers that store intermediate results that are not needed after

the checkpoints. They do not need to be recovered when faults are present. Examples for Register III are registers that store intermediate results for subfunction calls. After the subfunction finishes and returns to a main function, these data are not needed anymore. In some applications, Register I and Register II may have a portion that overlaps. That is, some registers that contribute to the output also need to be recovered during recovery.

There are two major reasons to initiate recovery processes at preset checkpoints instead of as soon as a fault happens. One reason is that with pre-scheduled checkpoints and recovery processes, a fault can be recovered after real-time transactions are finished, hence, transactions will not be interrupted or delayed by the recovery process. The other reason is in some applications, checkpoints can be inserted in a way that the amount of internal data or states, which need to be recovered, is minimized. For example, during a computation task there can be many intermediate values that only contribute to the current task and are not needed for the next task. If we insert checkpoints at completion of tasks, there is no need to recover registers that store these intermediate values since they are not needed anymore. Certainly, if a recovery process is not initiated immediately when a fault happens, extra faults might happen before the recovery process is actually initiated. Because of multiple faults, the system might not be recovered successfully. Longer delay for recovery processes increases the probability of multiple faults. However, a recovery process that can be initiated at any clock cycle is too expensive in terms of both hardware and time overhead. Therefore, recovery processes are scheduled at checkpoints.

During the system's normal operation, all copies execute their functions normally. System outputs are obtained by voting. At checkpoints, the checkpoint/recovery controller checks the error detection result. Figure 4 shows the logic of checkpoint and recovery. Checkpoints can be inserted by adding an additional *checkpoint state* in the finite state machine or by counting cycles to a pre-defined *checkpoint period*. As shown in Fig. 4 (a), if a checkpoint state is used, when the current state in the finite state machine is the checkpoint state, a checkpoint process is initiated. If a counter is used for checkpoints, the counter value is compared with the checkpoint period, and when they are the same, a checkpoint process is initiated. In a checkpoint process, error indication signals, ERR0, ERR1, and ERR2 from comparators, are examined to see if there is need

for recovery. The logic that initiates a recovery process is shown in Fig. 4 (b). If errors are detected, then a recovery process is initiated. If no errors are detected, the normal process will continue until the next checkpoint. During the recovery process, a signal will be raised to indicate that outputs are invalid, and operation is temporarily stopped. The state restoration block provides correct data to recover states stored in Register II. State restoration will be discussed in detail in the following paragraphs. Incoming data are buffered and will be processed after recovery is completed. When the restoration process is finished, normal operation will continue.

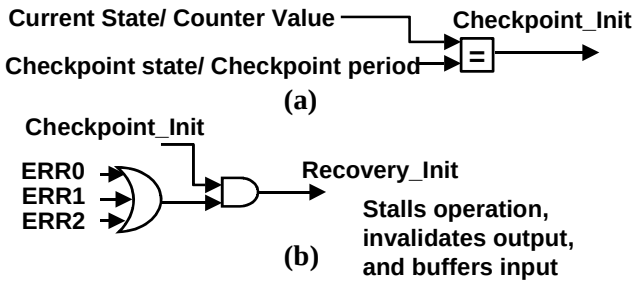


Figure 4. Logic used in the recovery scheme: (a) checkpoint logic, and (b) recovery logic.

The state restoration process is done by restoring internal states in the faulty copy from correct copies. Two schemes for obtaining correct states are described here. They are illustrated in Fig 5. One scheme, the *voting scheme*, is to obtain correct states by voting among the three copies. In [Adams 89], voters are used to obtain correct states for recovery. The voted results are loaded into all three copies. As shown in Fig. 5 (a), multiplexers (MUXs) are inserted between registers and signals that were originally connected to the registers. During normal function, original signals are selected to be loaded into the registers. During recovery, voted signals are selected to loaded into the registers. We present a new scheme, the *direct-load scheme*, which obtains correct states by loading states directly from one of the correct copies into the faulty one. As shown in Fig. 5 (b), MUXs are added at inputs of Register II. During normal operation, original signals are selected to be loaded into the registers. When recovery is initiated, the data in fault-free copies are loaded into the faulty registers. If a copy is fault-free, then its register value is held and the content is not changed.

To compare these two schemes, the fault scenarios can be categorized as follows:

(1) Only one copy is faulty: In this case, both schemes can remove incorrect states from the faulty copy.

(2) Multiple copies are faulty, but each faulty copy has incorrect data located at different registers: In this case, since each faulty copy has incorrect data at different registers, the majority voter will always provide correct data for recovery. Then the data are loaded into all three copies to recover faults. Hence, the voting scheme can potentially recover incorrect states successfully in the presence of multiple faults. However, for the direct-load scheme, it could not identify and correct the faulty copy due to multiple faults.

(3) Multiple copies are faulty, but faults are located in the same registers on different copies: In this case, since there are multiple faults, the direct-load scheme cannot recover them correctly. As for the voting scheme, since faulty copies have the same registers that contain incorrect data, the majority voting process is not able to give correct data for these registers. Hence, both schemes will not work in this situation.

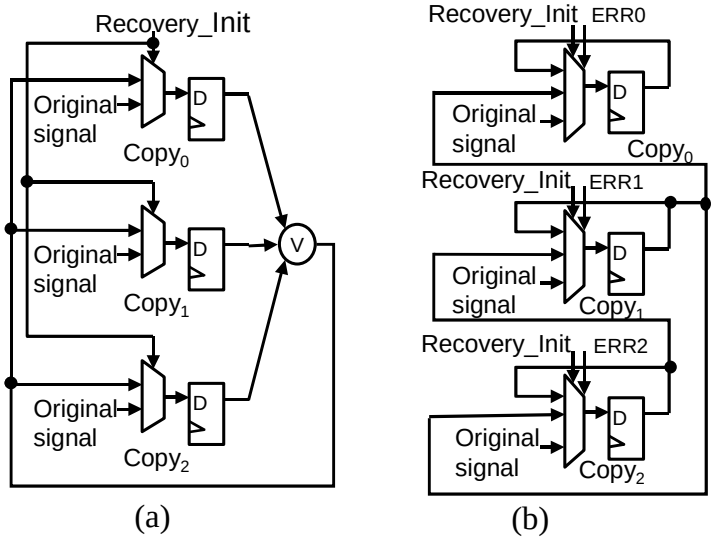


Figure 5. The data restoring process: (a) the voting scheme and (b) the direct-load scheme.

From the discussion of the above fault scenarios, we see that the voting scheme can recover more faults than the direct-load scheme. The direct-load scheme needs larger MUXs to select signals, and the voting scheme needs hardware for voters. We implement these two different schemes in the recovery technique to compare their hardware

overhead. To compare reliability between the two schemes, a detailed analysis is required, and it is beyond the scope of this paper. The analysis would require the information of data dependency so that we can calculate the probability of the above three situations.

Figure 6 shows a fault scenario for recovery. As shown in the figure, when a single fault occurs, erroneous outputs are masked by voters. Error detection results are checked at checkpoints by extra hardware as shown in Fig. 4. When a faulty copy is identified, the corrupted data is recovered from correct data in fault-free copies. After recovery normal operation is continued again. As shown in the figure, for single copy failures, the fault-free copies provide sufficient information for recovery. Therefore, no rollback is needed. However, failures in common logic such as voters and recovery logic cannot be recovered. To further improve reliability, a fault tolerance mechanism is added to common logic.

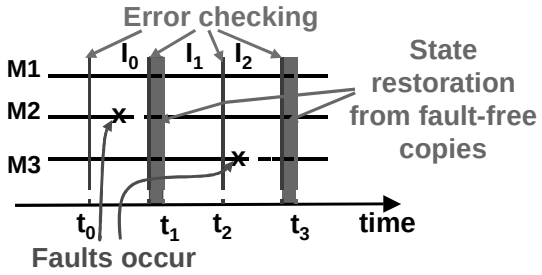


Figure 6. A fault scenario for recovery.

3. Reliability Analysis

Reliability analysis of the roll-forward recovery scheme is shown in this section. A Markov chain is used to model a TMR system with the recovery scheme. To show the improvement, the reliability of the system is calculated and compared with a simplex system, a TMR system, and a TMR-Simplex system. A detailed derivation of the analysis is in the appendix.

We choose a robot controller as our case study to calculate reliability because a robot controller is usually used in real-time applications and is very often life-critical so that it needs fault tolerance mechanisms [Hamilton 92]. A second-order linear algorithm is chosen as the control algorithm for the robot controller [Craig 89]. This work is continuation of our ROAR project [ROAR 01]. In our previous work [Yu 00], we have

implemented the robot controller in a reconfigurable platform, the WILDFORCE® board [Annapolis 01], and have added fault-masking capability in it via TMR. We have demonstrated that a reconfigurable platform is suitable for implementing robot controllers in terms of both reliability and performance.

Figure 7 shows a block diagram of a robot system [Craig 89]. The robot controller collects data of current robot position and its trajectory to calculate needed force. The interface between the robot and the controller includes sensors and A/D, D/A converters. The calculated value is converted to an analog signal, and that signal drives the motor to move the robot.

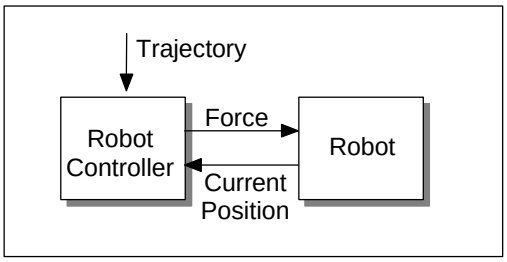


Figure 7. Block diagram of a robot system.

In our analysis, the hardware area and timing data of the robot controller are used to calculate its reliability. The failure probability of the recovery and checkpoint logic is taken into consideration. The robot controller executes the control algorithm, and each computation of the algorithm takes 7 clock cycles. For TMR in hardware redundancy, the whole computation takes 7 clock cycles, which is the same as the simplex design because delay of the voter is small and is not part of the critical path. The three-threaded controller takes 11 clock cycles to complete three computations with the same computation resource as a single-threaded design. Checkpoints are set at the completion of each computation. The recovery process takes one clock cycle. In a larger design with a larger area and more registers to restore when faults are present, more clock cycles will be needed for recovery. The hardware area data is obtained from synthesis using the Synplify® software. We chose the Xilinx Virtex XCV1000 as the target library. The reason we choose an FPGA as the synthesis target for our analysis is that the controller design is going to be implemented on a Virtex FPGA chip in the ROAR project. The unit of synthesized area is CLB (Configurable Logic Block) slices. One CLB slice contains

two registers and two LUTs (lookup tables.) We approximate the number of CLB slices as the maximum number of LUTs/2 and regs/2. A detailed description of implementation and synthesis is in Sec 4. The block area and timing information used for reliability calculations is listed in the appendix A.3. Only the voting scheme for state restoration is used in the calculation. The reason is that the voting scheme has larger area in state restoration logic and has larger overhead (which will be shown in Sec. 4). We use the larger area to compute the worst case for reliability for comparison with systems that do not have any recovery mechanism.

Figure 8 shows the reliability of the system, with the time axis normalized to mean time to failure (MTTF) of the simplex design. The figure shows that our recovery scheme has much better reliability than Simplex, TMR w/o recovery, and TMR-Simplex systems. Both hardware redundancy and multi-threaded TMR have shown the improvement. In addition, in the hardware-redundant design, the average lifetime, which is the area under the reliability curve, is 9.6 times the lifetime of the TMR design w/o any recovery. However, multi-threaded TMR has slightly lower reliability than hardware TMR, and its lifetime is 8.4 times the lifetime of the TMR design w/o any recovery. This is because in a multi-threaded TMR system, the execution time between checkpoints is longer than that in a hardware TMR system, therefore each thread is more vulnerable to errors between two checkpoints. Moreover, the multi-threading design has larger common logic, such as the thread schedular, that is shared by the three threads and is not protected by multi-threading. The shared logic can be triplicated to further improve reliability for multi-threading designs.

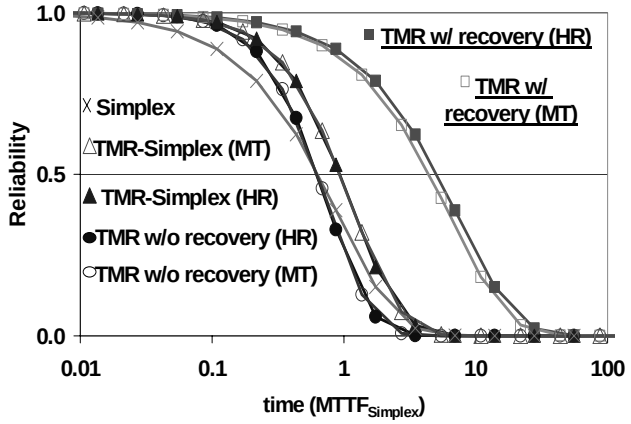


Figure 8. Reliability of the robot controller with different fault tolerance techniques.

4. Implementation

In order to estimate the overhead of the roll-forward recovery scheme, the scheme was implemented in the robot controller. The details of the implementation and the synthesized results are described in this section.

We want to investigate the cost of the recovery scheme using different redundancy techniques and different state restoring methods, as mentioned in Sec. 2. We also want to compare its cost with a TMR system without any recovery mechanism. Therefore, the robot system is implemented with different designs as follows:

(1) A simplex system.

(2) A TMR-HR system without any recovery scheme. It contains three copies of the controller and a voter for output results.

(3) A TMR-MT system without any recovery scheme. It contains a controller with three threads of computation and a voting mechanism.

(4) A TMR-HR system with the recovery scheme, and state restoration obtained through voting. The hardware contains three copies of the robot controller, each with extra multiplexers with registers for recovery. It also contains logic for checkpoint control and a voting mechanism for both output and internal states.

(5) A TMR-MT system with the recovery scheme, and state restoration obtained through voting. The hardware contains a controller with three computation threads. The registers that need to be recovered are connected with multiplexers at their inputs. The system also contains logic for checkpoint control and voting mechanism for both output and internal states.

(6) A TMR-HR system with the recovery scheme, and state restoration obtained through the direct-load scheme. The hardware is similar to that in (4), except that it does not have voters for state restoration.

(7) A TMR-MT system with the recovery scheme, and state restoration obtained through the direct-load scheme. The hardware is similar to that in (5), except that it does not have voters for state restoration.

To evaluate the cost when the system is implemented in both reconfigurable hardware and ASICs, the designs are synthesized with target libraries including Xilinx Virtex XCV1000 FPGA, and LSI lcbg10p library, using Synplify® and Synopsys® tools,

respectively. Each design is divided into multiple function blocks, and each block is synthesized with timing optimization. The results are listed in Table 1. The numbers in parentheses are the area normalized with the result obtained from the simplex design. Results of non-combinational logic (registers) and combinational logic (LUTs) are listed.

Table 1. Synthesis results for a robot controller with/without recovery mechanism.

	TMR Type	Recovery	State Restore	Xilinx XCV1000		LSI lcbg10p	
				Regs	LUTs	Non-comb	Comb
Simplex	N/A	No	N/A	402 (1.00)	654 (1.00)	11619 (1.00)	24404 (1.00)
TMR	HR	No	N/A	1206 (3.00)	1979 (3.03)	34857 (3.00)	73399 (3.01)
		Yes	Voting	1242 (3.09)	2323 (3.55)	35636 (3.06)	76575 (3.14)
			Direct-load	1242 (3.09)	2262 (3.45)	35636 (3.06)	76143 (3.13)
	MT	No	N/A	924 (2.30)	1514 (2.31)	27273 (2.35)	31432 (1.29)
		Yes	Voting	958 (2.38)	1819 (2.78)	28002 (2.41)	34740 (1.42)
			Direct-load	958 (2.38)	1906 (2.76)	27996 (2.41)	34266 (1.40)

From Table 1, several observations can be made:

(1) Recovery overhead: Recovery overhead, which is the difference between the area of TMR w/o any recovery and TMR w/ recovery, is 6%~9% in sequential logic and 11%~47% in combinational logic, with respect to simplex area. It is small compared to the overhead introduced by TMR (300% in hardware redundancy and 121%~241% in multi-threading).

(2) Combinational (LUTs) vs. non-combinational (registers) overhead: The non-combinational logic overhead of designs with recovery mechanism does not differ much from that of TMR designs without recovery. The reason is that the major overhead for recovery in non-combinational logic is the input buffers, and they are not many compared with registers in a simplex design. The combinational logic overhead varies a lot among different designs and different libraries. This will be discussed in the paragraphs (3) through (5).

(3) Comparison between different libraries: For the same designs, the sequential logic overhead does not differ much among different libraries (maximum 5%), but the combinational logic varies more (maximum 136%). The combinational overhead

obtained from LSI lcbg10p library is smaller than that from XCV1000 FPGA library. The reason is that in XCV1000, most combinational logic has to be synthesized into 4-input LUTs. Hence, the cost for a two-input logic block or a three-input logic block costs as much as a four-input logic block in the FPGA. Multi-threaded designs contain many MUXs in order to connect replicated registers to the same computation hardware. In addition, in the designs with a recovery mechanism, the voters create many three-input logic blocks, and two-input multipliexers are added for the registers that need to be restored. Therefore, in these designs, synthesized results with the FPGA library give more combinational overhead than those with LSI lcbg10p library. The lack of flexibility in logic block mapping in FPGAs induces larger combinational logic overhead. Nevertheless, by giving away the logic block mapping flexibility and logic mapping efficiency, the FPGA provides the flexibility of reconfiguration so that we can exploit it to further repair a design with permanent fault [Huang 01].

(4) Hardware redundancy vs. multi-threading: In our case, all multi-threaded designs, with or without recovery, require less hardware overhead than any of the hardware-redundant designs. The reduced sequential overhead is from the shared registers in common logic blocks. The reduced combinational overhead is from the shared computation resources in multi-threaded designs. However, the number of clock cycles of computation in multi-threaded designs is more than those of hardware redundant designs. This shows that in applications where hardware resources are limited but performance requirements are not so strict, multi-threading can be an appealing approach. However, this is not necessarily true for all applications. For some designs that do not have idle stages, multi-threading may not be an efficient scheme to implement TMR and can introduce more overhead than hardware redundancy.

(5) State restoration techniques: The direct-load method has smaller combinational overhead than the voted method for state restoration. Their non-combinational overhead is the same. The reduced combinational overhead comes from not using voters in the direct-load method.

Summarizing, we found that the recovery logic introduces small overhead compared with hardware of TMR in both hardware redundancy and multi-threading.

Among state restoration schemes, our direct-load scheme has less overhead than the voting scheme. The overhead is less in ASICs than in FPGAs, but with the design implemented in FPGAs we could obtain reconfigurability and more flexibility.

5. Conclusion

In this paper, we have presented a new roll-forward recovery scheme that improves reliability of TMR systems. Unlike other roll-forward recovery schemes, our scheme exploits redundancy in TMR so that no re-computation is needed for single-failure recovery. Therefore, it is very suitable for real-time applications. Reliability analysis of a case study has shown that reliability of a TMR design is significantly improved by adding the recovery scheme. We presented a new state restoration scheme, the direct-load scheme, which is capable of recovering faults in a single faulty module. Synthesis results show that the scheme has smaller overhead than the voting scheme, a conventional state restoration approach. However, it cannot recover as many multiple faults as the voting scheme. We have applied the scheme to two forms of TMR: hardware redundancy and multi-threading. Synthesis area data show that the recovery mechanism introduces small hardware overhead in both forms of TMR.

References

[Adams 89] Adams, S. J., "Hardware Assisted Recovery from Transient Errors in Redundant Processing Systems," *FTCS 19th Digest of Papers*, pp. 512-9, 1989.

[Adams 90] Adams S. J. and T. Sims, "A Tagged Memory Technique for Recovery from Transient Errors in Fault Tolerant Systems", *Real-Time Systems Symposium*, pp. 312-321, 1990.

[Annapolis 01] Annapolis Micro Systems Inc., http://www.annapmicro.com, 2001.

[Baze 97] Baze, M. P., "Attenuation of Single Event Induced Pulses in CMOS Combinational Logic," *IEEE Trans. on Nuclear Science*, Vol. 44, No. 6, pp. 2217-23, 1997.

[Bartlett 78] Bartlett, J. F., Tandem Computers Inc., Cupertino, CA, "A Nonstop Operating System," *Proc. of the Eleventh Hawaii International Conf. On System Sciences*, pp. 103-17, 1978.

[Buchner 97] Buchner, S., M. Baze, D. Brown, D. McMorrow, and J. Melinger, "Comparison of Error Rates in Combinational and Sequential Logic," *IEEE Trans. on Nuclear Science*, Vol. 44, No. 6, pp. 2209-16, 1997.

[Chande 89] Chande, P. K., A. K. Romani, and P. C. Sharma, "Modular TMR Multiprocessor System," *IEEE Trans. on Industrial Electronics*, Vol. 36, No. 1, pp. 34-41, 1989.

[Chandy 72] Chandy, K. M. *et al.*, "Rollback and Recovery Strategies for Computer Programs," *IEEE Trans. on Computers*, Vol. C-21, No. 6, pp. 546-56, 1972.

[Craig 89] Craig, J. J., *Introduction to Robots: Mechanics and Control*, Addison-Wesley Publishing Company, 2nd edition, 1989.

[Hamilton 92] Hamilton, D. L., J. K. Bennett, and I. D. Walker, "Parallel Fault-Tolerant Robot Control," *Proceedings of SPIE*, Vol. 1829, pp. 251-61, 1992.

[Huang 01] Huang, W.-J. and E. J. McCluskey, "Column-Based Precompiled Configuration Techniques for FPGA Fault Tolerance," to appear in *FCCM'01*, 2001.

[Kim 96] Kim, J. H., J. K. Lee, H. S. Eom, and J. W. Choe, "An Underwater Mobile Robot System for Reactor Vessel Inspection in Nuclear Power Plants," *Proc. of the Sixth International Symposium on Robots and Manufacturing*, pp. 351-6, 1996.

[Lala 86] Lala, J. H., L. S. Alger, R. J. Gauthier, M. J. Gauthier, and M. J. Dzwonczyk, "A Fault Tolerant Processor to Meet Rigorous Failure," *Proc. of IEEE/AIAA 7th Digital Avionics Systems Conf.*, pp. 555-562, 1986.

[Liden 94] Liden, P., P. Dahlgren, R. Johansson, and J. Karlsson, "On Latching Probability of Particles Induced Transients in Combinational Networks," *Proc. of IEEE 24th International Symposium on Fault-Tolerant Computing*, p. 340-9, 1994.

[Long 90] Long, J., W., W. K. Fuchs, and J. A. Abraham, "A Forward Recovery Using Checkpointing in Parallel Systems," *Proc. of the 1990 International Conf. on Parallel Processing*, Vol. 1, pp. 272-275, August 1990.

[Long 92] Long J. and W. K. Fuchs, "Implementing Forward Recovery Using Checkpoints in Distributed Systems," *Dependable Computing for Critical Applications 2*, pp. 27-46, 1992.

[Losq 76] Losq, J. "A Highly efficient Redundancy Scheme: Self-Purging Redundancy," *IEEE Trans. Comp*. C-25, pp. 569-578, 1976.

[Mathur 71] Mathur, F. P., "Reliability Estimation Procedures and CARE: The Computer Aided Reliability Estimation Program," *Jet Propulsion Laboratory Quarterly Tech. Review 1*, Oct 1971.

[Mathur 75] Mathur F. P. and P. DeSousa, "Reliability Modeling and Analysis of General Modular Redundant Systems," *IEEE Trans. Rel.*, R-24, No. 5, pp. 296-9, 1975.

[Nett 97] Nett, E. and M. Mock, "A Recovery Model for Extended Real-Time Transactions," *Proc. 1997 High-Assurance Engineering Workshop*, pp. 124-7, 1997.

[Pradhan 92] Pradhan, D. K., and N. Vaidya, "Roll-Forward Checkpointing Scheme: Concurrent Retry with Nondedicated Spares," *IEEE Workshop on Fault Tolerant Parallel and Distributed Systems*, pp. 166-74, 1992.

[ROAR 01] http://www-crc.stanford.edu/projects/roar/roarSummary.html, 2001.

[Saxena 98] Saxena, N. R. and E.J. McCluskey, "Dependable Adaptive Computing Systems," *IEEE Systems, Man, and Cybernetics Conf.*, pp. 2172-2177, Oct. 11-14, 1998.

[Siewiorek 73] Siewiorek, D. P. and E. J. McCluskey, "Switch Complexity in Systems with Hybrid Redundancy," *IEEE Trans. Comp.*, C-22, pp. 276-282, 1973.

[Siewiorek 00] Siewiorek, D. P. and R. S. Swarz, *Reliable Computer Systems – Design and Evaluation*, 3rd Ed, Digital Press, 2000

[Trivedi 82] Trivedi, K. S., *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*, Prentice-Hall, Inc., 1982.

[Webber 91] Webber, S. and J. Beirne, "The Stratus Architecture," *Digest of Papers. Fault-Tolerant Computing: Twenty-First International Symposium*, pp. 79-85, 1991.

[Wu 93] Wu, E. C., J. C. Hwang, and J. T. Chladek, "Fault Tolerant Joint Development for the Space Shuttle Remote Manipulator System: Analysis and Experiment," *IEEE Trans. on Robots and Manufacturing—Recent Trends in Research, Education, and Applications*, Vol. 9, No. 5, pp. 675-80, 1993.

[Xu 96] Xu, J. and B. Randell, "Roll-Forward Error Recovery in Embedded Real-Time Systems," *Proc. of 1996 Int'l Conf. On Parallel and Distributed systems*, pp. 414-21, 1996.

[Yu 00] Yu, S.-Y., N. Saxena, and E. J. McCluskey, "An ACS Robot Control Algorithm with Fault Tolerant Capabilities," *IEEE Symp. on Field-Programmable Custom Computing Machines*, pp. 175-84, 2000.

Appendix A: Reliability Analysis

A.1 Analysis of a TMR System with the Recovery Scheme

Reliability analysis for a TMR system with our roll-forward recovery technique is described in this section. Our scheme is designed for recovery of transient faults, therefore, only transient faults are considered in the analysis.

All modules are assumed to be identical and faults are independent of one another. The system will only produce a correct output if two or more of the copies produce the same result and common logic blocks such as the checkpoint controller, the output voter, and the error detector function correctly. Otherwise the system is considered as failed. In addition to the targeted module for error recovery, the reliability of common logic blocks is taken into consideration.

To obtain a lower bound on reliability, we assume that once a transient fault occurs in any component (such as a gate) of a logic block, the logic block fails to produce correct outputs until the error is recovered or is overwritten by further data. The reason is that although a transient fault may only appear in a logic block for a very short period of time, the erroneous data corrupt by the fault can be written into a register and stays.

Hence, the faulty data appear as permanent faults until they are recovered or overwritten. We also assume that an incorrect internal signal will always produce erroneous outputs at its following checkpoint and will be detected if the error detector is fault-free.

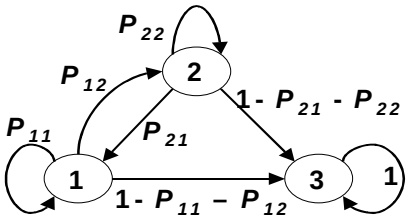


Figure A.1. A state diagram of a TMR system with the roll-forward recovery scheme.

We define a *checkpoint period* as the period between two checkpoints. Figure A.1 shows the Markov state diagram of a TMR system with the roll-forward recovery scheme. State transitions occur at the beginning of every checkpoint period. Assume that normal system operation time and recovery time between checkpoints are constants, so that transition probabilities are constants. Definition of each state is stated as follows:

State 1: At the beginning of a checkpoint period, the system is functioning and all of the three copies are fault-free.

State 2: At the beginning of a checkpoint period, the system is functioning and two of the three copies are fault-free. This happens when one copy becomes faulty during the previous checkpoint period, but a fault re-occurs in the faulty copy during the recovery process. An example of errors that will corrupt computation for the next checkpoint is a bit-flip in a register, which belongs to Register II in Fig. 3, during a recovery process. Since faults in Register II will damage subsequent operation, the copy fails to produce correct output at the next checkpoint. The effect of State 2 cannot be neglected when recovery processes are long or recovery logic is complicated.

State 3: The system fails. It happens when faults occur to any of the logic that is shared by the triplicated copies during normal operation or recovery process, or when two or more copies are faulty.

Assume that transient errors occur independently in each unit area in unit time. The system has a single clock source, and the clock period is constant. We define the transient error rate per unit area per clock cycle as *p*. Research has shown that transient error rate is different for combinational and sequential circuits [Buchner 97]. In our

analysis, we assume that storage elements are distributed uniformly among combinational circuits; hence, failure rate can be assumed as a constant per unit area per unit time. This can be close to reality if synthesized circuit is homogeneous so that the density of storage elements is approximately uniform. The probability that a logic block with area A functions in a c clock-cycle duration is $(1-p)^{A\times c}$.

To illustrate all area parameters that will be used in the following discussion, referring to Fig. 3 again for the recovery block diagram. The following symbols denote area of each block: A_M for a module, $A_{REG\text{-}II}$ for registers and related logic in Register II; A_{INP} for input buffer; A_{VOTE} for voters, A_{ECR} for ECR blocks; and $A_{RESTORE}$ for restoration logic. For a multi-threaded design, another parameter A_{SCHE} needs to be included for the area of a thread scheduler, which is shared by three threads.

The following notation is used for timing parameters:

c_{normal}: clock cycle counts for normal operation between two checkpoints.

c_{recov}: clock cycle counts for a recovery process.

$c_{checkpoint}$: clock cycle counts in a checkpoint period. Therefore,

$$c_{checkpoint} = \begin{cases} c_{normal} + c_{recov}, & \text{when there is recovery,} \\ c_{normal}, & \text{when there is no recovery.} \end{cases}$$

To express transition probabilities, the probability of each component being fault-free between two checkpoints are denoted as follows:

P_M: probability that a copy (hardware module or thread) is fault-free during normal operation and error detection between two checkpoints. For a copy with effective area A_M, $P_M = (1-p)^{A_M \times c_{normal}}$.

P_{COMM}: probability that common logic, which is shared by the three copies, is fault-free during normal operation. The duration is c_{normal}. For hardware redundancy, the common logic includes the voter, input buffer, and error detection/checkpoint logic/recovery control. The state restoration is not included because it will only be used during recovery.

$$P_{COMM} = (1-p)^{\left(A_{INP} + A_{VOTE} + A_{ECR}\right)\times c_{normal}}.$$

For multi-threading, it also includes the multi-thread scheduler, hence,

$$P_{COMM} = (1-p)^{\left(A_{INP} + A_{VOTE} + A_{ECR} + A_{SCHE}\right)\times c_{normal}}$$

P_{REC}: probability that the common logic is fault-free during recovery, c_{recov}, given that an error occurs during normal operation and the recovery control process needs to be initiated. The related logic includes the input buffer, error detection/checkpoint logic/recovery control, and state restoration logic. Therefore,

$$P_{REC} = (1-p)^{\left(A_{INP} + A_{ECR} + A_{RESTORE}\right)\times c_{recov}}.$$

$P_{M\text{-}REC}$: probability that no fault occurs to a copy during recovery, c_{recov}, to cause incorrect output for the next checkpoint. The logic that its faults will cause failure to a copy includes all the registers that are accessed during a recovery process, and all related logic such as multiplexers connected to these registers. Thus,

$$P_{M-REC} = (1-p)^{A_{REG-II}\times c_{recov}}.$$

$P_{M\text{-}REC}$ cannot be neglected if the time needed for recovery is not very small compared with checkpoint periods.

The symbol *Pij* means the probability of a transition from State *i* to State *j*. *Pij*'s are described as follows:

P_{11}: probability that the system stays in State 1. This happens when no faults occur to the hardware or when one copy is faulty but is correctly recovered. Hence,

P_{11} = Prob(no faults occur during normal function) + Prob(only one copy is faulty during normal function, and no faults occur during recovery process)

$= (P_M^3 \times P_{COMM}) + (3 \times P_M^2 \times (1\text{-}P_M) \times P_{COMM} \times P_{REC} \times P_{M\text{-}REC}^3)$.

P_{12}: probability of transition from State 1 to State 2, that is, when faults occur in one module during normal function and error detection, and fault occurs to essential logic in the faulty copy during recovery. No other faults can happen to the common logic or other two fault-free copies, otherwise the system will go to State 3. Hence,

P_{12} = Prob(only one module fails during normal function and the recover process is initiated) × Prob(only fault occurs to essential logic in the faulty module, given that the recovery process is initiated)

$= (3 \times P_M^2 \times (1\text{-}P_M) \times P_{COMM}) \times (P_{REC} \times P_{M\text{-}REC}^2 \times (1\text{-}P_{M\text{-}REC}))$.

P_{21}: Probability of transition from State 2 to State 1. This happens when no extra faults occur to the original fault-free copies, and the original faulty copy is recovered successfully. Hence,

P_{21} = Prob(no faults happen to the two fault-free modules and common logic during normal function and the recovery process is initiated) × Prob(no faults happen in the recovery process)

= $(P_M^2 \times P_{COMM}) \times (P_{REC} \times P_{M\text{-}REC}^3)$.

P_{22}: Probability that the system stays in State 2. This happens when no extra faults occur to the original fault-free hardware, and faults re-occur to the original faulty copy. Hence,

P_{22} = Prob(no faults happen to the two fault-free modules and common logic during normal function and the recovery process is initiated) × Prob(only fault occurs to essential logic in the faulty module, given that the recovery process is initiated)

= $(P_M^2 \times P_{COMM}) \times (P_{REC} \times P_{M\text{-}REC}^2 \times (1\text{-}P_{M\text{-}REC}))$.

Reliability of the system is obtained by calculating the probability of the system staying in the functioning state, State 1 and State 2, at checkpoint n. Let $D_i[n]$ be the probability that the system stays in State i at checkpoint n, where $1 \le i \le 3$ and $0 \le n$. Assume that the system is initially fault-free, which means the system stays at State 1 at checkpoint 0, so that

$D_1[0] = 1$, $D_2[0] = 0$, and $D_3[0] = 1 - D_1[0] - D_2[0] = 0$.

The transitions are expressed as recursive equations,

$D_1[n+1] = P_{11} \times D_1[n] + P_{21} \times D_2[n]$, and

$D_2[n+1] = P_{12} \times D_1[n] + P_{22} \times D_2[n]$.

For a system with fixed checkpoint periods, we use $R[n]$ to denote the reliability at the nth checkpoint. The reliability at checkpoint n is calculated as,

$R[n]$ = Prob (system in State1 at checkpoint n) + Prob (system in State2 at checkpoint n)

$= D_3[n] + D_2[n]$

$$= \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} D_3[n] \\ D_2[n] \end{bmatrix} = \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} P_{33} & P_{23} \\ P_{32} & P_{22} \end{bmatrix} \begin{bmatrix} D_3[n-1] \\ D_2[n-1] \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} P_{33} & P_{23} \\ P_{32} & P_{22} \end{bmatrix}^n \begin{bmatrix} D_3[0] \\ D_2[0] \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} P_{33} & P_{23} \\ P_{32} & P_{22} \end{bmatrix}^n \begin{bmatrix} 1 \\ 0 \end{bmatrix}. \qquad \text{(eq 1)}$$

A.2 Other Systems

In this section, the reliabilities of a simplex, a TMR without recovery, and a TMR-Simplex without recovery, are computed. These systems do not need checkpoints. However, for ease of comparison, we insert hypothetical checkpoints into computation processes. In this case, checkpoint periods are c_{normal} clock cycles. In Markov analysis a state transition happens at every checkpoint, that is, a state transition happens every c_{normal} clock cycles. Continuous time reliability analysis of these systems can be found in [Trivedi 82].

(1) Simplex:

A simplex system only contains a single copy of the computation, and no voting or error detection mechanism is added. P_M has the same definition as that in Sec. A.1.

$R[n]$ = Prob(the system is functioning from the beginning to checkpoint n) = P_M^n.

(2) TMR system without recovery:

In a TMR system without recovery, the original computing system is triplicated, and a voter is used to vote for output results. The state diagram is the same as Fig A.1. P_M has the same definition as that in Sec. A.1. We denote the probability that the output voter with area A_{VOTE} is fault-free during a checkpoint period as P_{VOTE} , and $P_{VOTE} = (1-p)^{A_{VOTE} \times c_{normal}}$.

Probabilities of transitions between states are:

$P_{11} = P_M^3 \times P_{VOTE}$.

$P_{12} = 3 \times P_M^2 \times (1\text{-}P_M) \times P_{VOTE}$.

$P_{21} = 0$, because of no recovery.

$P_{22} = P_M^2 \times P_{VOTE}$.

Reliability equation is the same as (eq 1), but with transition probabilities listed above.

(3) TMR-Simplex system without recovery:

A TMR-Simplex system without recovery switches from TMR to Simplex when faults are detected. The state diagram is the same as Fig. A.1. However, for TMR-Simplex we give states different definition:

State 1: This is the TMR state. All three copies and the voter/switch are fault-free.

State 2: This is the Simplex state. The simplex copy and the switch are fault-free.

State 3: The system fails. It happens when the voter or switch is faulty, or when two or more copies are faulty.

P_M has the same definition as that in (2). We denote the probability that the switch with area A_{SWITCH} is fault-free during a checkpoint period as P_{SWITCH} , and

$$P_{SWITCH} = (1-p)^{A_{SWITCH} \times c_{normal}} .$$

We denote the probability that the error detector with area A_{ED} is fault-free during a checkpoint period as P_{ED}. The error detector is used to detect the faulty copy to change TMR into Simplex configuration.

$$P_{ED} = (1-p)^{A_{ED} \times c_{normal}} .$$

Probabilities of transitions between states are:

$P_{11} = P_M^3 \times P_{VOTE} \times P_{ED} \times P_{SWITCH}$.

$P_{12} = 3 \times P_M^2 \times (1\text{-}P_M) \times P_{VOTE} \times P_{ED} \times P_{SWITCH}$.

$P_{21} = 0$.

$P_{22} = P_M \times P_{SWITCH}$.

Reliability equation is the same as (eq 1), but with transition probabilities listed above.

A.3 Synthesized Data for Reliability Calculation

Table A.1. Area data of the robot controller.

Item	CLB slices
Module (A_M) – Simplex, TMR w/o recovery (HR)	327
Module (A_M) – TMR w/ recovery (HR)	364
Single thread scheduler	4
Module (A_M) –TMR w/o recovery (MT)[3]	323
Module (A_M) – TMR w/o recovery (MT)	360
Input buffer (A_{INP})	16
Voter (A_{VOTE})	8
ECR block (A_{ECR})	17
Three-thread scheduler (A_{VOTE})	6
State restoration ($A_{RESTORE}$)	32
Register II ($A_{REG\text{-}II}$)	42
Error detector (A_{ED}) – TMR-Simplex	11
Switch (A_{SWITCH}) – TMR-Simplex	16

Table A.2. Clock cycles for the robot controller.

Item	Clock cycles
Task duration (c_{normal}) – HR	7
Task duration (c_{normal}) – MT	11
Recovery (c_{recov})	1

[3] The area is obtained by subtracting the single-thread scheduler area from the simplex module area.

Appendix C

Permanent Fault Repair for FPGAs with Limited Redundant Area

(An extended version of "Permanent Fault Repair for FPGAs with Limited Redundant Area," by Shu-Yi Yu and Edward J. McCluskey, *Proc. IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pp. 125-133, 2001.)

Permanent Fault Repair for FPGAs with Limited Redundant Area

Abstract

FPGA fault repair schemes remove faulty elements from designs through reconfiguration. In designs with high FPGA utilization, a sufficient number of routable fault-free elements may not be available for permanent fault repair. We present a new permanent fault repair scheme, in which the original design is reconfigured into another fault tolerant design that has smaller area, so the damaged element can be avoided. Three new schemes that fully utilize available fault-free area and provide low impact on availability are presented. Analytical results show that our schemes have an improvement on availability compared to a module removal approach, which removes a redundant module when it becomes faulty. Implementation results on a case study show that our scheme can effectively reduce design area; therefore, it can be used in situations when available FPGA area is limited because of permanent faults.

1. Introduction

Field-Programmable Gate Arrays (FPGAs) provide a solution to permanent fault repair in finer granularity because of their regular structure and reconfiguration capability. In FPGAs, a faulty module can be repaired by reconfiguring the chip so that a damaged configurable logic block (CLB) or routing resource is not used by a design. Many techniques have been presented to provide permanent fault removal for FPGAs through reconfiguration. One approach is to generate a new configuration after permanent faults are detected in computing systems. CAD tools and algorithms for fast FPGA re-routing and re-mapping were developed [Emmert 98] [Dutt 99]. Another approach is to generate pre-compiled alternative FPGA configurations and store the configuration bit maps in non-volatile memory, so that when permanent faults are present, a new configuration can be chosen without the delay of re-routing and re-mapping [Lach 99] [Huang 01a].

With the repair schemes mentioned above, permanent faults in an FPGA can be repaired as long as there are sufficient routable fault-free elements on the chip so that designs can avoid using faulty elements. However, for designs that use high percentage of FPGA resource, all routable fault-free elements could be exhausted. In addition, for very long mission times, routable fault-free elements can be exhausted due to multiple permanent fault repairs. In these cases, further permanent damage can no longer be repaired. To solve this problem, the design has to be reconfigured into a new fault tolerant design with smaller area, so that the permanent faults can be avoided. A traditional technique is to remove design elements at the module level, such as TMR/Simplex [Mathur 75] or self-purging redundancy [Losq 76]. However, a system with redundant modules removed may have much lower availability than the original system. For example, consider a TMR system with error detection and a duplex system, which is obtained by removing a module from TMR because of permanent faults. When a fault occurs in each system, the TMR system can still function, but the duplex system has to be stopped for recovery or repair. Hence, a duplex system has lower availability than a TMR system. In this paper, we present a new method of permanent fault repair in FPGAs to reduce impact on system availability by reconfiguring the original design into a new fault tolerant design that fully utilizes the reduced available fault-free FPGA area.

This paper is organized as follows. Section 2 defines the problem we are addressing. Section 3 presents three design methods for permanent fault repair. Section 4 describes an analytic model of a system with error recovery, and describes metrics we use to evaluate the designs. Section 5 shows the analytical results and compares these design methods. Section 6 describes the implementation of our scheme on a robot controller as a case study. Section 7 concludes the paper.

2. Problem Definition

The problem we are solving here is how to find a configuration for an FPGA-based design when available chip area is reduced because of permanent faults. The new configuration needs to have small availability reduction compared to the module removal approach. Since TMR is widely used in fault tolerant systems [Siewiorek 00] and can be

used for longer missions when it is implemented with recovery mechanisms [Yu 01], we focus on area reduction in TMR systems.

To explain the features required of an FPGA-based design, Fig. 1 illustrates the error recovery steps for FPGA-based systems. First, concurrent error detection (CED) mechanisms detect an error. If an error occurs for the first time, it is treated as a transient error; otherwise, it is treated as a permanent error. When a transient error occurs, the system recovers from corrupt data and resumes normal operation. When a permanent fault occurs, fault diagnosis is initiated to determine the location of the damaged resource, and a suitable configuration is chosen according to the available area. Then computation is resumed.

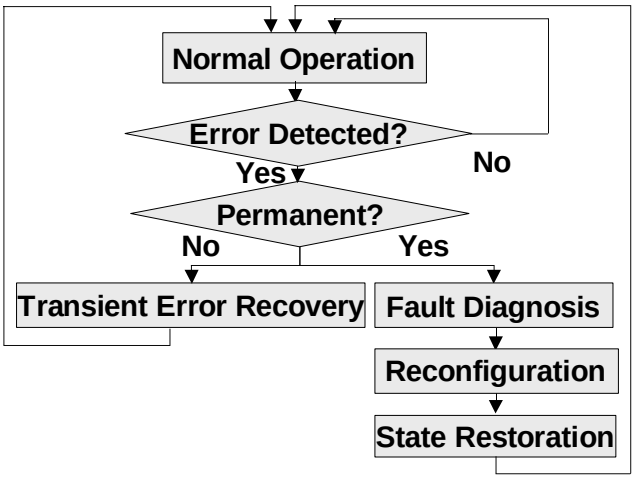


Figure 1. Error recovery in FPGAs.

Possible transient recovery schemes include *rollback* [Chandy 72] and *roll-forward* recovery [Long 90] [Pradhan 92] [Yu 01a]. In rollback recovery, the system state is backed up to some point in its processing, which is called a *checkpoint*. When an error occurs and is detected during operation, the system will restore its state back to the previous checkpoint and re-compute. However, re-computation has time overhead. In roll-forward recovery, the system will recover the corrupt data by copying correct data from fault-free redundant modules to the faulty module. Hence, re-computation delay is avoided.

From the error recovery flow, it is clear that CED and transient error recovery schemes are required for FPGA-based designs. We assume that fault diagnosis and

reconfiguration control mechanisms are available off-chip. These mechanisms can be implemented in another FPGA for fault tolerance and recovery purposes [Huang 01b]. Configuration files can be pre-compiled and stored in memory. States and data for recovery after permanent fault repair are stored by the other FPGA, too. CED and transient recovery mechanisms are built on-chip within an FPGA-based design. States and data for transient error recovery are handled by the design itself. In this paper, we present designs that have less area than a TMR but still have error recovery capabilities.

3. Design Candidates

We present three fault tolerant design techniques: (1) duplex with a checking block, (2) hybrid TMR-Simplex-CED, and (3) hybrid Duplex-Simplex-CED. Figure 2 (a) shows a TMR system with roll-forward recovery, and Figs. 2 (b), (c), and (d) illustrate the design techniques respectively. These designs all contain the features required for error recovery and are able to recover from single faults. We assume that a design can be partitioned into two portions. Unlike conventional fault tolerant designs, these designs are adjusted according to the available FPGA area. When a transient fault occurs, rollback or roll-forward recovery is used depending on location of faults and design structure. When a permanent fault occurs, an error indication is raised. The choice of rollback or roll-forward will be explained in the following paragraphs.

In a duplex system with a checking block, one of the original three modules in TMR is changed to a checking block. The checking block implements a portion of the normal function of a module, and the portion size depends on the available FPGA area. As shown in Fig. 2 (b), the original M3 is changed into block A, and the area saved is approximately the area of block B. The outputs of the checking block are compared with the outputs from blocks A in M1 and M2. The two modules, M1 and M2, are compared with each other. When there is a mismatch between the duplex modules, the module that agrees with the checking block is selected, and roll-forward is used to restore data in the other one. If there is no mismatch between the two modules but they disagree with the checking block, roll-forward is used to restore data in the checking block. When there is a mismatch between the duplex modules and they both agree with the checking block, rollback is used. This situation happens when faults occur in block B.

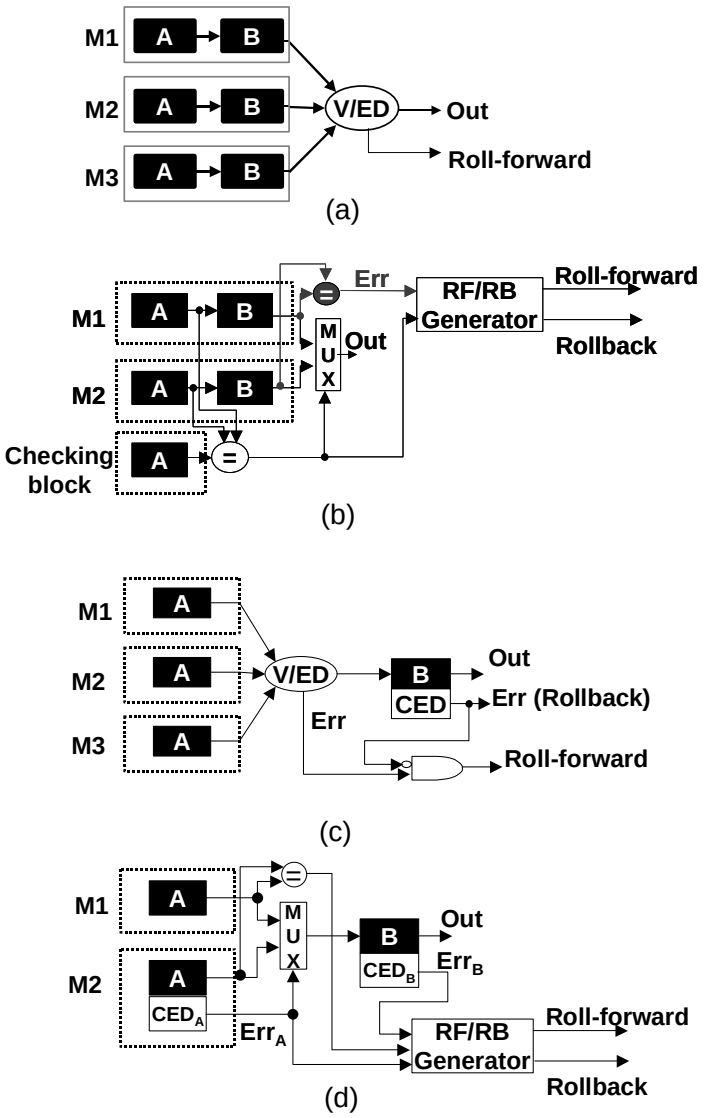


Figure 2: Designs with less redundant area than TMR. (a) The original TMR design. (b) A duplex system with a checking block. (c) A hybrid TMR-Simplex-CED design. (d) A hybrid Duplex-Simplex-CED design.

In a hybrid TMR-Simplex-CED design, we partition the original simplex design into two parts. As shown in the example of Fig. 2 (c), the design is partitioned into blocks A and B. A is triplicated; and CED, such as a parity checker or diversified duplication [Mitra 99], is added to B. A and B can have signals connected to each other. The area saved by the design compared with TMR is approximately the area of block B. The partition depends on the available FPGA area and design constraints. When an error

occurs in the partition with CED, rollback is used. Otherwise, when errors occur in the partition with TMR, roll-forward is used [Yu 01].

A hybrid Duplex-Simplex-CED design is shown in Fig. 2 (d). Block A is duplicated, and their outputs are compared to each other. CED block is built for block A in M2 and block B. The area saved is approximately the area of block B. Recovery approaches depend on CED results and the comparison between M1 and M2. When errors occur in block B with CED, rollback is used for error recovery. Otherwise, when there is a mismatch between the duplex and the CED block in M2 indicates an error, the results from block A in M1 will be used, and roll-forward is used to restore data in block A in M2. When there is a mismatch and the CED block in M2 does not indicate an error, the results from the block A in M2 will be used, and roll-forward is used to restore data in block A in M1.

We are going to compare our designs with a traditional module removal approach. For a TMR system, one approach to module removal for permanent fault repair is to remove the faulty module and change the system into a duplex [Jing 91]. Roll-forward can be used for transient error recovery in TMR, and rollback can be used in a duplex. We refer to the module removal approach as *TMR-Duplex* throughout the chapter. Figure 3 illustrates the idea of TMR-Duplex.

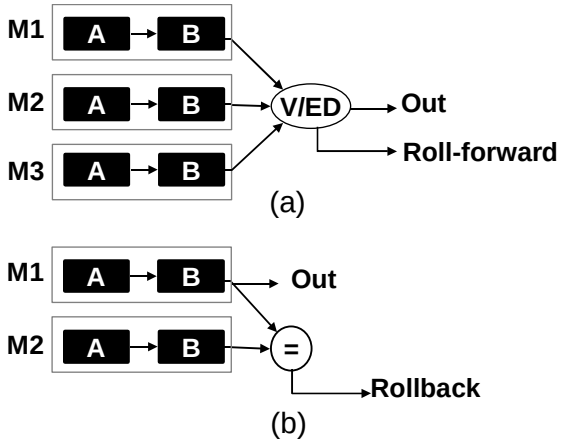


Figure 3: Permanent fault repair through a module removal approach, TMR-Duplex: (a) the original TMR design, and (b) duplex.

4. Evaluation Metric

Consider a computing system with both rollback and roll-forward recovery capabilities. When a transient fault occurs, the choice of rollback or roll-forward depends on the location of the fault. Errors are assumed to occur independently in each unit area in unit time. We use *rollback probability* to evaluate a design. It is the probability of initiating rollback recovery when a system is under normal function. It represents the overhead of recovery. For each rollback recovery, the computation process rollbacks to its previous checkpoint. Hence, the average re-computation time for each checkpoint is the product of the rollback probability and the time between two consecutive checkpoints. In a real-time application, a high rollback probability may not be acceptable, since a deadline may be missed because of re-computation. In our analysis, only single faults are considered because the probability of multiple faults is very small. We assume 100% single fault detection coverage for all CED techniques.

Referring to Figs. 2 (b) through (d), a rollback recovery will only occur if an error occurs in one of blocks B. Since we assume errors happen independently in unit area, rollback probabilities can be computed as probabilities of transient errors happening in blocks B, that is,

Rollback probability = Transient error probability per unit area unit time× blocks B area.

5. Comparison

In this section, we compute the rollback probabilities for different design techniques and compare them with the module removal approach, TMR-Duplex. Rollback probability depends on the total available FPGA area and the CED area overhead.

The rollback probabilities of the design techniques and TMR-Duplex are plotted in Fig. 4. We use 90% of CED overhead as an example. The effect of CED overhead will be described in the next paragraph. The probabilities are normalized to that of a duplex system. The area axis is normalized to the area of a simplex system. All designs fully utilize the available area except TMR-Duplex. In TMR-Duplex, when the available area is smaller than the area of a TMR, the design is changed into a duplex and its area is twice the simplex area.

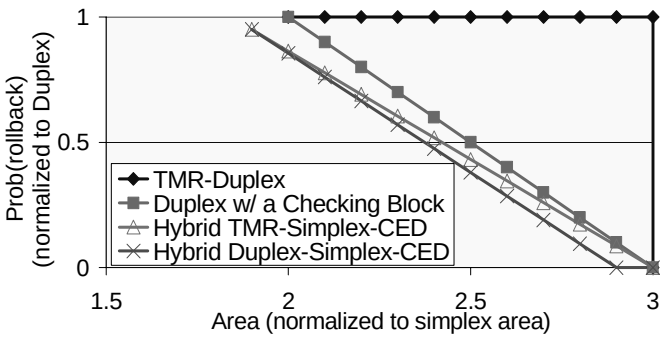


Figure 4. Rollback probabilities when CED overhead is 90%

As shown in Fig. 4, our approaches all have smaller rollback probabilities than TMR-Duplex. The relation of the rollback probabilities among the three design techniques and TMR-Duplex is

Hybrid Duplex-Simplex-CED < Hybrid TMR-Simplex-CED
< Duplex with a checking block < TMR-Duplex.

When the CED overhead is less than 100%, the relation should always be the inequality above. The reason is that if a design utilizes more portion of CED, when CED overhead is less than 100%, it can have more area for redundancy so that it can have lower rollback probability. From our designs, the hybrid Duplex-Simplex-CED always has the largest portion of CED, and the hybrid TMR-Simplex-CED has the second largest portion of CED. These two designs can reduce more area overhead than the duplex with a checking block when the CED overhead is smaller than 100%. The lower the CED overhead is, the more the differences among the approaches are. When the CED overhead is 100%, the three design approaches will have the same rollback probabilities since the area they need are the same. One example of 100% CED overhead is duplication, assuming that the area of comparators is negligible.

For CED overhead greater than 100%, the relation of the rollback probabilities among the three design techniques and TMR-Duplex is

Hybrid Duplex-Simplex-CED > Hybrid TMR-Simplex-CED
> Duplex with a checking block > TMR-Duplex.

6. Case study

In this section, we describe implementation of our permanent fault repair scheme in a case study.

We choose a robot controller as our case study, because robot controllers are used in many mission-critical applications. This is a continuous research of our ROAR project [ROAR 01]. In our previous works [Yu 00], we implemented the robot controller on a reconfigurable platform to demonstrate that reconfigurable platforms are suitable for reliable robot applications, and we have also implemented a roll-forward scheme on the robot controller for transient error recovery [Yu 01]. The controller is triplicated for fault tolerance purpose. A second-order linear algorithm is used as the control algorithm for the robot controller [Craig 89].

The controller consists of registers, a finite state machine, and arithmetic units for computation. For ease of implementation, we choose duplication as our CED technique for the controller. Since the three design techniques have the same rollback probability when duplication is used as the CED technique, we chose hybrid-TMR-Simplex-CED for our implementation. The module removal scheme, TMR-Duplex, is also implemented for a comparison purpose.

Figure 5 illustrates the hybrid TMR-Simplex-CED structure for the robot controller. A simplex robot controller is partitioned into blocks A and B. A is triplicated into A0, A1, and A2; and B is duplicated into B0 and B1 for concurrent error detection. In our design, input and output signals are connected to block A. For different designs, I/O does not have to always be connected to block A. Blocks A and B have signals connected to one another. The output signals of A are voted before connected to B. The output signals of B0 and B1 are compared with each other for error detection. When there is a disagreement between B0 and B1, rollback is initiated. Comparators are added to compare Out0, Out1, and Out2, with the final voted output Out for error detection purpose. If there is no disagreement between B0 and B1 but a disagreement is detected among final outputs, roll-forward is initiated. Error history is stored in registers. If an error occurs for a first time, a record will be stored in the error history registers, and the error will be treated as a transient error. If another error occurs within a certain cycles

after the previous error, it will be treated as a permanent error. When a permanent error occurs, an indication signal is raised for further fault diagnosis.

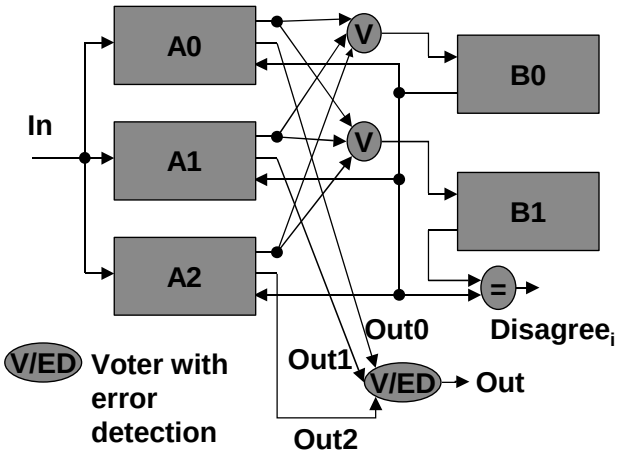


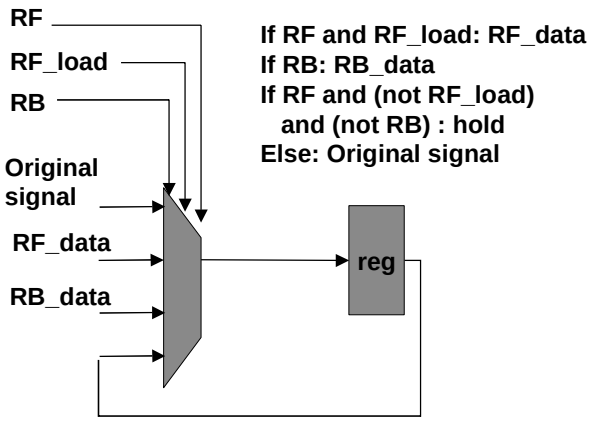
Figure 5. A hybrid TMR-Simplex-CED structure for the robot controller.

At every checkpoint, the critical states that are needed for rollback recovery are stored in triplicated registers. When rollback is initiated, the stored states are loaded into corresponding registers in blocks A and B, so that a computation process can be rolled back to its previous checkpoint. We use the *direct-load scheme* [Yu 01] for state restoration in roll-forward recovery. In the direct-load scheme, the registers that contain critical states in A1 are connected to those registers in A0 and A2, and the registers that contain critical states in A0 is connect to those in A1. Therefore, when A0, A1, or A2 becomes faulty, correct states are loaded from A1, A0, or A1, respectively.

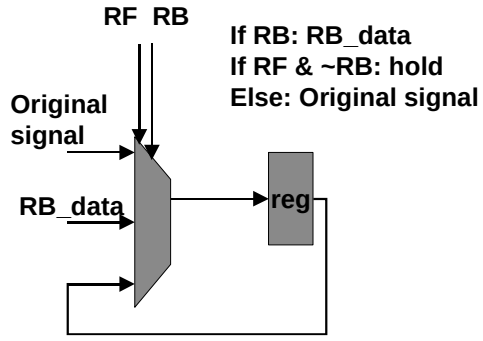
Figure 6 illustrates the state restoration in blocks A and B. As shown in Fig. 6 (a), both rollback and roll-forward mechanisms are built in each of block A0, A1, and A2. Signal *RF* indicates the roll-forward recovery. Signal *RF_load* indicates that this block is a faulty block and its states need to be loaded with correct values. RF and RF_load are generated from the disagreement results of the final output voters. Three RF_load signals are generated for A0, A1, and A2. When one of blocks A is faulty, the corresponding RF_load is asserted. RF is the ORed signal of the three RF_load signals. Signal *RB* indicates the rollback recovery. When there is disagreement between B0 and B1, RB is asserted. For block A, if both RF and RF_load are asserted, roll-forward recovery is initiated and data in this block needs to be restored. Hence, *RF_data*, the data from

another fault-free block, are loaded to the registers in the faulty block. If only RF is asserted, it means that roll-forward recovery is initiated but data in this block need not be restored. Hence, the data stored in the registers are loaded back to themselves. If RB is asserted, *RB_data*, which are data from the pre-stored data at the previous checkpoint, are loaded to the registers. If none of these signals are asserted, the controller is in the normal operation; hence, the original signals from computation are loaded into the registers.

Blocks B only need rollback recovery mechanisms. As shown in Fig. 6 (b), if RB is asserted, RB_data are loaded into the registers for rollback recovery. If RF is asserted, no recovery is needed in blocks B, so the values that are stored in the registers are loaded back to themselves. If none of them are asserted, the controller is operating normally. Hence, the original signals from computation are loaded to the registers.



(a)



(b)

Figure 6. State restoration in the robot controller: (a) block A, and (b) block B.

We partition the controller in a way to minimize signals connecting blocks A and B. The reason is that additional voters are needed for signals from blocks A to blocks B, and comparators are needed for signals from blocks B to blocks A.

The TMR-Duplex design can be implemented in two ways. We denote them as TMR-Duplex$_1$ and TMR-Duplex$_2$. TMR-Duplex$_1$ uses similar structure as self-purging redundancy [Losq 76], and both roll-forward and rollback recovery mechanism are implemented. The system switches from TMR with roll-forward recovery to Duplex with rollback recovery when a permanent fault occurs and is detected. TMR-Duplex$_2$ makes use of reconfiguration. The system changes from TMR with roll-forward recovery to Duplex with rollback recovery by changing the configuration of the FPGA.

The designs are synthesized with Xilinx XCV300 library. The area results are listed in Table 1, including area for Lookup Tables (LUT) and registers (REG). The columns of partition for TMR and Simplex-CED are area results before adding recovery logic. The numbers in parentheses are the area normalized to a simplex controller without recovery logic. The column of total area is the area including all recovery overhead.

Table 1: Synthesis area of different designs.

Design		Partition for TMR		Partition for Simplex-CED		Total	
		LUT	REG	LUT	REG	LUT	REG
TMR-Duplex$_1$	TMR with roll-forward	---	---	---	---	2821 (3.24)	1371 (3.56)
	Duplex with rollback	---	---	---	---	1992 (2.20)	1101 (2.63)
TMR-Duplex$_2$		---	---	---	---	3428 (3.79)	1568 (3.74)
Hybrid TMR-Simplex-CED		263 (0.29)	54 (0.13)	642 (0.71)	365 (0.87)	2581 (2.85)	1170 (2.79)

Compared with the original design, TMR with roll-forward, it is shown that the hybrid TMR-Simplex-CED design has smaller area. It means that the new design can effectively reduce design area. Therefore, it can be used when available FPGA area is limited. Compared with TMR-Duplex$_1$, although the new design has larger area than the Duplex with rollback recovery, but as shown in the previous section, the new design has

the advantage of smaller rollback probability. Compared with TMR-Duplex$_2$, the new design has smaller area, and it also has smaller rollback probability.

6. Conclusion

In this paper we presented three new permanent fault repair schemes for FPGA-based computing systems. When no routable fault-free elements are available for conventional permanent fault repair schemes, our schemes reconfigure the design into another fault tolerant design, which needs smaller area. We have examined three design structures to adapt to limited FPGA area and to provide reliability and availability. These designs are found to have improved availability compared to the module removal approach. Hence, they are more suitable for real-time applications where availability is a critical issue. The choice of designs depends on the CED overhead. Our repair scheme is implemented in a robot controller as a case study. The synthesis results show that our scheme can effectively reduce design area; therefore, it can be used when available area is reduced because of permanent faults.

References

[Chandy 72] Chandy, K. M. et al., "Rollback and recovery strategies for computer programs," *IEEE Trans. on Computers*, vol. C-21, No. 6, pp. 546-56, 1972.

[Dutt 99] Dutt, S., V. Shanmugavel, and S. Trimberger, "Efficient Incremental Rerouting for Fault Reconfiguration in Field Programmable Gate Arrays," *Digest of Technical Papers, IEEE/ACM Int. Conf. on Computer-Aided Design*, pp. 173-176, 1999.

[Emmert 98] Emmert, J. M., and D. Bhatia, "Incremental Routing in FPGAs," *Proc. of 11th Annual IEEE Int. ASIC Conf.*, pp. 217-221, 1998.

[Huang 00] Huang, W.-J., and E. J. McCluskey, "Transient Errors and Rollback Recovery in LZ Compression," *Proc. 2000 Pacific Rim International Symposium on Dependable Computing*, pp. 128-135, Los Angeles, CA, USA, 18-20 Dec. 2000.

[Huang 01a] Huang, W.-J. and E. J. McCluskey, "Column-Based Precompiled Configuration Techniques for FPGA Fault Tolerance," to appear in *FCCM'01*, 2001.

[Huang 01b] Huang, W.-J. and E. J. McCluskey, "A Memory Coherence Technique for Online Transient Error Recovery of FPGA Configurations," *9th ACM Int. Symp. On Field-Programmable Gate Arrays (FPGA'01)*, pp. 183-192, 2001.

[Jing 91] Jing, M.-H. and L.J.C. Woolliscroft, "A Fault-Tolerant Multiple Processor System for Space Instrumentation," *Int'l Conf. on Control*, Vol. 1, pp. 411-416, 1991.

[Lach 99] Lach, J., W. H. Mangione-Smith, and M. Potkonjak, "Algorithms for Efficient Runtime Faulty Recovery on Diverse FPGA Architectures", *DFT'99*, pp. 386-394, 1999.

[Long 90] Long, J., W., W. K. Fuchs, and J. A. Abraham, "A Forward Recovery Using Checkpointing in Parallel Systems," *Proc. of the 1990 Int. Conf. on Parallel Processing*, vol. 1, pp. 272-275, 1990.

[Losq 76] Losq, J. "A Highly Efficient Redundancy Scheme: Self-Purging Redundancy," *IEEE Trans. Comput.*, C-25, No. 6, pp. 269-78, 1976.

[Mathur 75] Mathur F. P. and P. DeSousa, "Reliability Modeling and Analysis of General Modular Redundant Systems," *IEEE Trans. Rel.*, R-24, No. 5, pp. 296-9, 1975.

[Mitra 99] Mitra, S., N.R. Saxena, and E.J. McCluskey, "A Design Diversity Metric and Reliability Analysis for Redundant Systems," *Proc. 1999 Int. Test Conf.*, pp. 662-671, 1999.

[Pradhan 92] Pradhan, D. K., and N. Vaidya, "Roll-forward Checkpointing Scheme: Concurrent Retry with Nondedicated Spares," *IEEE Workshop on Fault Tolerant Parallel and Distributed Systems*, pp. 166-74, 1992.

[Siewiorek 00] Siewiorek, D. P. and R. S. Swarz, *Reliable Computer Systems – Design and Evaluation*, 3rd Ed, Digital Press, 2000.

[Yu 00] Yu, S.-Y., N. Saxena, and E. J. McCluskey, "An ACS Robot Control Algorithm with Fault Tolerant Capabilities," *IEEE Symp. on Field-Programmable Custom Computing Machines*, pp. 175-84, Napa Valley, CA, USA, 16-19 April 2000.

[Yu 01] Yu, S.-Y. and E. J. McCluskey, "On-line Testing and Recovery in TMR Systems for Real-Time Applications," *ITC'01*, pp. 240-249, 2001.

Appendix D

Fault Tolerant Communication between Repair Controllers in the Dual-FPGA Architecture

(*CRC Technical Report 01-8*, "Fault Tolerant Communication between Repair Controllers in the Dual-FPGA Architecture," by Shu-Yi Yu and Edward J. McCluskey.)

Fault Tolerant Communication between repair controllers in the Dual-FPGA Architecture

Abstract

Redundant buses have been used to repair permanent faults in chip interconnects. However, redundant buses can significantly reduce bandwidth in chip interconnects. In this report, we present a new scheme to repair single faults in interconnects between two FPGAs. Our scheme eliminates the need of redundant buses and only needs one spare wire. Our scheme is applied in Stanford's Dual-FPGA architecture, which permits permanent fault repair without device replacement. We concentrate on the communication between repair controllers in the architecture. Our communication scheme uses parity checks for error detection. A new "command-bounce" diagnosis is presented to diagnose a faulty wire; repair is performed by reconfiguring both of the FPGAs. The limitations of the Dual-FPGA architecture are also discussed in the report.

1. Introduction

Permanent fault repair in reconfigurable hardware, such as Field-Programmable Gate Arrays (FPGAs), can be accomplished through reconfiguration [Emmert 98] [Dutt 99] [Lach 99] [Emmert 00] [Huang 01a]. In an FPGA-based design, when a computing element becomes faulty, the design can avoid using the faulty element by reconfiguration. Because a faulty chip need not be replaced but only reconfigured, FPGAs provide a more economical solution for permanent fault repair than conventional repair schemes [Siewiorek 73] [Losq 76].

Adaptive Computing Systems (ACS) use reconfigurable hardware for computation [Rupp 98]. Different dependable ACS architectures have been presented in the literature [Gokhale 98] [Saxena 00]. FPGAs are used in these architectures. When a fault occurs in an FPGA, it is repaired by reconfiguration. However, microprocessors are used in these architectures to control reconfiguration tasks. Because microprocessors are not reconfigurable, chip replacement is still needed for repair when permanent faults occur in microprocessors. To remove the need of device replacement, Stanford's ROAR project

developed a Dual-FPGA architecture, where the reconfiguration tasks for permanent fault repair are executed in FPGAs. The architecture comprises two FPGAs and memory, and no hard-core microprocessors are used [Mitra 00]. A similar two-FPGA structure was used in [Xilinx 00a] to provide recovery for transient errors in both configuration memory and user data. However, permanent fault repair is not discussed in [Xilinx 00a]. The Dual-FPGA architecture is capable of both transient error recovery and permanent fault repair. Unlike other ACS architectures, the conventional hard-core microprocessors are replaced by FPGAs. Therefore, chip replacement is no longer needed for permanent fault repair. Because of the flexibility of FPGAs, the aforementioned dependable ACS architectures can be mapped into the Dual-FPGA architecture by implementing soft-core microprocessors in FPGAs.

The Dual-FPGA architecture is shown in Fig. 1. Two FPGAs are used for task execution, and they monitor each other. Memory devices are used to store configuration files and user data. Repair controllers are embedded in each of the FPGAs to perform error monitoring, fault diagnosis, and reconfiguration tasks. When a permanent fault occurs in one of the FPGAs and is detected, the repair controller in the FPGA reports the fault to the controller in the other FPGA. The other controller evokes fault diagnosis for reconfiguration.

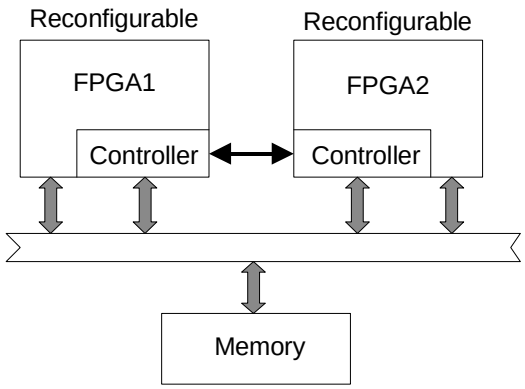


Figure 1: Dual-FPGA ACS Architecture.

The Dual-FPGA architecture is capable of repairing and recovering the system from all single faults occurred in the user application circuitry and memory. The system can also be repaired from most of the faults in configuration circuitry. However, there

exist single point failures in the architecture. These failures will be discussed in Sec. 4. Transient faults in the user application circuits can be recovered by implementing error detection and recovery mechanisms in user application modules. Transient faults in configuration circuitry can be recovered by downloading correct configuration. Permanent faults in the Dual-FPGA architecture can be categorized into *intra-chip faults* and *inter-chip faults*. Intra-chip faults are faults inside a single chip. They include faults in memory devices and in FPGAs. Inter-chip faults are faults occurring on wires connecting chips. These include faults in the bus between the FPGAs and the memory, the bus between the two FPGAs, and wires connecting the two repair controllers. For intra-chip faults, faults in FPGAs can be repaired through reconfiguration [Emmert 00] [Huang 01a]. The reconfiguration tasks are executed by the other FPGA. Faults in memory devices are corrected by error correcting codes or masked by using redundant devices. For inter-chip faults that occur in communication buses and wires between chips, redundant buses have been used in literature [Chan 00]. For faults in FPGA reconfiguration ports or the bus that is used for configuration data transmission, an alternative reconfiguration port can be used. For example, Xilinx Virtex FPGAs provide several reconfiguration modes that use different reserved pins for reconfiguration [Xilinx 01]. These different reconfiguration ports provide redundancy for tolerating permanent faults in the reconfiguration bus.

The use of a redundant bus for fault tolerant communication is generally expensive, however. Input and output pins of FPGAs are limited, and using redundant pins with a redundant bus can reduce the bandwidth to one half or one third, when duplication or triple modular redundancy (TMR) is used for inter-chip communication. Faults in communication between FPGAs and memory need to be tolerated by using redundant memory devices, since input and outputs of memory chips are not reconfigurable. For communication between the two FPGAs, the reconfiguration capability can be used in conjunction with fault tolerance techniques to reduce the number of redundant pins. In this report, we present a communication scheme that provides fault tolerance without the use of redundant buses. We concentrate on communication between repair controllers, because the fault tolerance of the repair

process needs to be guaranteed in the Dual-FPGA architecture. Our scheme can also be used for fault tolerant data communication between the two FPGAs.

This research is part of the ROAR project [ROAR 01]. In this report, we explain the function of the repair controllers in the dual-FPGA architecture, and present a new fault tolerance technique for the controllers. The report is organized as follows. Section 2 describes the repair process in the Dual-FPGA architecture and the functions of the repair controllers. Section 3 presents the fault tolerant communication scheme between the repair controllers. Section 4 discusses the limitation of the architecture and suggests possible solutions. Section 5 concludes the report.

2. Repair in Dual-FPGA Architecture

In this section, we describe the repair process in the Dual-FPGA architecture, and the role the repair controllers play in the repair process. We will show a block diagram of the Dual-FPGA architecture, and explain its functionality of each block in detail.

Figure 2 shows a general flow of error recovery for FPGA-based systems. Concurrent error detection (CED) mechanisms are built in functional units. When a transient error occurs, the system recovers from corrupt states and resumes normal operation. When a permanent fault occurs, a repair process needs to be initiated. In Fig. 2, the shaded block denotes the repair process. It includes three steps: fault diagnosis, reconfiguration, and internal state restoration. First, fault diagnosis is initiated to determine the location of the damaged resource. A suitable configuration is chosen according to the location of the fault. The FPGA is reconfigured, and system internal states are restored. Then the computation can be resumed.

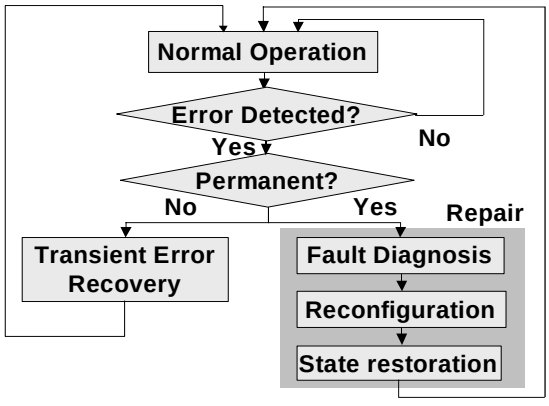


Figure 2: Error recovery in FPGAs.

2.1 Block Diagram

Figure 3 shows a block diagram of the Dual-FPGA architecture with a detailed internal diagram of one of the FPGAs. The other FPGA has an identical internal structure. This block diagram will be used to illustrate the repair process in the architecture. To simplify our discussion, we divide the repair tasks into different units and describe them individually. Each FPGA contains the following blocks: one or more user function units, a readback/writeback unit, a fault diagnosis unit, a configuration unit, and a repair controller. The communication between the two FPGAs is accomplished through the repair controller. The repair tasks can also be implemented in the repair controller.

In the FPGAs, the user function units perform user computation tasks. Error detection mechanisms are implemented in each of the user function units. The readback/writeback unit is responsible for correcting transient errors in configuration cells. To explain the repair process in the Dual-FPGA architecture, suppose a permanent fault occurs in one of the user function units in FPGA1. We use the subscript "1" to indicate a unit in FPGA1, and the subscript "2" to indicate a unit in FPGA2. The fault is detected by the error detection mechanism in the faulty unit, and an indication signal is raised. Repair $controller_1$ collects fault indication signals from all of the user function units, and receives the fault indication from the faulty unit. When repair $controller_2$ asks repair $controller_1$ about its current error status, repair $controller_1$ reports the fault. To repair FPGA1, repair $controller_2$ sends commands to fault diagnosis $unit_2$ to initiate diagnosis. Fault diagnosis $unit_2$ applies test patterns to FPGA1 and collects responses to locate the fault. When a fault is located, fault diagnosis $unit_2$ selects a proper configuration that avoids the fault, and sends the configuration selection to configuration $unit_2$. Configuration $unit_2$ generates proper address according to the configuration selection information, and reconfigures FPGA1. When FPGA1 is reconfigured, configuration $unit_2$ resets user function units in FPGA1. After reset, the computation processes in FPGA1 can be resumed.

The details of each block are explained in the following subsections.

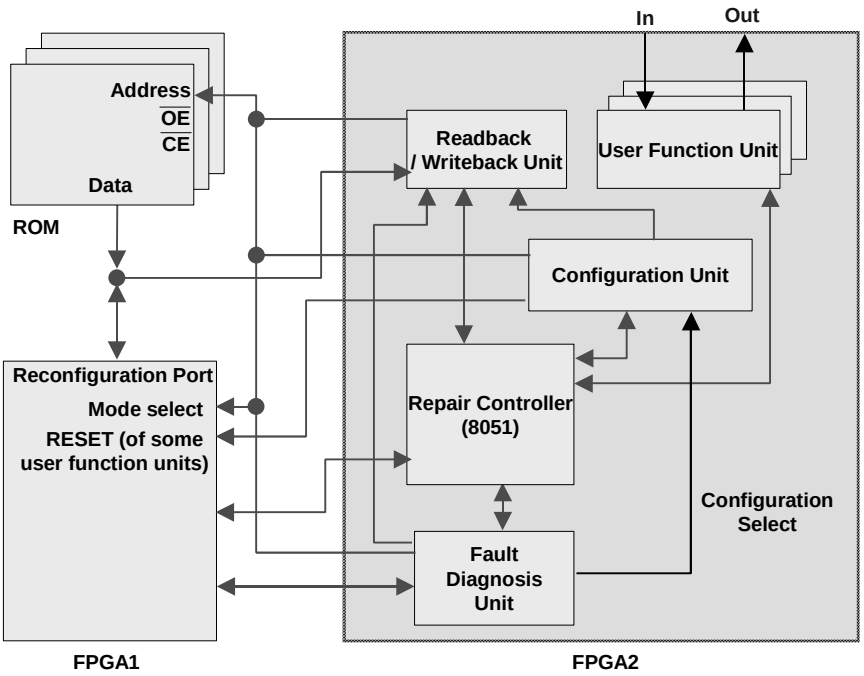


Figure 3: Block diagram of the Dual-FPGA architecture.

2.2 User Function Units

The user function units contain the logic functions of user applications, such as soft microprocessors, or co-processors that execute user computation tasks. There can be one or more user function units in an FPGA. Error detection and transient error recovery mechanisms are built in the user function units. When an error occurs and is detected for the first time, it is treated as a transient error. The faulty user function unit will be recovered by the built-in transient error recovery mechanism. If the error is detected again after the recovery process, it is treated as permanent. In this case, a signal *perm_ind*, that indicates a permanent fault, will be generated.

When a permanent fault occurs, a repair process is needed. After the repair process, computations are resumed. To resume the computations, internal states in user function units need to be restored. Therefore, user function units that need to be reconfigured have to store their internal states in memory devices. After the repair process completes, these internal states are loaded from memory devices so that the computations can be continued. We define the process of storing internal states in memory devices as *state restoration preparation*. The schemes for state restoration

preparation depend on the transient error recovery schemes used in user function units. Some user function units may use *checkpointing* for transient error recovery. In checkpointing, correct system states are backed up at a certain computation points called checkpoints. When a transient error occurs, the function unit will load correct states stored at its previous checkpoint for recovery. Some user functional units do not need checkpointing to store correct internal states. For example, systems with *Triple Modular Redundancy* (TMR) can obtain correct internal states from fault-free modules when one of the modules is faulty [Yu 01]. For user function units that use checkpointing, when a permanent fault is detected, these units will store their correct internal states that are backed up at their previous checkpoints in memory devices. For user function units that do not use checkpointing, these units will copy their correct internal states into memory devices for state restoration preparation. This process is controlled by either the repair controller or the function units themselves (see Sec. 2.6.) After the faulty FPGA is repaired, the user function units load their states back to the chip from memory devices, so the computations can be resumed.

Not all of the user function units need to abort their computations and store their internal states for a repair process. The need of state restoration depends on FPGA reconfiguration schemes. Current FPGAs support different reconfiguration schemes: *partial reconfiguration* [Xilinx 01], and *whole-chip reconfiguration.* Partial reconfiguration only configures a part of an FPGA, while whole-chip reconfiguration configures a whole chip during repair. During a repair process, if whole-chip reconfiguration is used, all of the user function units need to abort their operation. If partial reconfiguration is used, only the faulty unit is reconfigured. The faulty unit is isolated during reconfiguration so that its outputs will not affect the remaining part. Its operation is aborted, and internal states are stored in memory devices. The rest of the user function units continue their operation during reconfiguration.

2.3 Readback/Writeback Unit

Readback and *writeback* operations have been presented for detecting and correcting transient errors in FPGA configuration memory [Conde 99] [Carmichael 99]. Readback reads the current configuration out from the FPGA and compares it with stored

reference configuration files. Writeback writes the stored configuration back into the FPGA. Both readback and writeback can be performed at the same time as computation processes are executed in the FPGA. User functions do not need to stop during readback or writeback.

There are two major approaches for correcting a configuration error in FPGAs. One uses both readback and writeback [Carmichael 99], and the other only performs writeback, which is called *scrubbing* [Xilinx 00b]. For the readback/writeback approach, FPGA configuration data are read back for error detection. When an error is detected, the stored configuration is written back into the FPGA for error correction. For the scrubbing approach, the readback step is omitted, and the stored configuration is written back into the FPGA at a chosen interval, which depends on transient error rates. However, scrubbing writes the configuration cells more frequently than the readback/writeback approach. When a configuration cell is being written, a transient error in the writing circuit may be written into the cell. On the contrary, when a configuration cell is being read, errors in the writing circuitry will not change the content of the cell. Therefore, scrubbing has a higher probability of introducing an error to the configuration memory than the readback/writeback approach [Xilinx 00b]. These two approaches can recover transient errors in the configuration memory. However, they cannot repair systems from permanent faults when faults occur in configuration ports or in configuration circuitry that reads or writes configuration cells.

A readback/writeback unit performs the readback and writeback operation in the configuration memory of the other FPGA. The readback/writeback unit will temporarily stop its readback and writeback processes when fault diagnosis or reconfiguration is performed in the other FPGA. After the completion of fault diagnosis and reconfiguration, the readback and writeback processes will be started again. Different from previous readback/writeback approaches, if a mismatch occurs again right after writeback, the mismatch will be recorded. We define the reoccurred mismatch as a *persistent mismatch*. A signal called *persistent_mismatch* that indicates the second mismatch will be raised. A persistent mismatch may be caused by a permanent fault in the configuration port or in the configuration circuitry. Further diagnosis is needed to identify the location of the fault.

In Xilinx Virtex FPGAs, two different reconfiguration modes, SelectMAP™ and Boundary-Scan, are available to support readback [Xilinx 01]. These two modes use different input and output pins for reconfiguration. To provide fault tolerance for reconfiguration ports, the readback/writeback unit supports reconfiguration through both modes. Hence, when one of the reconfiguration ports becomes faulty, the reconfiguration can be accomplished through the other port. The current configuration mode is stored in a register. If an FPGA needs to be repaired, its readback/writeback unit will store the current configuration mode in memory devices before repair. The configuration mode will be loaded back after reconfiguration completes.

To provide fault tolerance for the readback/writeback unit, a concurrent error detection mechanism is added to the unit. A *perm_ind* signal will be raised when an error is detected multiple times.

2.4 Fault Diagnosis Unit

The fault diagnosis unit is responsible for fault location and configuration memory lookup. When a permanent fault occurs in the other FPGA, the fault diagnosis unit (in the fault-free FPGA) performs fault diagnosis to locate the fault. After the location is identified, a suitable configuration is chosen to avoid the fault. The selected configuration information is sent to the configuration unit for reconfiguration.

Different fault diagnosis algorithms can be used to locate faults in an FPGA. Most of the diagnosis schemes are *configuration-independent*, that is, the diagnosis processes locate faults in the entire FPGA no matter whether the faulty elements are used in the current configuration [Culbertson 97]. To reduce testing time, several *configuration-dependent* diagnostic schemes have been developed [Das 99] [Huang 01b]. Configuration-dependent diagnosis schemes only locate faulty elements that are used in the current configuration. For the scheme in [Huang 01b], additional error detection circuitry is built in each of the user function units. When a fault occurs, the error detection circuitry will identify a faulty unit. The faulty FPGA sends the faulty unit identification to the fault-free FPGA. The fault-free FPGA receives the identification and reconfigures the faulty FPGA to move the faulty unit to another place. In this scheme, no extra diagnosis patterns need to be applied; therefore, the diagnosis time overhead can be

significantly reduced. However, the faulty unit information needs to be transmitted from the faulty FPGA to the fault-free FPGA. This introduces timing overhead to the communication between the two FPGAs. The detail of the communication will be explained in Sec. 2.6.

During fault diagnosis, neither readback nor writeback can be performed. Hence, the fault diagnosis unit sends a signal to the readback/writeback unit indicating whether fault diagnosis is being performed.

To provide fault tolerance for the fault diagnosis unit, concurrent error detection such as duplication needs to be built in the unit. When an error in the fault diagnosis unit is detected multiple times, a *perm_ind* signal will be raised.

2.5 Configuration Unit

The configuration unit is responsible for configuring the other FPGA when permanent fault repair is needed. It accepts configuration commands from either the repair controller or the fault diagnosis unit. When both the repair controller and the fault diagnosis unit send configuration commands, it will accept the command from the fault diagnosis unit. The commands include reconfiguration requests and configuration selection information. According to the configuration selection information, the configuration unit generates proper addresses to access the configuration file. New configuration data are written into the faulty FPGA so that the faulty elements can be avoided.

The configuration unit is also responsible for resetting user function units when reconfiguration completes. Some function units may need to be reset so that they can start their normal function. For example, a finite state machine needs to be reset to its initial state. We use a special pin, RESET, for the reset purpose. When RESET is asserted in an FPGA, its function units are reset to their initial state. When reconfiguration completes, the configuration unit asserts RESET of the other FPGA for a certain period of time and de-asserts it, so that these function units can restart their function. In Xilinx Virtex FPGAs, there is no dedicated pin for RESET [Xilinx 01]. To provide fault tolerance to RESET, a redundant RESET pin can be used.

Similar to the readback/writeback unit, the configuration unit supports two different reconfiguration modes so that a permanent fault in the reconfiguration ports can be tolerated. The current configuration mode is stored in a register. If an FPGA needs to be repaired, its configuration unit will store the current configuration mode in memory devices before repair. The configuration mode will be loaded back after the reconfiguration completes.

During reconfiguration, neither readback nor writeback can be performed. Hence, the configuration unit sends a signal to the readback/writeback unit indicating whether configuration is being performed.

To provide fault tolerance, concurrent error detection, such as duplication, needs to be built in the configuration unit. When an error is detected multiple times, an indication signal *perm_ind*, that indicates a permanent fault, will be raised.

2.6 Repair Controller

The repair controller plays a critical role in the repair process. It is responsible for communication with the other FPGA. It is also responsible for controlling other units to perform repair tasks. To provide fault tolerance for the repair controller, it is duplicated and the outputs are compared. If a mismatch occurs, the controller's process is recovered by retry. If the mismatch occurs again after retry, a *perm_ind* signal that indicates a permanent fault will be raised.

Figure 4 illustrates the flow of the repair functions in the repair controller. These steps are explained as follows.

(1) Initialization.

After the reset of the FPGA is de-asserted, the repair controller initializes its e*rror status register*. This register stores the current error status of the FPGA. Its value is initialized to NO_ERR when the FPGA reset is de-asserted. If the controller detects a permanent fault indication among units in its FPGA, it sets the register to ERROR.

After register initialization, the repair controller reads the signals in the communication channel between the controllers, and determines whether the other repair controller has requested reconfiguration. This is used to repair faults in the

communication channel between repair controllers. Details of the repair process will be explained in Sec 3.3. If a reconfiguration request is received, the repair controller sends a command to the Reconfiguration Unit to initiate reconfiguration.

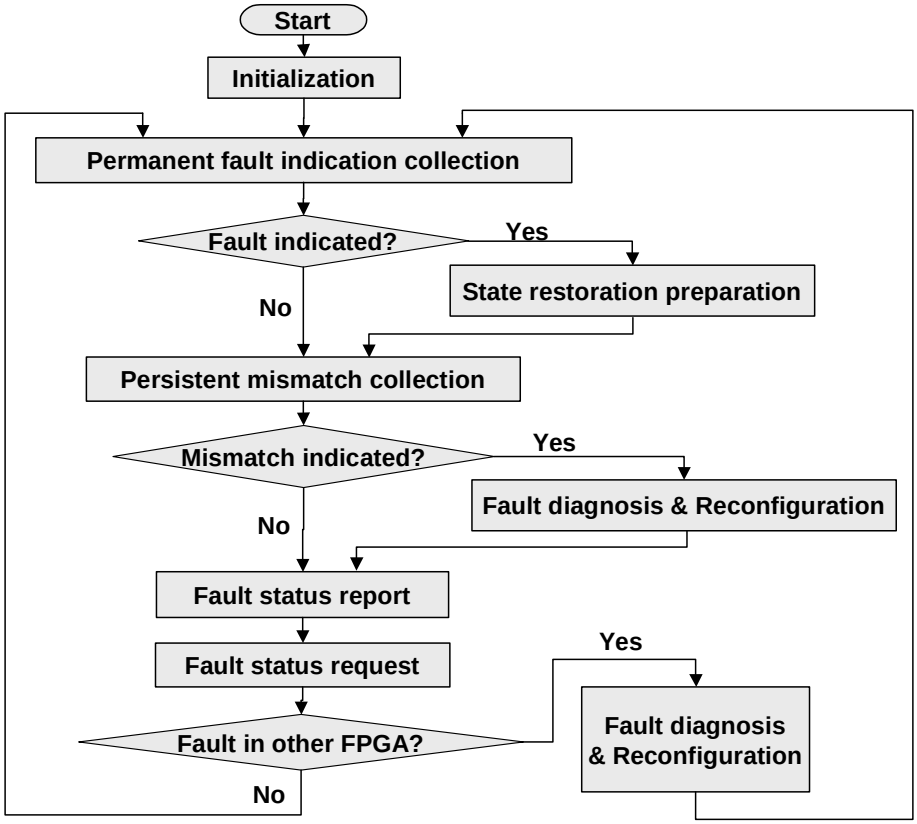


Figure 4: Repair function flow of the repair controller.

(2) Permanent fault indication collection.

The repair controller collects the *perm_ind* signals, which indicate permanent faults, from all of the user function units, the readback/writeback unit, the fault diagnosis unit, and the configuration unit from the same FPGA. It also collects *perm_ind* from the controller itself.

The methods for collecting *perm_ind* signals depend on the fault diagnosis schemes for the controller's FPGA. If the diagnosis scheme requires faulty unit identification, the repair controller needs to identify which one of the units indicates a permanent fault. If the diagnosis scheme does not require the faulty unit identification,

the repair controller only needs to collect all *perm_ind* signals and does not need to know what unit is faulty.

Figure 5 illustrates an example of implementing the permanent fault indication collection. Figure 5 (a) shows the implementation when faulty unit identification is needed. A multiplexor is used to select a *perm_ind* signal among different units. The repair controller sends unit ID (identification) to the multiplexor to select the user function unit from which it wants to collect *perm_ind*. The symbol $perm\_ind_i$ denotes the permanent fault indication signal from the i^{th} unit. When a permanent fault indication is received, the controller stores the current unit ID in a register and changes the error status register to ERROR. Figure 5 (b) shows the implementation when no faulty unit identification is needed. An OR gate is used to collect the permanent fault indication signals from all of the units in the same FPGA. As long as one of the function units indicates a permanent fault, the controller sets its error status register to ERROR.

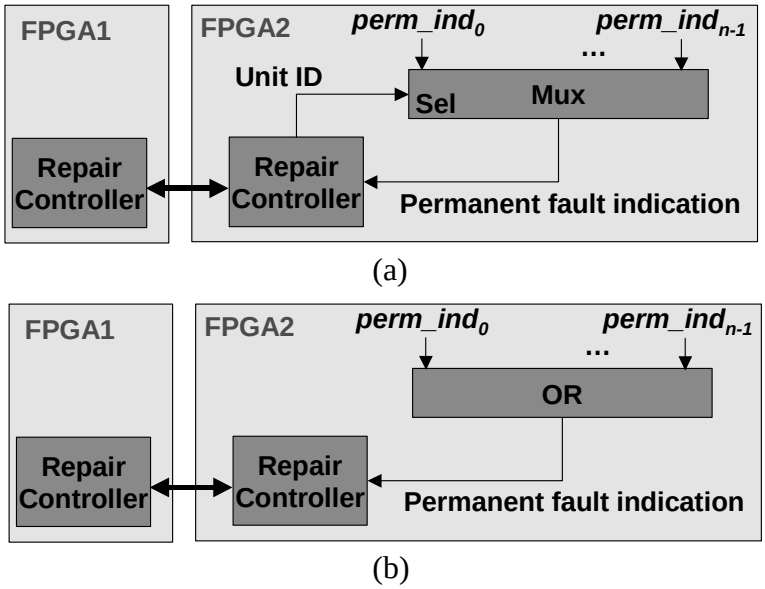


Figure 5: Permanent fault indication collection: (a) with faulty unit identification information, and (b) without faulty unit identification information.

(3) State restoration preparation.

State restoration preparation is defined in Sec. 2.2. For state restoration preparation, a repair controller's tasks depend on FPGA reconfiguration schemes. If whole-chip reconfiguration is used, the repair controller will notify all units for state

restoration preparation when it receives a permanent fault indication. If partial reconfiguration is used and only a faulty unit needs to be reconfigured, the repair controller does not need to notify other units for state restoration preparation. The faulty unit will start its own state restoration preparation process when its *perm_ind* is raised.

(4) Persistent mismatch collection.

The repair controller examines the *persistent_mismatch* signal, which is defined in Sec. 2.3, from the readback/writeback unit. When a persistent mismatch indication is received, the repair controller will initiate fault diagnosis to locate the fault. If the fault is in the configuration port, the repair controller requests both the readback/writeback unit and the configuration unit to switch to the other configuration port. If the fault is located in the configuration memory, a new configuration file will be loaded into the other FPGA and the faulty element will not be used.

(5) Fault status report.

Before explaining the fault status report, we list the basic commands that are used for the communication between the two controllers:

IDLE: a controller is not sending any request or any reply.

REQUEST: the command used for requesting error status from the other controller.

NO_ERR/CNFG: this command has two meanings. It can be used as NO_ERR, a reply to REQUEST when no error is recorded in the *error status register*; or as CNFG, a reconfiguration request. If the command is received during initialization, it is treated as CNFG; otherwise, it is treated as NO_ERR. The use of NO_ERR will be explained in detail in the following paragraphs, and the use of CNFG will be described in Sec. 3.3.

ERROR: the reply to REQUEST when an error is recorded in the *error status register*.

RETRY/DIAG: the command used for retry and diagnosis. When an error is detected in the communication channel, the repair controller at the receiving end sends RETRY/DIAG to request the other controller to resend its command. If an error is detected again after retry, the repair controller at the receiving end sends

RETRY/DIAG again to indicate the start of fault diagnosis. The details will be described in Sec. 3.1 and Sec. 3.2.

Now we are going to explain the fault status report between the repair controllers. To report the fault status, one of the repair controllers first reads the signals in the communication channel and determines whether it is REQUEST. If the received command is REQUEST, the repair controller will reply with its current error status. According to the content of the *error status register*, it will reply NO_ERR/CNFG when the content of the register is NO_ERR, and reply ERROR when the content of the register is ERROR. If faulty unit identification is required, the repair controller will send the faulty unit ID to the other controller following the ERROR reply. Serial transmission for the unit ID can be used to reduce pins needed between the controllers. If the received word is not REQUEST, the repair controller will skip reporting its fault status.

(6) Fault status request.

Before sending a fault status request, the repair controller first checks if the fault diagnosis unit is testing the other FPGA or if the configuration unit is reconfiguring the other FPGA. If the other FPGA is being diagnosed or reconfigured, the repair controller will not send a fault status request. If the other FPGA is neither diagnosed nor reconfigured, the repair controller will send the fault status request command, REQUEST, to the other repair controller. At the moment REQUEST is sent, the other controller may be executing other steps and may not respond immediately. The repair controller will wait for the response from the other controller. A *timer* is used to detect a *timeout*. A *timeout* occurs when the response is not received within a certain period of time, which depends on the control program in the repair controller. It can happen if the transmitted commands are lost during communication, or when a fault occurs in clock circuitry. If a timeout is detected, the repair controller will send REQUEST again. If a timeout is detected again, the repair controller will send a command to the fault diagnosis unit to initiate diagnosis.

When a reply is received from the other controller, the repair controller examines the received command and determines whether it is NO_ERR/CNFG or ERROR. If it is NO_ERR/CNFG, the repair controller sends an IDLE signal to indicate that the

communication channel is idle. If the received command is ERROR, depending on the fault diagnosis scheme, the repair controller may also receive the faulty unit ID from the other controller.

To summarize, Fig. 6 illustrates the fault status request and fault status report between the two controllers.

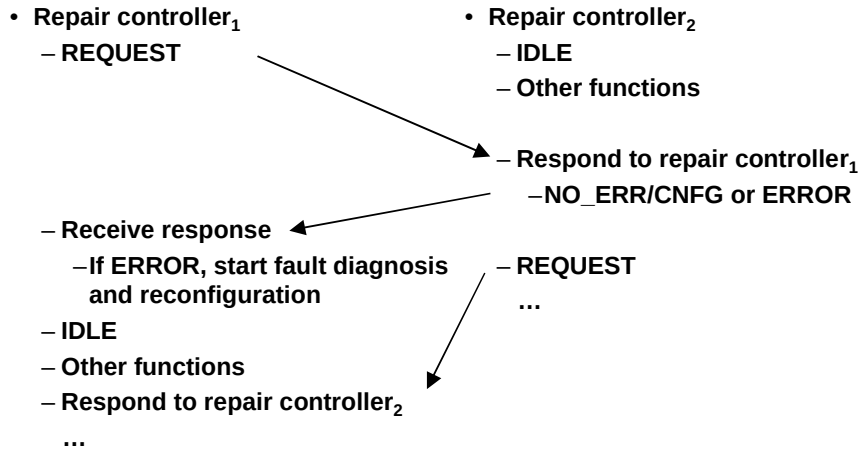


Figure 6: Fault status request and fault status report between the repair controllers.

(7) Fault diagnosis and reconfiguration.

When an error report, ERROR, is received from the other repair controller, the repair controller will start the repair process. First, it will send a command to the fault diagnosis unit to initiate fault diagnosis. If faulty unit identification is needed, it also passes the identification to the fault diagnosis unit. Then the repair controller waits until the fault diagnosis and the reconfiguration complete.

3. Fault Tolerant Communication Scheme between repair controllers

In this section we describe the fault tolerant communication scheme between repair controllers. To provide fault tolerance, three steps are needed (1) error detection in the communication channel, (2) fault diagnosis to find the faulty wire, and (3) reconfiguration to avoid the faulty wire. These steps are described in the following sections. The wires between the controllers are assumed to be bi-directional. Only single faults are considered in our scheme.

3.1 Error Detection

In this section, we first explain the codes we chose for the five commands described in Sec. 2.6, and then explain the error detection flow in the communication channel between the repair controllers.

Many error detection codes have been developed in literature [Wicker 95]. Examples include Hamming codes, cyclic redundancy check (CRC) codes, and parity checks. These codes are used in different applications. Hamming codes have been used in networking for error detection [DEC 80] but they are better known for their error correction capabilities [Furutani 89]. CRC codes use shift registers and XOR gates to detect errors in multiple bits. They are suitable for error detection when the transmitted messages are long. Hence, they are mostly used in networking protocols, such as Ethernet and IEEE 802.5 [Peterson 00]. Parity checks are widely used in many applications such as computation logic [Siewiorek 00] because they are simple to implement and they can effectively detect single errors inside logic modules. Multiple dimension parity checks are also used in networking [Peterson 00].

In our communication scheme, we use a special code for error detection. Unlike most of the error detection codes, the codewords we use have different *Hamming distances* from one another. *Hamming distance* is the number of different bits between two words [Wicker 95]. Distance 2 provides single error detection capabilities, while distance 3 provides single error correction capabilities.

Figure 7 shows the Hamming distance among the five commands described in Sec. 2.6. To explain the choice of the Hamming distance, we divide the commands into two sets according to when they are used:

Set_1: {REQUEST, NO_ERR/CNFG, ERROR, IDLE}. Set_1 is the commands that are used during normal fault status requests and fault status reports, when no faults are present in the communication channel.

Set_2: {REPLY/DIAG, NO_ERR/CNFG}. Set_2 is the commands that are used for fault tolerance purposes, when a fault occurs in the communication channel. NO_ERR/CNFG is included in both of the sets because it can be used during both normal functions (NO_ERR) and when a fault occurs (CNFG).

We need to detect single errors in Set_1. Because parity checks are easy to implement, we chose parity checks for the error detection scheme for commands in Set_1. To assign the same parity to every of the four commands, the commands need to have even Hamming distance from one another.

For the commands in Set_2, the code needs to provide not only single error detection capabilities but also single error correction capabilities. The reason is that these two commands need to be distinguished from all other commands even when a fault is present in the communication channel. Otherwise, a fault could corrupt the commands so the commands would not be understood by the other controller. To have single error correction capabilities, these two commands need to have at least Hamming distance 3 from all other commands.

From the discussion in the previous two paragraphs, we assign REQUEST, ERROR, and IDLE to have Hamming distance 2 from one another. RETRY/DIAG is assigned to have distance 3 from any other commands. For NO_ERR/CNFG, because its Hamming distance from other commands need to be an even number and at least 3, we choose it to have distance 4 from any other commands.

Table 1 lists an example of the codeword that satisfies the Hamming distance requirement.

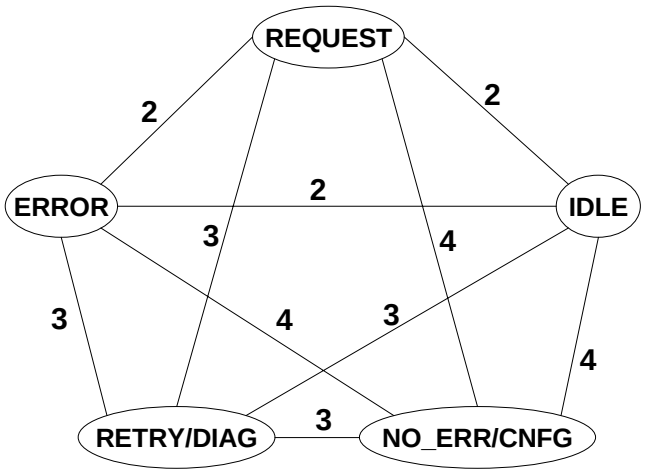


Figure 7: Hamming distance among the commands.

Table 1. An example of the codeword that satisfies the Hamming distance requirement.

Command	Codeword
REQUEST	11001
NO_ERR/CNFG	00100
ERROR	11010
IDLE	11111
RETRY/DIAG	00011

Figure 8 illustrates the flow of error detection for the communication between the repair controllers. When a repair controller receives a command from the other controller, it first determines whether it is RETRY/DIAG. If RETRY/DIAG is received, it indicates that the previous command the controller sends out is corrupt. Therefore, the controller resends the previous command. If RETRY/DIAG is received again, diagnosis will be needed. The box Diagnosis-Tx will be explained in Sec. 3.2. If the received command is not RETRY/DIAG, the repair controller performs parity checks to detect if there is a parity violation. If no violation is detected, the controller will process the received command. If parity violation is detected, the controller sends RETRY/DIAG to the other controller to ask for retry. If parity violation is detected again after retry, diagnosis will be needed. The box Diagnosis-Rx will be explained in Sec. 3.2.

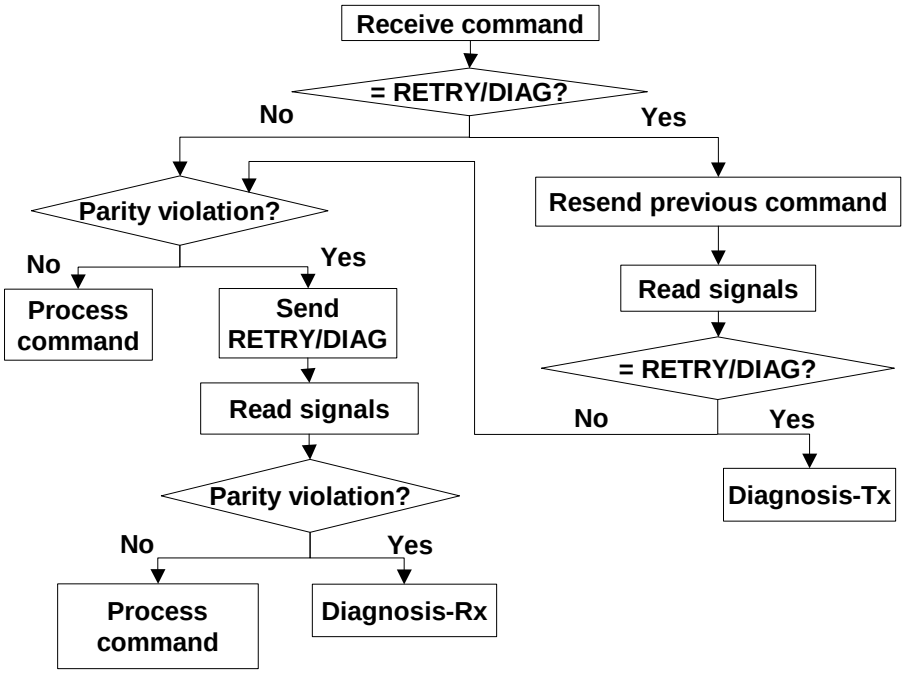


Figure 8: Error detection flow for the communication between the repair controllers.

3.2 Interconnect Diagnosis

Different techniques have been presented to test and diagnose interconnect between chips. Most of the techniques rely on boundary scan [Wagner 87] [Wang 89] [Chan 92] [Feng 95]. These techniques scan test patterns into scan registers and collect test responses by scanning data out. Other techniques replace chips with bus emulators to apply test patterns and collect test responses [Hsu 96]. However, all of the schemes mentioned above need more than one test pattern. In this section, we present a new diagnosis scheme for the interconnection between the repair controllers in the Dual-FPGA architecture. Instead of testing interconnects off-line and applying multiple patterns, we apply diagnosis on-line and use only one test pattern.

The scheme we present is called *command-bounce diagnosis*. It uses corrupt commands as test patterns for diagnosis. Suppose a permanent fault occurs in the communication channel and is detected as described in Sec. 3.1. When a permanent fault is detected, a repair controller can be performing one of two roles: it can be the controller that receives the corrupt command and detects the fault, or the controller that sends the corrupt command. Referring to Fig. 8, if the controller receives the corrupt command, it will go to the Diagnosis-Rx step. If the controller sends the corrupt command, it will go to the Diagnosis-Tx step. We will discuss Diagnosis-Rx and Diagnosis-Tx in the following two paragraphs.

For Diagnosis-Rx, the repair controller will resend RETRY/DIAG to the other controller to notify that the diagnosis process is going to start. Following the RETRY/DIAG command, the repair controller bounces (resends) the corrupt command back to the other controller as the test pattern.

For Diagnosis-Tx, the repair controller has received RETRY/DIAG that indicates the start of diagnosis. The repair controller reads one command in the communication channel followed by RETRY/DIAG and compares it with the previous command it sends out.

To explain the comparison, suppose there are n bits in the communication channel. One of the repair controllers sends out a command **a**, where

$$\mathbf{a} = (a_0 \quad \cdots \quad a_i \quad \cdots \quad a_{n-1}),$$

and the symbol a_k indicates the k^{th} bit in the command **a**, k = 0, 1, .. n-1. Assume an error occurs in the i^{th} bit. The data in the i^{th} bit is corrupt, and the other controller receives a corrupt command $\tilde{\mathbf{a}}$, where

$$\tilde{\mathbf{a}} = \left(a_0 \quad \cdots \quad \overline{a_i} \quad \cdots \quad a_{n-1}\right).$$

To compare these two commands, the controller at the receiving end performs a bit-wise XOR to the two commands:

$$\mathbf{a} \oplus \tilde{\mathbf{a}} = \left(a_0 \oplus a_0 \quad \cdots \quad a_i \oplus \overline{a_i} \quad \cdots \quad a_{n-1} \oplus a_{n-1}\right) = \left(0 \quad \cdots \quad 1 \quad \cdots \quad 0\right).$$

$\uparrow$ the i_{th} bit

As shown in the equation above, all of the bits in the XOR result are 0 except the i^{th} bit. Hence, the faulty bit can be identified.

3.3 Reconfiguration

To repair faulty interconnection economically, we use reconfiguration as our repair scheme. Unlike the repair of faulty elements inside an FPGA [Emmert 98] [Huang 01a], the repair process for faulty wires connecting FPGAs requires reconfiguration of multiple FPGAs. Figure 9 illustrates the reconfiguration process. The two FPGAs are connected to each other through the communication channel. As shown in Fig. 9 (a), pins of FPGA1 are connected to pins of FPGA2 through wires. Spare wires are needed to replace the function of faulty wires. Suppose a permanent fault occurs in a wire connecting the two FPGAs, and is detected and located by the repair controller in FPGA1. To repair the system, we need to reconfigure both FPGAs to avoid the faulty wire. In Fig. 9 (b), FPGA1 reconfigures FPGA2, so that the output that is originally connected to the faulty wire is moved to connect a spare wire. After the reconfiguration, in Fig. 9 (c), FPGA2 reconfigures FPGA1, so that the output that is originally connected to the faulty wire is moved to connect the spare wire.

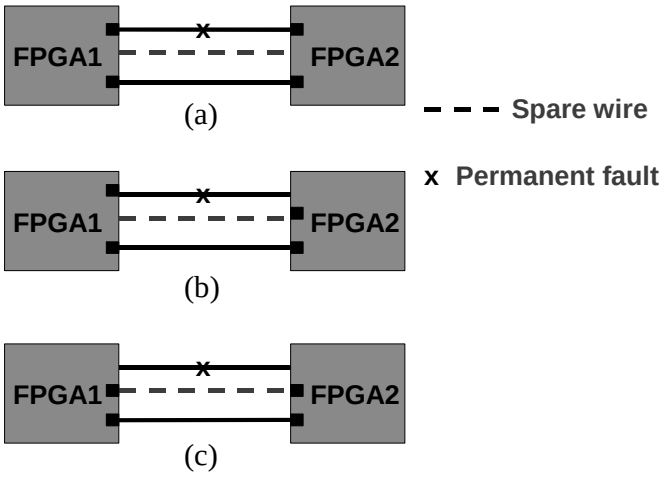


Figure 9: Reconfiguration process when a fault occurs in inter-chip communication: (a) original configuration, (b) FPGA2 is reconfigured, and (c) FPGA1 is reconfigured.

To perform reconfiguration, a repair controller can be in two different roles. It may be the repair controller that locates the fault and reconfigures the other FPGA first, such as the repair controller in FPGA1 in the example of Fig. 9, or the repair controller that is reconfigured first, such as the repair controller in FPGA2 in the example of Fig. 9. We denote the repair controller in FPGA1 as *repair controller*$_1$ and the repair controller in FPGA2 as *repair controller*$_2$. Figure 10 shows the reconfiguration flow for repair controller$_1$. Figure 11 shows the reconfiguration flow for repair controller$_2$.

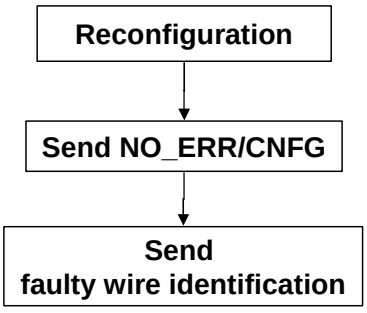


Figure 10: Reconfiguration flow for repair controller$_1$.

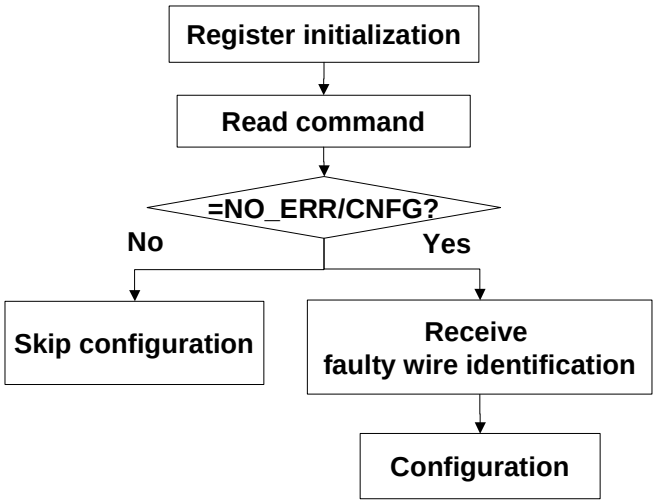


Figure 11: Reconfiguration flow for repair controller$_2$.

When repair controller$_1$ locates the interconnect fault, as shown in Fig. 10, the first step is to reconfigure FPGA2. Repair controller$_1$ sends configuration commands to its configuration unit to initiate reconfiguration. When the reconfiguration completes, repair controller$_1$ sends the reconfiguration request, NO_ERR/CNFG, to repair controller$_2$. Following NO_ERR/CNFG, it also sends the faulty wire identification to repair controller$_2$ so that repair controller$_2$ can reconfigure FPGA1 according to the identification information.

Repair controller$_2$ does not perform reconfiguration tasks until FPGA2 is reconfigured. As shown in Fig. 11, after initializing its registers, repair controller$_2$ reads a command in the communication channel and determines whether it is NO_ERR/CNFG. If NO_ERR/CNFG is received, repair controller$_2$ reads the faulty wire identification in the communication channel. After that, it sends configuration commands to its configuration unit and waits for the completion of the reconfiguration. If the received command is not NO_ERR/CNFG, repair controller$_2$ will skip the reconfiguration and starts its normal function, as described in Sec. 2.6.

The faulty wire identification needs to be coded so that every word can be distinguished from one another even when a wire is mismatched between the two repair controllers. To achieve this, we encode the identification information with the Reduced (6, 3) Hamming code that provides Single-Error-Correcting-Double-Error-Detecting (SEC-

DED) capability. Because there are only 5 wires in the communication channel, we replicate the last bit of the Hamming code 4 times to make it two words. Table 2 lists the codeword we use for the faulty wire identification.

Table 2. Codeword for the faulty wire identification.

Faulty wire	First word	Second word
1	00000	00000
2	00110	11111
3	01011	00000
4	01101	11111
5	10011	11111

3.4 Comparison

In this section, we are going to compare our scheme with the conventional scheme that uses redundant buses for fault tolerance.

Assume two repair controllers are connected to each other through a communication channel. There are totally n commands can be sent through the communication channel. Assume that each command is sent in one clock cycle, and one parity bit is used for error detection. Therefore, at least $\lceil \log_2(n) \rceil+1$ wires are needed.

For a fault tolerant scheme that uses redundant buses, the total number of wires needed is

$$2 \times (\lceil \log_2(n) \rceil+1).$$

For our communication scheme, we need at least $\lceil \log_2(n) \rceil+1$ wires to transmit n commands and the parity information. In addition, one spare wire is needed. Two special commands, RETRY/DIAG and CNFG, are added in the scheme. As described in Sec. 3.1 and 3.3, these two commands need to have at least Hamming distance 3 between all other commands. Therefore, additional wires are needed to transmit these two commands. Assume l wires are added. We need to find the lower bound of l. The total number of wires needed is

$$\lceil \log_2(n) \rceil + 1 + \text{one spare wire} + l = \lceil \log_2(n) \rceil + 2 + l.$$

For the ease of discussion, we denote a word as

$$(i_0 \quad i_1 \quad \cdots \quad i_{k-1} \quad a_0 \quad \cdots \quad a_{l-1} \quad p),$$

where $i_0 \sim i_{k-1}$ are the information bits, $a_0 \sim a_{l-1}$ are the additional bits, and p is the parity bit.

Consider the worst case, $n=2^k$. In this case, no matter what word we choose for the two special commands, we can always find a command that have the same information bits, $i_0 \sim i_{k-1}$, as one of the special commands. Let us denote the two special command as ***s1*** and ***s2***, where

$$\boldsymbol{s1} = \left(i_0^{S1} \quad i_1^{S1} \quad \cdots \quad i_{k-1}^{S1} \quad a_0^{S1} \quad \cdots \quad a_{l-1}^{S1} \quad p^{S1} \right), \text{ and}$$

$$\boldsymbol{s2} = \left(i_0^{S2} \quad i_1^{S2} \quad \cdots \quad i_{k-1}^{S2} \quad a_0^{S2} \quad \cdots \quad a_{l-1}^{S2} \quad p^{S2} \right).$$

The command that has the same information bits as ***c1*** and ***c2***, where

$$\boldsymbol{c1} = \left(i_0^{S1} \quad i_1^{S1} \quad \cdots \quad i_{k-1}^{S1} \quad a_0^{C1} \quad \cdots \quad a_{l-1}^{C1} \quad p^{C1} \right), \text{ and}$$

$$\boldsymbol{c2} = \left(i_0^{S2} \quad i_1^{S2} \quad \cdots \quad i_{k-1}^{S2} \quad a_0^{C2} \quad \cdots \quad a_{l-1}^{C2} \quad p^{C2} \right).$$

To provide Hamming distance 3, we need ***s1*** to have at least three bits different from ***c1***, and ***s2*** to have at least three bits different from ***c2***. In addition, ***s1*** needs to have at least three bits different from ***s2***. We can obtain Hamming distance 3 between ***s1, s2*** and ***c1***, ***c2*** by assigning

$$p^{S1} = \overline{p^{C1}}, \; p^{S2} = \overline{p^{C2}}$$

$$a_0^{S1} = \overline{a_0^{C1}}, \; a_0^{S1} = \overline{a_0^{C1}},$$

$$a_1^{S1} = \overline{a_1^{C1}}, \text{ and } \; a_1^{S1} = \overline{a_1^{C1}}.$$

If we choose all of the n commands to have the same a_0 and a_1 as ***c1*** and ***c2***, then all of these commands will have at least Hamming distance 3 from ***s1*** and ***s2***.

We can obtain Hamming distance 3 between ***s1*** and ***s2*** by choosing their corresponding information bits so that the Hamming distance between the information bits is 3. In this case, l needs to be at least 2. However, if the total number of the information bits, k, is smaller than 3, then this cannot be achieved. In this case, one more bits needs to be added, and l needs to be at least 3.

The total number of wires needed is

$$\lceil \log_2(n) \rceil + 2 + l.$$

If $\lceil \log_2(n) \rceil < 3$, $l = 3$. If $\lceil \log_2(n) \rceil \geq 3$, $l = 2$.

Compared with the conventional scheme, our scheme uses fewer wires when

$$\lceil \log_2(n) \rceil + 2 + l < 2 \times (\lceil \log_2(n) \rceil + 1).$$

Solving the equation above,

$$n > 4.$$

Therefore, our scheme is more suitable when the total number of commands is greater than four, not including the special commands RETRY/DIAG and CNFG. The more commands are needed in the communication, the more wires our scheme can reduce compared to the conventional scheme.

4. Limitation and Discussion

Although the Dual-FPGA architecture can be repaired from most of the failures caused by single faults, it has some limitations. In this section, we will discuss the limitation of the architecture and suggest possible solutions.

(1) Memory contention: if the two FPGAs have access to the same memory device, the faulty FPGA may prevent the fault-free FPGA from accessing the memory. To prevent this, one solution is to assign different memory devices to different FPGAs, so that the two FPGAs do not share the same memory chip.

(2) Communication between FPGAs and memory devices: The discussion of our fault tolerant communication scheme focused on the communication between repair controllers. It can also be applied to data communication between the two FPGAs. However, this scheme is not suitable for the communication between an FPGA and memory devices. For the communication between an FPGA and a memory device, redundant memory chips and buses are needed. The reason is that current commercial memory devices do not have any fault tolerance capability to repair faults in input and output pins. Although error-correcting codes are implemented in memory devices, they have not been implemented in input or output pins. In addition, there is no redundant input and output pin for memory devices. Therefore, if one of the pins becomes faulty, the whole memory device needs to be removed from the system. Connecting to redundant memory devices

and redundant buses reduces the available pins of an FPGA. One possible solution is to use FPGAs as memory devices and store data in the block RAMs and select RAMs. However, in current technology, the memory capacity of FPGAs is still limited (around the order of mega bits per chip), and it is not practical to use current commercial FPGAs to store configuration files.

(3) Single point failures: There are a few pins in FPGAs reserved for reconfiguration purposes. These pins are not reconfigurable. For most of the special purpose pins, their faults can be tolerated by changing to another reconfiguration mode. However, there are a few pins that are needed in all of the configuration modes. Therefore, when they become faulty, their functions cannot be replaced by other pins. For example, in Xilinx Virtex FPGAs, pins PROGRAM, DONE, and INIT are used in all of the configuration modes. Single point failures will occur if faults occur in one of these pins.

(4) Power up: in our repair scheme, SelectMAP™ and Boundary-Scan are chosen as the reconfiguration modes. The reason is that among the four available reconfiguration modes in Xilinx Virtex FPGAs, only these two support the readback mechanism. However, these two modes cannot be used at system startup for the Dual-FPGA architecture. The reason is under these two modes, an FPGA needs external hardware to reconfigure itself. During system power-up, both of the FPGAs have blank configurations and cannot perform configuration tasks for the other FPGA. One solution is to use an external non-reconfigurable controller to configure both FPGAs during system startup. Another solution is to set one of the FPGAs to Master Serial Mode and the other to Slave Serial Mode for configuration. The two FPGAs are chained together, and the FPGA in the Master Serial mode will read out data from memory devices to configure both FPGAs. An external hardware is needed to set the initial reconfiguration mode of the two FPGAs to Master Serial Mode

and Slave Serial Mode, respectively. Both of the two solutions require external hardware for configuration at system power-up. To provide fault tolerance for the external configuration hardware, duplication or TMR can be used.

5. Conclusion

In this report, we have presented a fault tolerant communication scheme between the repair controllers in Stanford's Dual-FPGA Architecture. The scheme is capable of repairing the system from failures caused by single faults in the communication channel. Unlike conventional fault tolerance schemes for communication, our scheme takes advantage of flexibility of FPGAs and does not need to use redundant buses for permanent fault repair. Therefore, the number of the needed wires is significantly reduced. The repair process includes three steps: (1) error detection, (2) fault diagnosis, and (3) reconfiguration. Parity checks are used for error detection. A diagnosis scheme, the command-bounce diagnosis, was presented in the report. Unlike other diagnosis schemes, command-bounce diagnosis only requires to apply one test pattern to identify the faulty wire. Repair is performed by reconfiguring both of the FPGAs. The limitations of the Dual-FPGA architecture have also been discussed in the report.

References

[Abramovici 90] Abramovici, M., M. A. Breuer, and A. D. Friedman, *Digital Systems Testing and Testable Design*, Revised Printing, IEEE Press, 1990.

[Carmichael 99] Carmichael, C., E. Fuller, P. Blain, and M. Caffrey, "SEU Mitigation Techniques for Virtex FPGAs in Space Applications," http://www.xilinx.com/products/hirel_qml.htm, 1999.

[Chan 92] Chan, J. C., "An Improved Technique for Circuit Board Interconnect Test," *IEEE Trans. on Inst. and Meas.*, Vol. 41, No. 5, pp. 692-698, 1992.

[Chan 00] Chan, S., H. Luong, W. Charlan, R. Fukuhara, E. Homberg, P. Jones, G. Pixler, "The Implementation of a COTS Based Fault Tolerant Avionics Bus Architecture," *IEEE Aerospace Conf. Proc*, Vol. 7, pp. 297-305, 2000.

[Conde 99] Conde, R. F., A. G. Darrin, F. C. Dumont, P. Luers, S. Jurczy, N. Bergmann, and A. Dawood, "Adaptive Instrument Module - A Reconfigurable Processor for Spacecraft Applications," http://www.xilinx.com/products/hirel_qml.htm, 1999

[Culbertson 97] Culbertson, W. B., R. Amerson, R. J. Carter, P. Kuekes, and G. Snider, "Defect Tolerance on the Teramac Custom Computer," *Proc. of the 1997 IEEE Symp. on FPGA's for Custom Computing Machines*, pp. 140-147, Napa Valley, CA, USA, 16-18 April 1997.

[Das 99] Das, D. and N. A. Touba, "A Low Cost Approach for Detecting, Locating, and Avoiding Interconnect Faults in FPGA-based Reconfigurable Systems," *12th Int'l Conf. on VLSI Design*, pp. 266-9, Goa, India, 7-10 Jan. 1999.

[DEC 80] DEC, Intel, and Xerox, "The Ethernet: A Local Area Network Data Link Layer and Physical Layer Specifications," V. 1.0, 1980.

[Dutt 99] Dutt, S., V. Shanmugavel, and S. Trimberger, "Efficient Incremental Rerouting for Fault Reconfiguration in Field Programmable Gate Arrays," *Digest of Technical Papers, IEEE/ACM Int. Conf. on Computer-Aided Design*, pp. 173-176, 1999.

[Emmert 98] Emmert, J. M., and D. Bhatia, "Incremental Routing in FPGAs," *Proc. of 11th Annual IEEE Int. ASIC Conf.*, pp. 217-221, 1998.

[Emmert 00] Emmert, J. M., C.E. Stroud, B. Skaggs, and M. Abramovici, "Dynamic Fault Tolerance in FPGAs via Partial Reconfiguration," *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, pp. 165-174, Napa Valley, CA, USA, 17-19 April 2000.

[Furutani 89] Furutani, K., K. Arimoto, H. Miyamoto, T. Kobayashi, K. Yasuda, K. Mashiko, "A Built-in Hamming Code ECC Circuit for DRAMs," *IEEE Journal of Solid-State Circuits*, V. 24, I. 1, pp. 50-56, Feb. 1989.

[Gokhale 98] Gokhale, M. and J.M. Stone, "NAPA C: compiling for a Hybrid RISC/FPGA Architecture," *Proc. IEEE Symp. FCCM'98*, Apr 1998.

[Hansen 90] Hansen, P., "Chapter 7: Taking Advantage of Boundary-Scan in Loaded-Board Testing," *The Test Access Port and Boundary Scan Architecture*, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[Hsu 96] Hsu; P.-C. and S.-J. Wang, "Testing and diagnosis of board interconnects in microprocessor-based systems," *Proceedings of the Fifth Asian Test Symposium*, pp. 56-61, 1996.

[Huang 01a] Huang, W.-J. and E. J. McCluskey, "Column-Based Precompiled Configuration Techniques for FPGA Fault Tolerance," to appear in *FCCM'01*, 2001.

[Huang 01b] Huang, W.-J., S. Mitra, and E. J. McCluskey, "Fast Run-Time Fault Location in Dependable FPGAs," to appear in *Proc. IEEE Defect and Fault Tolerance*, 2001.

[Lach 99] Lach, J., W. H. Mangione-Smith, and M. Potkonjak, "Algorithms for Efficient Runtime Faulty Recovery on Diverse FPGA Architectures", *DFT'99*, pp. 386-394, 1999.

[Losq 76] Losq, J. "A Highly Efficient Redundancy Scheme: Self-Purging Redundancy," *IEEE Trans. Comput.*, C-25, No. 6, pp. 269-78, 1976.

[Mitra 00] Mitra, S., W.-J. Huang, N.R. Saxena, S.-Y. Yu, and E.J. McCluskey, "Dependable Adaptive Computing Systems, The Stanford CRC ROAR Project," *2000 Pacific Rim International Symposium on Dependable Computing Fast Abstracts (PRDC 2000)*, pp. 15-16, Los Angeles, CA, USA, 18-20 Dec. 2000.

[Peterson 00] Peterson, L. L. and B. S. Davie, *Computer Networks – A Systems Approach*, 2nd Ed, Morgan Kaufmann, 2000.

[ROAR 01] http://www-crc.stanford.edu/projects/roar/roarSummary.html, 2001.

[Rupp 98] Rupp, C. R., M. Landguth, T. Garverick, E. Gomersall, H. Hold, J. M. Arnold, and M. Gokhale, "The NAPA Adaptive Processing Architecture," *IEEE Symp. on FPGAs for Custom Computing Machines*, pp. 28-37, Napa Valley, CA, USA, 15-17 April 1998.

[Siewiorek 73] Siewiorek, D. P. and E. J. McCluskey, "Switch Complexity in Systems with Hybrid Redundancy," *IEEE Trans. Comp.*, C-22, pp. 276-282, March 1973.

[Siewiorek 00] Siewiorek, D. P. and R. S. Swarz, *Reliable Computer Systems – Design and Evaluation*, 3rd Ed, Digital Press, 2000.

[Wagner 87] Wagner, P. T., "Interconnect Testing with Boundary Scan," *Int. Test. Conf.*, pp. 52-57, 1987.

[Wang 89] Wang, L.-T., M. Marhoefer, and E. J. McCluskey, "A Self-Test and Self-Diagnosis Architecture for Boards Using Boundary Scans," *Int. Test Conf.*, 1989.

[Wicker 95] Wicker, S. B., *Error Control Systems for Digital Communication and Storage*, Prentice-Hall, Inc., 1995.

[Xilinx 00a] Xilinx Application Note 181, "SEU mitigation Design Techniques for XQR4000XL," http://www.xilinx.com/xapp/xapp181.pdf, March 28, 2000.

[Xilinx 00b] Xilinx Application Note 216, "Correcting Single-Event Upsets Through Virtex Partial Configuration," http://www.xilinx.com/xapp/xapp216.pdf, June 1, 2000.

[Xilinx 01] Xilinx Virtex Datasheet, http://www.xilinx.com/apps/virtexapp.htm#databook, 2001.

[Yu 01] Yu, S.-Y. and E. J. McCluskey, "On-line Testing and Recovery in TMR Systems for Real-Time Applications," *Proc. 2001 Int'l Test Conf.*, pp. 240-249, 2001.