Center for Reliable Computing

TECHNICAL REPORT

Dependable Computing Techniques for Reconfigurable Hardware

Wei-Je Robert Huang

TR 01-7

June 2001

Center for Reliable Computing
Gates Building 2A, Room 236
Computer Systems Laboratory
Dept. of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

Abstract:

This technical report contains the digest of Wei-Je Huang's PhD thesis "Dependable Computing Techniques for Reconfigurable Hardware."

Funding:

This research was supported by the Advanced Research Projects Agency under Contract No. DABT63-97-C-0024.

DEPENDABLE COMPUTING TECHNIQUES
FOR
RECONFIGURABLE HARDWARE

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 PHILOSOPHY

Wei-Je Huang

June 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.

_____

Nirmal R. Saxena

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.

_____

Giovanni De Micheli

Approved for the University Committee on Graduate Studies:

_____

To Pei-Jen, Ethan, and my parents,
For their love, affection, and support.

ABSTRACT

Reconfigurable hardware, such as Field-Programmable Gate Arrays (FPGAs), has become a very important component in various applications, including communication, networking, and storage systems. With the re-programmable feature and a tremendous increase in on-chip logic and routing resources, contemporary reconfigurable hardware also opens new opportunities in designing dependable systems with fine-grained hardware redundancy.

In traditional dependable systems using hardware redundancy, fault tolerance is realized by replicating functional modules at the level of a chip or a board. Such coarse-grained hardware redundancy is very expensive. In reconfigurable hardware, however, a more cost-effective method is made possible by using an alternative configuration in which the faulty parts are replaced with originally unused resources in the same device.

Dependable computing using reconfigurable hardware requires three layers of support. First, Concurrent Error Detection (CED) is required in order to guarantee the correctness of outputs and detect errors. Second, transient error recovery is needed to restore normal operations from failures due to temporary environmental disturbances. Third, permanent fault recovery, including fault location techniques and reconfiguration strategies for tolerating permanent faults, is required to repair a system with physical failures in the hardware. Since these techniques are executed at run-time, it is desired to minimize their performance impact and to ensure high availability.

We investigate the issues in each layer of support for dependable computing in reconfigurable hardware. For CED techniques, we present the *inverse comparison* scheme suitable for applications with unique inverses. For transient error recovery, we examine a memory coherence issue in the recovery process and present a *dirty-bit memory coherence* technique with low execution time overhead. For permanent fault recovery, we present an integrated scheme using the *column-based precompiled configuration* technique for repair and the *blind reconfiguration* approach for fault location. Our techniques ensure very fast and cost-effective fault recovery in reconfigurable hardware.

ACKNOWLEDGEMENTS

I express my sincere appreciation to my advisor, Prof. Edward J. McCluskey, for his constant guidance, encouragement, and support during my time at Stanford. He has modeled many of the qualities and characteristics to which I aspire, and I have been greatly inspired by his enthusiasm for both teaching and research.

I would like to thank Dr. Nirmal R. Saxena, my associate advisor, for being on my orals and dissertation reading committees. I would also like to thank Prof. Krishna Saraswat for being the chairman of my orals committee, and Prof. Giovanni De Micheli for reading my dissertation and agreeing to be the final member of my orals committee.

I am very grateful to my current and former colleagues at the Center for Reliable Computing (CRC): Ahmad Al-Yamani, Dr. Jonathan Chang, Ray Chen, Eddie Yu-Che Cheng, Kan-Yuan Cheng, Dr. Santiago Fernandez-Gomez, Sam Kavusi, Chat Khunpitiluck, Moon-Jung Kim, James Chien-Mo Li, Dr. Siyad Ma, Dr. Samy Makar, Dr. Subhasish Mitra, Dr. Nahmsuk Oh, Philip Shirvani, Mehdi Tahoori, Dr. Nur Touba, Chao-wen Tseng, Sungroh Yoon, Yoonjin Yoon, Catherine Shu-Yi Yu, and Steve Zeng. I am greatly indebted to Siegrid Munda for her excellent administrative support.

I would like to thank my professors at Stanford and the CRC visitors: Prof. Jacob Abraham, Prof. Bella Bose, and Prof. Dhiraj Pradhan for many interesting discussions. In addition, I would like to thank Dr. Zhi-Min Ling, James Chen, Dave Mark, Shahin Toutounchi, and Greg Hyver from Xilinx for valuable comments on my research.

I want to thank my friends for making my life more enjoyable. I would like to mention just a few in particular: Chih-Hao Chang, Tsung-Ning Liu, Chung-Ta Lee, JT Hsu, Min-Chen Ho, Mike Cheng, Alex Teng, Shiang-Feng Lee, Chih-Hung Lin, Jim and Alma Phillips, and people from Stanford Taiwanese Student Association.

Finally, I wish to thank my parents and sisters for their endless support and tremendous love. Most of all, I would like to thank my wife, Pei-Jen Hsu, for her encouragement, understanding, patience, and love, and my beloved son, Ethan, for his inspiration in my final year of PhD study.

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

TABLE OF CONTENTS

Appendix B

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, 2000.

Appendix C

Huang, W.-J., and E.J. McCluskey, "A Memory Coherence Technique for Online Transient Error Recovery of FPGA Configurations," *Proc. ACM International Symposium on Field-Programmable Gate Arrays*, pp. 183-192, 2001.

Appendix D

Huang, W.-J., and E.J. McCluskey, "Column-Based Precompiled Configuration Techniques for FPGA Fault Tolerance," to appear in *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines*, 2001.

Appendix E

Huang, W.-J., S. Mitra, and E.J. McCluskey, "Run-Time Fault Location in Dependable FPGAs," *Technical Report, CRC TR-01-5*, Center for Reliable Computing, Stanford University, 2001, http://crc.stanford.edu.

LIST OF ILLUSTRATIONS

LIST OF TABLES

Chapter 1
Introduction

1.1 Background

With the prevalence of computing systems in various applications, dependability has become an essential factor for computing system design. For example, in unmanned space environments, dependability of a computing system has a major impact on the cost of a mission because it is difficult to replace the system once it becomes faulty. Other examples include mission-critical terrestrial applications such as nuclear reactors, fly-by-wire systems, and life-critical systems.

Dependability comprises availability, reliability, testability, maintainability, and fault-tolerance [Siewiorek 92]. *Availability* is the probability that the system is operational at a specified time. *Reliability* is the probability that the system is operational up to a specified time. *Testability* is the ease with which the system can be tested, including the capability for both manufacturing tests and on-line tests during operation. *Maintainability* is the probability that a failed system can be repaired within a specified time. *Fault-tolerance* is the ability of the system to operate correctly in spite of some specified faults. In general, a dependable system requires error detection during operation for testability, and fast recovery and repair schemes for enhancing availability, reliability, maintainability, and fault-tolerance.

Unfortunately, although there is a thrust from users for improving dependability of computing systems, the trend of technology scaling results in more sensitive circuits and greater challenges for reliability [Dai 99, Nagaraj 99, Seifert 01]. With the shrinking of device feature size to deep sub-micron, the amount of charge required to activate a transistor reduces. This makes transistors more susceptible to upsets caused by radioactive particles [Dai 99, Seifert 01]. Meanwhile, however, the die size becomes larger and larger in order to package more functions in a chip. As a result, the larger dies become even more susceptible to failures.

Dependable computing systems can generally be realized by two hardware-oriented approaches: fault avoidance and fault tolerance, for achieving better reliability. *Fault avoidance* refers to techniques that are used to prevent the occurrence of faults

[Pradhan 96]. Examples of fault avoidance techniques are to use shielded or radiation-hardened devices to alleviate the influence of radioactive particles in a radiation-intensive environment. However, shielding increases the volume and weight of the device, and radiation hardening is very expensive and is not widely available. In addition, development of radiation-hardened devices generally lags behind the most advanced technology. For example, while the most advanced *Field-Programmable Gate Arrays* (FPGAs) have the maximum capacity of 10M system gates and 3.5Mb on-chip memory (e.g., Xilinx Virtex-II XC2V10000 FPGAs), the largest radiation-hardened FPGAs contain only 1M system gates and 130Kb on-chip memory (e.g., Xilinx QPRO-Virtex XQV1000 FPGAs) [Xilinx 01].

For fault tolerance, *hardware redundancy* is a commonly used approach [Siewiorek 92, Pradhan 96]. Traditionally, hardware redundancy is realized at a *coarse-grained* level. All functional modules in a system are replicated at the level of a chip or a board so that faults in part of the system can be tolerated.

A typical example of fault tolerance using hardware redundancy is a *Triple Modular Redundant* (TMR) system, which is shown in Fig. 1-1(a). In a TMR system, three functional modules are designed to perform the same functions, and a voter is used to determine the correct output and mask faults in the system.

Another example of fault tolerance using hardware redundancy is a *hybrid redundant* system [Siewiorek 92], which is shown in Fig. 1-1(b). In a hybrid redundant system, N out of (N+S) identical modules are active, and their outputs are voted to produce the system output. When a disagreement among the outputs is detected, the modules in the minority are considered to be failed and are replaced by the equivalent number of spare modules.

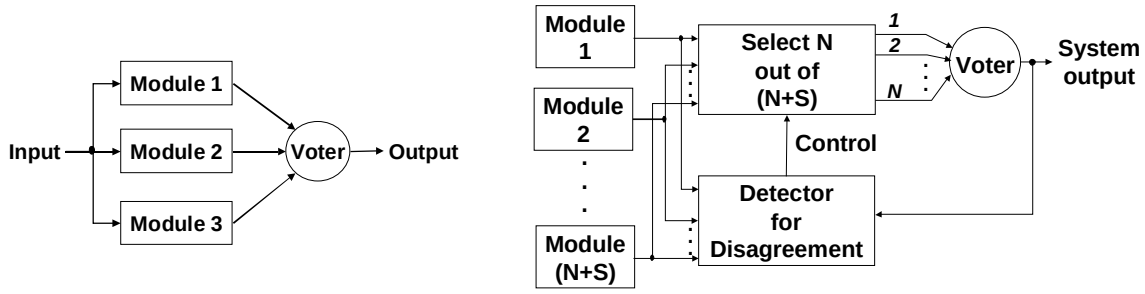

Figure 1-1: Coarse-grained hardware redundancy. (a) TMR. (b) Hybrid redundancy.

Although a TMR or a hybrid redundant system achieves substantial improvement in dependability, such coarse-grained hardware redundancy is very expensive in terms of area overhead. Therefore, there is a strong drive for a more cost-effective solution for dependable computing systems.

Recently, reconfigurable hardware, such as SRAM-based FPGAs or *Complex Programmable Logic Devices* (CPLDs), has emerged as excellent candidates for building cost-effective dependable computing systems [Culbertson 97, Mange 98, Saxena 98]. Unlike traditional Programmable Logic Arrays (PLAs) that are one-time configurable, contemporary reconfigurable hardware uses memory, such as SRAM or DRAM cells, to store the functional configuration and can thus be reprogrammed at run-time.

With the advance of process technology and device architecture, current-generation reconfigurable hardware can be used to implement millions of logic gates with the maximum clock rate up to several hundred-MHz [Altera 01, Xilinx 01]. Integrated with the run-time re-programmable feature, reconfigurable hardware provides the flexibility to optimize a computing system for different figures of merit, including time-to-market, performance, cost, and fault tolerance.

From the time-to-market perspective, the design cycle of a novel product is drastically reduced by using reconfigurable hardware. There are two reasons for this advantage. First, components using reconfigurable hardware can be directly constructed by downloading the configuration bit-stream created by CAD tools without actual fabrication of detailed circuitry. Second, designs can be emulated and verified using actual components of the product, and making design changes is much easier than the traditional Application-Specific Integrated Circuit (ASIC) approach. Since time-to-market is very crucial in the highly competitive electronics industry, FPGAs and CPLDs have become key components in various applications, including communication, networking, and storage systems. A typical example is the extensive use of FPGAs in switches and routers in current networking industry [Cisco 01, Nortel 01].

From the performance perspective, the abundance of logic and routing resources in current reconfigurable hardware makes it possible to implement highly parallel architectures to boost the throughput of many applications [Hauck 98, Huang 00a]. Even though the clock frequency of applications in reconfigurable hardware is not as high as

current general-purpose microprocessors, the performance gain because of the parallelism in reconfigurable hardware is still considerable for some applications.

From the cost perspective, optimized designs of different applications can be multiplexed in time domain on the same reconfigurable hardware to achieve better efficiency [Chang 98, Trimberger 98]. Compared to ASICs that can only implement a certain "glue-logic" in a chip, the versatility in reconfigurable hardware significantly reduces the cost both for a commercial computing system and for the design process of a product.

From the fault tolerance perspective, the re-programmable feature makes it possible to tolerate faults within a chip by using an alternative configuration that replaces faulty parts with previously unused resources in the same chip. Equivalently, this method realizes the hardware redundancy at a *fine-grained* level. Because reconfigurable hardware is not specifically tailored for a certain function, applications normally do not utilize all resources in reconfigurable hardware. Consequently, fine-grained, idle resources for replacement are generally available within reconfigurable hardware, and the resulting cost for fault tolerance can be significantly lower than conventional coarse-grained hardware redundancy approaches.

Various dependable, *Adaptive Computing System* (ACS) architectures using reconfigurable hardware have been proposed. Two examples, the *Multi-Threaded ACS* (MT-ACS) architecture [Saxena 00] and the *Dual-FPGA ACS* architecture [Mitra 00b], are shown in Fig. 1-2.

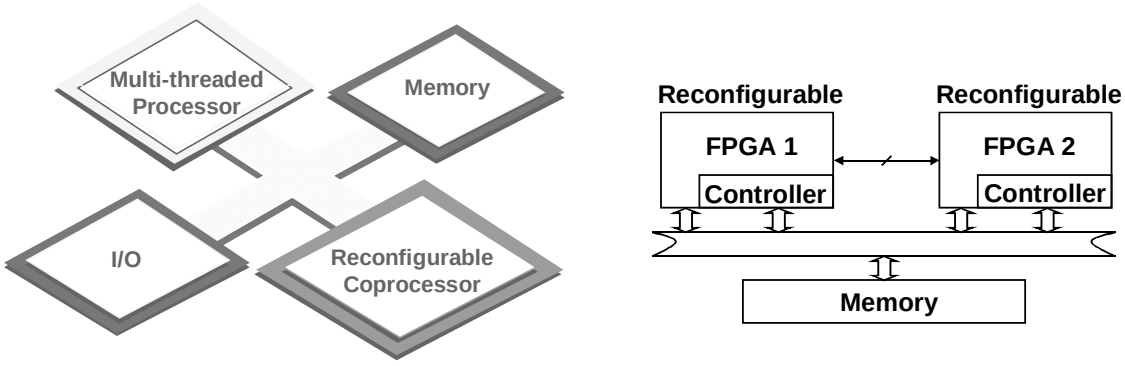

Figure 1-2: Dependable ACS architectures. (a) MT-ACS. (b) Dual-FPGA ACS.

In the MT-ACS architecture, a multi-threaded processor, memory, I/O devices and a reconfigurable coprocessor are connected to a bus. Applications exhibiting

parallelism can be mapped and executed on the reconfigurable coprocessor for performance improvement. Fault tolerance in the system is accomplished by both voting on multiple threads for the same computations in the multi-threaded processor and the re-programmable feature of the reconfigurable coprocessor.

In the Dual-FPGA ACS architecture, each FPGA is configured to run certain applications with some error detection schemes. The controller on each FPGA monitors the error signal from the other FPGA and reconfigures the other FPGA to tolerate faulty parts when necessary.

To fully exploit the advantage of reconfigurable hardware in building dependable computing systems, the following run-time supports are necessary and are investigated in this dissertation:

- *Concurrent Error Detection* (CED) schemes that detect errors on-line for applications implemented in reconfigurable hardware. Such schemes should have low performance overhead and guarantee the correctness of system output.
- Transient error recovery techniques for restoring normal operations from failures due to temporary environmental disturbances. Short latency is desired for these techniques in order to reduce system downtime and enhance system availability.
- Permanent fault recovery techniques for repairing a system with physical failures in the hardware. These include fault location techniques for providing diagnostic information and repair techniques by reconfiguration for permanent fault tolerance. Short diagnostic latency is desired for reducing system downtime, but a very fine-grained fault location resolution is not always necessary. A reasonable fault location resolution that is acceptable for the run-time reconfiguration strategy used for repair is sufficient.

1.2 Contributions

We have investigated various dependable computing techniques in reconfigurable hardware. The techniques include low-overhead, application-specific CED schemes in

FPGAs, transient error recovery techniques in FPGAs with a related memory coherence strategy, and a reconfiguration scheme for both permanent fault location and fault tolerance. In particular, the following summarizes the contributions of this dissertation:

- We present an *inverse comparison* CED technique suitable for applications that have unique inverse transformation. This technique has been developed and implemented in the LZ compression application in FPGAs and has low area overhead.
- We have developed and implemented *rollback recovery* techniques for restoring normal operations in the LZ compression application from transient errors that do not change configuration data of the reconfigurable hardware.
- We have identified the methodology using the *configuration readback* and *writeback* features in contemporary FPGAs for error detection and correction in configuration data. Integrating this methodology with the rollback recovery technique, the joint error recovery scheme restores normal operations from transient failures in reconfigurable hardware.
- We have developed a *dirty-bit memory coherence* strategy for transient error recovery in FPGA configuration data. The *memory coherence* issue arises in transient error recovery schemes because the configuration error recovery using readback and writeback may corrupt user memory data stored in the FPGA. Simulation results show that our technique guarantees memory coherence during transient error recovery in reconfigurable hardware and has low overhead on user operations.
- We have developed a new technique, the *column-based precompiled configuration* approach, for tolerating permanent faults in reconfigurable hardware. The post-fault-location system downtime can be minimized, and the storage overhead for extra configuration data is significantly reduced compared to previous reconfiguration techniques.
- We have developed a new technique, the *blind reconfiguration* approach, as a fast alternative for run-time fault location in reconfigurable hardware. With the integration of CED schemes and the reconfiguration strategy for

fault tolerance in reconfigurable hardware, our technique achieves significant availability improvement by minimizing the number of reconfigurations required for the entire fault location and recovery process.

1.3 Outline

This dissertation summarizes my work in developing dependable computing techniques in reconfigurable hardware. Detailed descriptions of results are found in the appendices, which are reprints of published or submitted papers.

Chapter 2 overviews the architectural model for reconfigurable hardware used in this dissertation. Our model is consistent with SRAM-based FPGAs commercially available in current industry. Therefore, in this dissertation, we use the term "FPGA" interchangeably with reconfigurable hardware.

Chapter 3 describes CED techniques for detecting errors during operations. A survey of application-independent approaches is presented followed by a summary of an application-specific technique, the inverse comparison CED, developed for applications with unique inverse.

Chapter 4 describes transient error recovery techniques in reconfigurable hardware and the related memory coherence issue. A rollback recovery scheme for LZ compression and FPGA configuration error recovery techniques are described, and the dirty-bit memory coherence strategy is presented for solving the memory coherence issue.

Chapter 5 presents reconfiguration strategies for tolerating permanent faults. We begin the chapter by a review of previous approaches and a discussion of their advantages and drawbacks. We then present the column-based precompiled configuration approach for FPGA fault tolerance. Experimental results in storage overhead reduction for extra configuration data are provided and compared with previous approaches, and theoretical analyses for dependability improvement are presented.

Based on the column-based precompiled configuration approach, Chapter 6 presents a fast alternative, the blind reconfiguration approach, for run-time fault location in FPGAs. An enhanced version of our approach that minimizes the overall latency of

fault location and recovery is also presented. We then briefly discuss a case study of applying our approach in the LZ compression application in FPGAs.

Chapter 7 concludes the dissertation.

Chapter 2

Reconfigurable Hardware Model: SRAM-Based FPGAs

Figure 2-1 shows the model for the programmable logic array of reconfigurable hardware used in this dissertation. This model is consistent with contemporary SRAM-based FPGAs [Xilinx 01]. In this model, the programmable logic array of reconfigurable hardware consists of three basic elements: *Configurable Logic Blocks* (CLBs), *Connection Boxes* (CBs), and *Switch Boxes* (SBs).

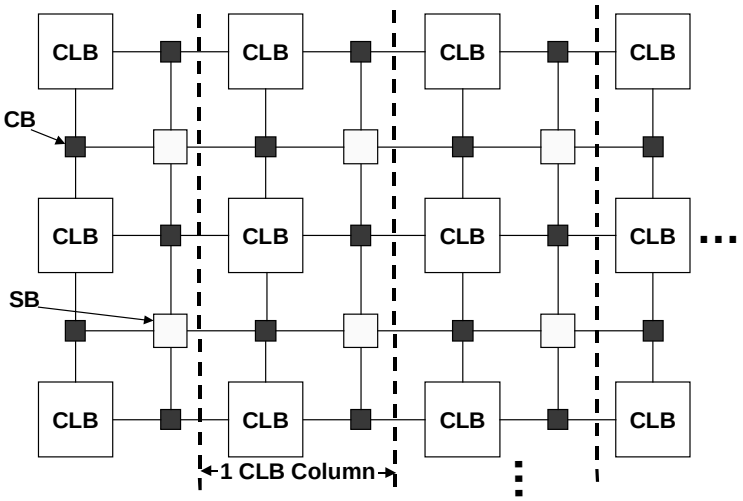

Figure 2-1: The programmable logic array of reconfigurable hardware.

In this architecture, a CLB contains SRAM lookup tables (LUTs) that store the truth tables of user-defined combinational logic functions. In this way, a combinational logic gate is realized by looking up the value in the LUT that is addressed by the corresponding gate inputs.

Because LUTs are implemented in SRAM cells, they can also be configured as memory modules in user applications. Here, *memory modules* refer to all types of memory implemented as RAM modules, such as RAM and FIFO. Bistables such as flip-flops and latches, however, are excluded from memory modules implemented in LUTs because of area efficiency. Instead, they are implemented as separate modules in each CLB to realize sequential logic. Each CLB may also contain multiplexers and other dedicated circuitry in order to enhance functional flexibility and to speed up certain paths with long delay, such as the carry chain in a ripple adder [Xilinx 01].

To implement a logic network, CLBs are connected through horizontal and vertical wiring channels located between two neighboring rows or columns. To enhance the connectivity, there are various kinds of wires with different lengths for connecting various CLBs that are separated by variable numbers of blocks. For example, in Xilinx Virtex-series FPGAs, *single lines* connect adjacent CLBs, while *hex lines* connect CLBs that are three or six blocks apart [Xilinx 01].

There are two types of routing devices, CBs and SBs, to direct the signal flows among CLBs and wiring channels. CBs route the inputs and outputs of a CLB to the adjacent wiring channels. SBs connect horizontal and vertical wiring channels. Both CBs and SBs are matrices of *Programmable Interconnect Points* (PIPs). The state of PIPs in these switch matrices is controlled by SRAM cells, which are configured according to the desired functionality.

Logic functions in the reconfigurable hardware are configured through the configuration bit-stream, which contains a mixture of commands, configuration data, SRAM cell addresses for storing configuration data, and control overhead such as parity checks. Configuration data contain values in LUTs and interconnect states.

The entire configuration data of the reconfigurable hardware are partitioned vertically or horizontally into *configuration frames*, which are the smallest amount of data that can be accessed with a single configuration command. Data stored in configuration memory can be accessed by two types of operations: configuration readback and configuration writeback. *Configuration readback* refers to the operation of reading the on-chip configuration memory contents out of the chip through designated ports. This operation does not affect user applications running in the reconfigurable hardware. *Configuration writeback* refers to the operation of writing valid configuration data into the on-chip configuration memory cells. Because the functional definition of the reconfigurable hardware is changed by configuration writeback, user operations may be affected.

Partitioning the entire configuration data into frames in the reconfigurable hardware enables *concurrent partial reconfiguration* of the functional definition of the circuit. Because configuration writeback can be executed frame by frame, part of the configuration can be altered without stopping the operations running in the

reconfigurable hardware, providing these operations do not interact with the frames under configuration writeback.

Without loss of generality, we assume that the entire configuration data are partitioned vertically such that a CLB column in the programmable logic array contains one or more configuration frames. In this architecture, a *CLB column* includes all CLBs and the corresponding switch matrices (CBs and SBs) in the same column of the array. The actual circuitry, programmable logic and routing resources, and the configuration architecture of each CLB column are identical.

Figure 2-2 shows an example of configuration frame partitioning in Xilinx Virtex-series FPGAs [Xilinx 01]. The entire configuration data are partitioned vertically along each column of blocks in the array. The number of frames and the configuration structure in each CLB column are identical, and there are various types of columns (other than the CLB columns) with different numbers of frames that represent configuration for clock distribution, primary inputs and outputs for the chip, and on-chip SRAM blocks for user applications.

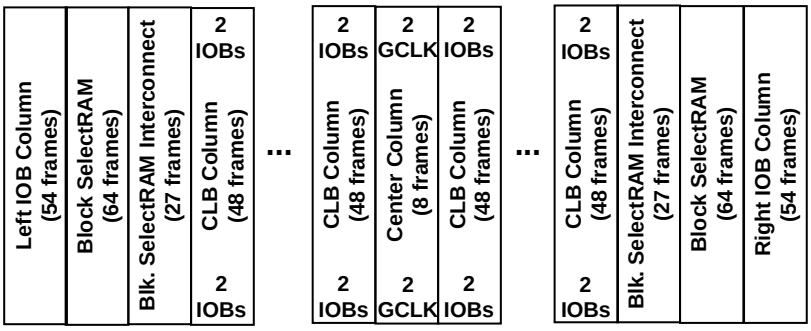

Figure 2-2: Configuration frame architecture in Xilinx Virtex-series FPGAs.

Chapter 3

Concurrent Error Detection

The first step towards dependable computing is to guarantee the data integrity of system outputs. *Data integrity*, or the *fault-secure* property, means that the system either produces correct outputs or indicates errors when incorrect outputs are produced [Siewiorek 92, Mitra 00a]. Concurrent Error Detection (CED) techniques, which ensure data integrity by detecting errors during operations, are studied in this chapter. In particular, we briefly discuss an application-specific CED technique, inverse comparison, for applications with unique inverse. Detailed discussions can be found in Appendix A.

3.1 Related Work

For CED techniques, correctness of the system is typically verified by redundant computations either in the space domain (*hardware redundancy*) or in the time domain (*time redundancy*). For example, Fig. 3-1 shows a general architecture of a CED scheme using hardware redundancy [Pradhan 96, Mitra 00a]. Besides the original functional unit *F*, an additional independent unit *P* is used for predicting some special characteristic of the system output for every input sequence. Such characteristic can be the output itself, its parity, 1's or 0's count, transition count, etc. A checker unit then compares the output characteristic from *F* with the prediction from *P* and produces an *error* signal when a mismatch occurs.

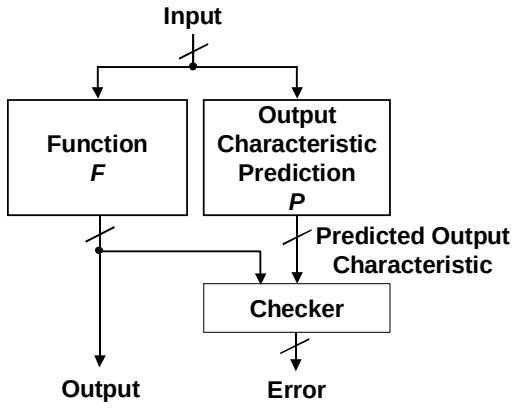

Figure 3-1: Concurrent error detection with hardware redundancy.

Duplex systems are a classical example of CED using hardware redundancy [Sellers 68, Kraft 81, Sedmak 78, Spainhower 99]. In such systems, the output characteristic predictor *P* implements the same function as the original functional module *F*, and the checker compares the output produced by both modules. Duplicated modules can be identically implemented, or *design diversity* can be used to protect the system against *Common-Mode Failures* (CMF) that produce multiple faults due to a single cause [Mitra 00a].

Parity prediction and uni-directional error detecting codes are other examples of CED using hardware redundancy. *Parity prediction* checks whether the even or odd property of the number of 1's in the output is observed. Techniques using parity prediction for datapath circuits and general combinational and sequential circuits are described in [Nicolaidis 93, Nicolaidis 97, Touba 97, Zeng 99]. *Uni-directional error detecting codes*, which check the number of 1's or 0's in the output, are described in [Berger 61, Bose 85].

In additional to different system-level designs that exploit various output characteristics, checker circuitry has also been developed in the past for CED using hardware redundancy [Wakerly 78, McCluskey 90]. If self-checking checkers described in [McCluskey 90] are used, the aforementioned CED schemes using hardware redundancy can guarantee data integrity against single faults. However, for CED techniques using hardware redundancy, data integrity is assured at the expense of area overhead. Such area overhead is typically around 100% or more compared to the original functional module [Mitra 00a].

Compared to hardware redundancy, CED techniques using time redundancy can reduce the area overhead in the system. CED techniques using time redundancy include alternating logic [Reynolds 78], alternate-data retry [Shedletsky 78], data complementation [Takeda 80], recomputing with shifted operands [Patel 82], and error detection by duplicated instructions [Oh 01]. However, time redundancy techniques generally cause performance degradation in the system.

The aforementioned CED techniques using hardware redundancy or time redundancy are application-independent. To achieve better efficiency in CED schemes, *application-specific CED* (also known as *algorithm-based CED*) techniques that exploit

special properties of the original function are also studied in the past. For the specific application, such techniques generally achieve lower area and performance overhead than application-independent CED schemes. For example, efficient CED techniques have been developed for Fast Fourier Transform networks [Jou 88].

3.2 Inverse Comparison CED

For applications with unique inverses, we present an *inverse comparison* technique in Appendix A as a potential candidate for achieving efficient concurrent error detection. Typical applications exhibiting this property are one-to-one mapped encoders (or decoders) in communication or storage systems. Since the inverse transformation is unique, the requirement for a dependable encoder is to ensure that the source data can be identically reconstructed in the decoder. If we perform the inverse transform on the encoded codewords before they are sent out, the resulting data should perfectly match the original source inputs. Therefore, instead of duplicating the encoding computations, data integrity can be guaranteed by checking if source data can be identically reconstructed from the encoded codewords.

Figure 3-2 shows the architecture of the inverse comparison CED scheme used in an encoder. In this scheme, an additional decoder (inverse process) reconstructs data from the encoded codewords (forward process output), and a checker compares the reconstructed data with the delayed source input. In practice, there are two primary concerns for the inverse comparison CED technique: (1) area overhead and execution time overhead due to the introduction of decoder, and (2) potential mismatch between reconstructed data and source data due to round-off errors.

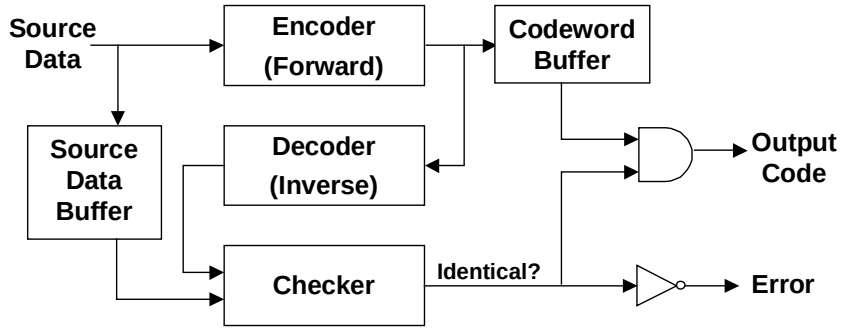

Figure 3-2: Inverse comparison CED scheme.

First, to avoid large area or execution time overhead, the complexity of the inverse process should be significantly lower than the forward process. Also, if the throughput of the inverse process matches that of the forward process, we can pipeline the error detection computations so that the overall throughput degradation for the forward process is negligible.

Second, round-off errors inherent in floating-point calculations need to be distinguished from real errors during operations when comparing the reconstructed data and the original source input. To achieve this objective, techniques such as *backward error assertions with iterative refinement* [Boley 95] have been developed for error detection in solving linear equations. For applications that only involve integer computations and bit manipulations, however, mismatch due to round-off errors does not cause problems. The inverse comparison CED technique is thus more feasible in integer-based or bit-manipulation applications.

3.3 Case Study: LZ Compression Encoder in FPGAs

In this section, we present a summary of a case study of an inverse comparison CED scheme applied in a commonly used data compression algorithm, *Lempel-Ziv* (LZ) compression [Ziv 77, Ziv 78], implemented in FPGAs. Detailed discussions can be found in Appendices A and B.

There are three major reasons for choosing this application as our case study of inverse comparison. First, as analyzed in Appendix B, LZ compression has a serious error propagation problem in that a single encoding error can cause significant damage in the reconstructed data at the decoding end. In this way, even if high precision may not be required in the original data before compression (e.g., digitally encoded voice or image data in which a single-bit error may not cause serious problems for human perception), the error propagation in the decoded data because of an encoding error can still be catastrophic. Therefore, data integrity is even more important in this application.

Second, LZ compression involves massive parallelism in searching for a maximum-length matching phrase in a dictionary. Although such massive parallelism can be realized by using reconfigurable hardware, CED using an application-independent hardware redundancy approach, such as duplication of every parallel hardware

component, can be very expensive. Consequently, it is desired to design an area-efficient CED scheme for this application implemented in reconfigurable hardware.

Third, LZ compression not only exhibits the unique inverse property but also has a simple decoding process. Also, LZ compression does not involve floating-point operations and thus does not have round-off error problems. These features make the inverse comparison technique a perfect example of the CED scheme for LZ encoder.

3.3.1 LZ Compression Algorithm and Encoder Architecture

The fundamental concept of LZ data compression is to replace variable-length phrases in the source data with pointers to a matching position in a dictionary. The dictionary is constructed dynamically by using previously encountered source input phrases. Figure 3-3 shows the encoder diagram of the LZ data compression. Initially, the dictionary array is empty. In each cycle, the latest source data symbol and all the dictionary elements shift one position leftwards, evicting the oldest data symbol in the array. In this way, the dictionary always contains the most recent source phrases and is updated dynamically.

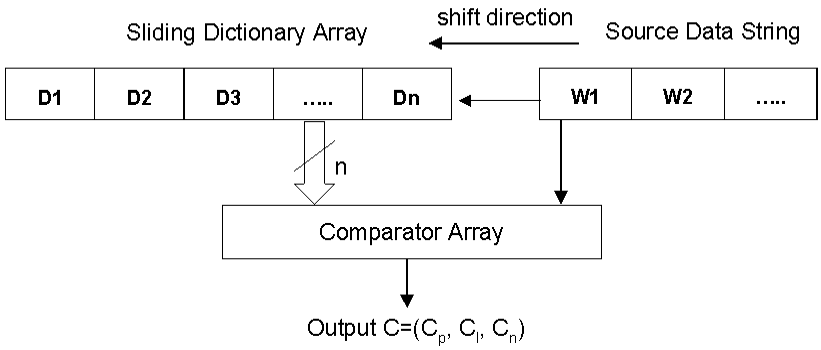

Figure 3-3: LZ encoder diagram.

To encode, source data strings are compared with the current dictionary array to find the maximum-length matching phrase. Once such a matching phrase is found, an output codeword $C=(C_p, C_l, C_n)$ is generated. Each codeword contains three elements: the pointer (C_p) to the starting position of the matching phrase in the dictionary, the length (C_l) of the matching phrase, and the next source data symbol (C_n) immediately following the matching phrase.

Figure 3-4 shows a high-performance, systolic-array architecture for an LZ encoder as proposed in [Jung 98]. This architecture achieves a throughput of one input

symbol per cycle and is used in our case study. In this architecture, the LZ encoder consists of a systolic array of 512 shift registers with corresponding *Processing Elements* (PEs). Each shift register is 8-bit wide, and each PE is basically a comparator that matches source symbols with data in shift registers. The priority encoder takes the match results from the PE array and encodes the match position in the array. The OR-tree takes the match results from the PE array and determines if a match is found. A length counter is used for counting the matching length, and there are input and output buffers and controllers for coordinating input and output sequences.

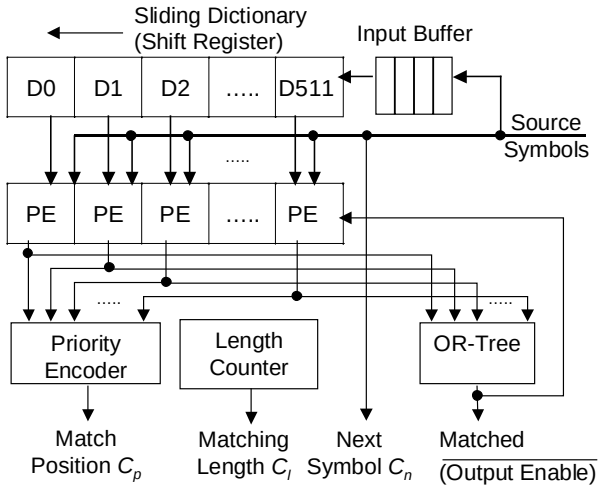

Figure 3-4: Systolic-array architecture of LZ encoder.

Note that all dictionary elements are compared with source symbols in each cycle, and thus the dictionary has to support multiple ports for extracting every element at the same time. Therefore, the sliding dictionary in Fig. 3-4 cannot be placed in memory modules implemented by LUTs when we map the LZ encoder architecture onto FPGAs. Instead, they are implemented by dedicated flip-flops in CLBs.

Decoding of LZ compressed codewords is much simpler than encoding. It comprises memory accesses to a decoding dictionary, and thus is expected to require smaller area than the encoder circuitry.

3.3.2 Extra Hardware for Inverse Comparison

Additional hardware for the inverse comparison CED in an LZ encoder contains three major components. They are (1) the decoder hardware, (2) source data buffers and output codeword buffers, and (3) the error checker.

The major unit in the decoder hardware is a decoding dictionary. Unlike the encoding dictionary that can only be realized by flip-flops in CLBs, the decoding dictionary can be mapped in dual-port RAM modules implemented in LUTs or dedicated on-chip memory blocks. This is because the LZ decoding does not involve parallel comparisons. Since RAM modules are more area-efficient than high-performance flip-flops, contemporary FPGAs have more LUT entries than flip-flops in each CLB. For example, for Xilinx Virtex-series FPGAs [Xilinx 01], each CLB slice has the capacity of a 16x1 dual-port RAM module implemented in two LUTs, while it contains only two flip-flops in the same block. Therefore, even though a separate copy of memory is used for the decoding dictionary in the inverse comparison, the area overhead for the dictionary is still very small.

Source data buffers and output codeword buffers are used for reducing the throughput degradation caused by the inverse comparison CED scheme. Like the decoding dictionary, these buffers can be mapped in the dual-port RAM modules implemented in LUTs or dedicated on-chip memory blocks for area efficiency. *Non-blocking buffers*, which have sufficient capacity for buffering data and preventing stalls in the encoding operation, can be designed such that the encoder throughput is not degraded. Detailed design of non-blocking buffers in this case study can be found in Appendix A.

Compared to the massively parallel PE array in the encoder, the area overhead due to the error checker is trivial. As a result, the area overhead because of the inverse comparison CED in LZ encoder hardware is contributed mostly by memory elements – the decoding dictionary and buffers – which can be mapped using area-efficient methods in contemporary FPGAs.

3.3.3 Area Overhead and Performance Impact

To estimate the area overhead and performance impact, we emulated a LZ encoder with inverse comparison CED on a WILDFORCE reconfigurable computing engine [Annapolis 01]. The board contains five Xilinx XC-4036XLA FPGAs [Xilinx 01]. Detailed layout and architecture of the testbed can be found in Appendix A.

We choose 64 entries for both source data buffer and codword buffer to approximate non-blocking buffers. The resulting latency overhead due to the inverse comparison CED is only 0.5%.

Table 3-1 shows the area overhead for the inverse comparison CED and duplex CED schemes. The area overhead is measured by the number of CLBs used in Xilinx XC-4036XLA FPGAs. Since LZ compression has a simple decoding process, the inverse comparison CED scheme achieves significant reduction in area overhead.

Table 3-2 shows the comparison of the LZ encoder throughput for the WILDFORCE emulation with inverse comparison CED and C programs without CED schemes on general-purpose processors. The purpose of the comparison is to illustrate how reconfigurable hardware improves the LZ encoder throughput even when the inverse comparison CED scheme is used. The C program without CED is obtained from [Nelson 96]. It is clear that even with a slower clock rate, the WILDFORCE emulation can achieve more than 30 times speed up due to the massive parallelism in the reconfigurable hardware.

Table 3-1: Area overhead for different CED schemes.

CED scheme	None	Duplex	Inverse Comparison
Number of CLBs	4,219	8,440	4,607
CLB overhead	0	100%	9.20%

Table 3-2: Throughput comparison.

System (Hardware)	WILDFORCE (4 Xilinx 4036XLA)	C program (1 UltraSparc II)	C Program (1 Pentium II Xeon)
Max. clock rate	16 MHz	300 MHz	450 MHz
Throughput	128 Mbps	3.0 Mbps	4.2 Mbps

3.3.4 Effect of Undetected Faults

In the inverse comparison CED scheme, data integrity is guaranteed by ensuring that the output codewords can be used to reconstruct source data correctly at the decoding end. However, some faults may never be detected by such a CED scheme because the resulting codeword can still be correctly decoded and the output data integrity is still preserved in the presence of these faults.

In LZ compression with inverse comparison CED, a typical example of undetected faults are those that have the same effect as if one or more PE outputs are stuck at "unmatched" logic value. This includes the stuck-at-unmatched PE outputs and the case where intermediate signals in the OR-tree are stuck at 0. In the presence of such faults, the codewords can still be used to reconstruct original source data because the codewords are generated according to the PEs that produce a "matched" signal.

Equivalently, the effect of such undetected faults is the reduction of the number of elements in the dictionary, which degrades the compression ratio in the LZ encoder. Simulation results in Appendix A show that the average *compression ratio degradation*, which is measured by the decline in the percentage of data size reduced by compression, is about 23%.

To prevent compression ratio degradation due to undetected faults, we can schedule a testing mode in between two successive compression jobs to check if such type of faults exists in the encoder. The primary goal of the testing mode is to detect any fault in the OR-tree network and PE outputs that causes the effect of stuck-at-unmatched PEs. To achieve this goal, we can stimulate the fault by forcing only one of the PEs to produce a "matched" result in each cycle and observe whether the *Matched* signal in Fig. 3-4 remains asserted during the test mode. This can be implemented by shifting a unique symbol *S* into the dictionary once at the beginning of the testing mode and comparing every dictionary element with *S* in each cycle. The only difference from the normal encoding operation is that the unique symbol *S* is only shifted once into the dictionary in order to produce only one "matched" result among all PE outputs in each cycle. Detailed discussion of the test mode can be found in Appendix A, and the test latency for an *N*-entry dictionary design is *N* cycles.

3.4 Summary

The inverse comparison CED scheme guarantees data integrity in the output for applications with unique inverse transforms. It causes small overhead when the inverse process has a lower complexity than the forward operations. The LZ compression algorithm is a typical example that exhibits such characteristics, and the inverse comparison CED is very efficient for ensuring data integrity for such an application.

However, there are situations where inverse comparison CED never detects a fault because the output produced can still be correctly inverted. Although data integrity is not compromised, such faults can degrade the functional performance (such as the compression ratio in LZ compression algorithm) in the application. A case study using LZ compression as a target application shows that the compression ratio degradation due to such faults is graceful, and we can schedule a testing mode in between successive compression tasks to detect such faults.

Chapter 4

Transient Error Recovery and Memory Coherence

Transient faults, also known as *soft faults*, describe faults resulting from temporary environmental conditions [Siewiorek 92]. Because of transient faults during operation, errors occur even in the absence of physical defects in the hardware. A typical example of transient errors is Single-Event Upsets (SEUs) induced by radiation, such as alpha particles and cosmic rays [May 79a, May 79b, Woods 86, Shirvani 00].

The occurrence rate of transient errors can be substantial. For example, SEUs are a major concern in a space environment. In [Shirvani 00], about 5.5 upsets per-megabyte, per-day is reported on commercial off-the-shelf (COTS) memory components in the ARGOS satellite, which has a sun-synchronous, 834-kilometer altitude orbit with a mission life of three years. In [Carmichael 99], upsets at a rate of one per hour are expected on three Xilinx Virtex V1000 FPGAs [Xilinx 01] for applications in a projected low earth orbital path.

For terrestrial applications, transient errors are also non-trivial. For example, [Ziegler 96] reported the observation of SEUs at the ground level, and the mean-time-to-occurrence for transient faults is about 354 hours for a Sun-2 distributed file server system [Lin 90]. Therefore, recovery from transient errors is essential for dependable computing systems.

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. Examples of system transients are SEUs that change user memory contents, and cross coupling in signal lines that change the voltage level of a signal. In Sec. 4.1, we summarize general techniques for recovery from system transients and briefly discuss two recovery techniques that we developed for the case study in Sec. 3.3, LZ compression with inverse comparison CED. Detailed discussions and results can be found in Appendix B.

Configuration transients cause errors in configuration memory and alter the configuration of the reconfigurable hardware. Examples of configuration transients are SEUs that change the values in configuration memory cells that control the state of PIPs or represent the truth table of user-defined logic functions. From the user perspective, configuration transients manifest themselves as permanent faults in the hardware because functions of the hardware change with the altered configuration memory contents. Therefore, system transient recovery techniques are not applicable to restore normal operations from configuration transients. We discuss configuration transient recovery techniques in Sec. 4.2.

When both types of transient recovery techniques are integrated, a memory coherence problem occurs because the configuration transient recovery process may change user data in reconfigurable hardware. We briefly discuss this issue and present a memory coherence technique in Sec. 4.3. Detailed discussions and simulation results for memory coherence in transient recovery schemes in reconfigurable hardware can be found in Appendix C.

4.1 Recovery from System Transients

For system transients, traditional transient error recovery techniques, such as forward recovery or rollback recovery, can be applied to correct errors. These approaches are designed at the system level and are applicable for applications in both ASICs and reconfigurable hardware.

The *forward recovery* technique [Jewett 91, Serlin 84, Johnson 89, Long 90, Pradhan 94, Pradhan 96] is to continue executing the program without retrying the faulty computations after an error is detected. Redundant modules that execute the same process simultaneously are used to mask the faulty module if an error occurs and to ensure the correctness of the system during recovery. This approach is expensive in terms of area overhead.

Another major recovery approach, *rollback recovery* [Bernstein 88, Hunt 87, Strom 88, Wu 90, Bowen 92, Pradhan 96, Huang 00b] is to roll the system back to a reliable state (called *checkpoint*) to retry the computations again. The state of the system at checkpoints are verified by the CED scheme and stored in a reliable storage, which can

be protected by Error-Correcting Codes (ECCs) or duplicate memory. Because the system is rolled back to the previous checkpoint, the rollback recovery scheme introduces degradation in the computation throughput. Hence, the rollback recovery latency, including the time required for loading the reliable state and the re-computing period for re-executing faulty computations, has to be estimated in order to determine if the throughput penalty is tolerable for the target application.

As a case study, we developed two rollback recovery techniques (see Appendix B for details) for the LZ compression application in reconfigurable hardware, based on the LZ encoder with inverse comparison CED described in Sec. 3.3. In LZ compression with inverse comparison CED, a checkpoint is defined at the instant when a maximum-length matching phrase is found and a codeword is generated accordingly. In this case, the system state at each checkpoint consists of the dictionary contents, which are obtained from previous source symbols.

The first scheme, *reload-retry,* is a typical rollback recovery approach that reloads the dictionary before retry. The desired dictionary state can be scanned in using the existing shift-register structure of the dictionary, and the area overhead of the reload-retry scheme is primarily caused by extra storage for the dictionary state at the previous checkpoint and input source symbols between two consecutive checkpoints. For LZ compression with 512-entry dictionary, such area overhead (in terms of the number of CLBs) measured in Xilinx XC4036XLA FPGAs is 16%, in addition to the area overhead for the inverse comparison CED scheme.

The error recovery latency of the reload-retry scheme is dominated by the time required to reload the dictionary, which is proportional to the number of dictionary entries. Statistical analyses show that the reload-retry scheme can recover the LZ encoder within one dictionary reload cycle, which corresponds to 32 μsec for a 512-entry dictionary reload at a clock rate of 16 MHz implemented in the case study of Sec. 3.3.

The second scheme, *direct-retry*, is to flush the dictionary contents when an error is detected and retry the faulty encoding process immediately with an empty dictionary. In this way, because the dictionary state is not stored and reloaded, the area overhead and recovery latency are both smaller. The area overhead (in terms of the number of CLBs) for LZ compression with 512-entry dictionary measured in Xilinx XC4036XLA FPGAs

is only 3%, in addition to the area overhead for the inverse comparison CED scheme. The worst-case recovery latency is $2L_{max}$ cycles, where L_{max} is the limit of the matching length in LZ compression. This corresponds to 8 μs for L_{max} = 63 at a clock rate of 16MHz.

Because LZ compression reduces the size of the data by finding matching phrases in the dictionary, the direct-retry approach results in compression ratio degradation during a period when the dictionary is not fully reloaded after the retry begins. For a 512-entry dictionary, simulation results show a compression ratio degradation of 22% during the first 512 cycles after the retry begins. Therefore, the reload-retry scheme is suitable for applications with a very high compression ratio requirement, while the direct-retry scheme is suitable for applications with a short recovery latency and very small area overhead requirement.

4.2 Recovery from Configuration Transients

For configuration transients, configuration readback and writeback mechanisms can be applied to restore the original, fault-free configuration [Carmichael 99, Huang 01a]. As mentioned in Chapter 2, both operations are executed in units of configuration frames.

The basic concept of the configuration transient recovery scheme is as follows. Configuration data read back from the reconfigurable hardware can be fed into a verification circuitry for error detection and correction. Once errors in configuration data are detected and corrected, correct configuration frames can be written back to complete the recovery.

The configuration transient recovery process can be illustrated by the following example using the dependable Dual-FPGA ACS system in Fig. 4-1, which is redrawn from Fig. 1-2(b) in Chapter 1. When the CED scheme of the application in FPGA1 detects an error, an autonomous system transient recovery scheme, such as rollback recovery or forward recovery, is initiated in FPGA1. In this way, the application in FPGA1 tries to recover itself from transient faults that do not alter the configuration. Meanwhile, FPGA2 observes the error signal from FPGA1 and starts reading

configuration frames stored in FPGA1. The system transient recovery process in FPGA1 can operate simultaneously with the configuration readback process initiated by FPGA2.

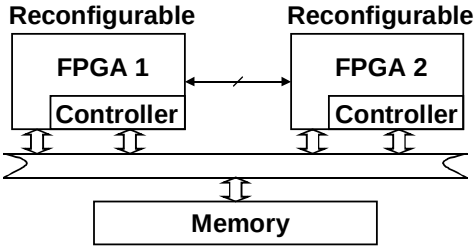

Figure 4-1: Dual-FPGA ACS system.

After receiving configuration frames from FPGA1, the controller in FPGA2 verifies them using the correct configuration information stored in a reliable storage. The verification process can be either a direct comparison with the original correct configuration data, or some ECC manipulations. If errors are detected, the faulty configuration frame can be corrected with the help of the fault-free configuration copy or ECC and can be written back to FPGA1. The case when the CED scheme in FPGA2 detects an error can be recovered in a reciprocal way.

As explained in Chapter 2, some data in configuration frames may represent memory contents in user applications because LUTs can also be configured as user memory modules. Such data vary dynamically with the progress of user applications, while other data in configuration frames that define logic functions and interconnect states in the hardware remain fixed. These dynamic contents should be masked from the configuration error recovery process, and errors in these user data are corrected by the system transient recovery schemes.

The *detection and correction cycle* of configuration transients refers to the time to complete the detection and correction of the entire configuration data. In [Carmichael 99], the detection and correction cycle of configuration transients in a Xilinx Virtex V1000 FPGA is 40ms, while the SEU occurrence rate is measured at one per hour across three FPGAs of this type. Therefore, with a very high probability, errors caused by configuration transients can be corrected using configuration readback and writeback operations within one detection and correction cycle.

The configuration transient recovery process can either be executed as a periodic background scrubbing operation [Carmichael 99], or be initiated simultaneously with a

system transient recovery process once an error in reconfigurable hardware is detected by a CED scheme [Huang 01a]. In either case, the configuration transient recovery process is executed concurrently with some system-level user operations in order to reduce the execution time overhead in user applications. Here, *system-level user operations* refer to all processes in reconfigurable hardware except the configuration readback and writeback processes for configuration transient recovery. Typical examples of system-level user operations are normal operations for user applications, CED operations for the applications, and system transient recovery operations.

4.3 Memory Coherence

4.3.1 Problem Definition

In the transient error recovery scheme for reconfigurable hardware, user memory data stored in LUTs can be accessed by two concurrent processes - the configuration transient recovery process and system-level user operations. Therefore, a *memory coherence* issue arises: we need to ensure that both concurrent processes see the same, most-updated copy of data.

Figure 4-2 illustrates an example of the memory coherence issue. In this example, suppose that an 8-entry LUT in the CLB at the first row of the first column (R1C1) of the CLB array is configured as a truth table of hexadecimal value *0x04* to represent a 3-input logic function. Also, the LUT in the CLB at the second row of the same column (R2C1) is configured as a RAM module. In Chapter 2, we have assumed that the configuration data frames are partitioned vertically such that each frame contains data from different rows in the same column.

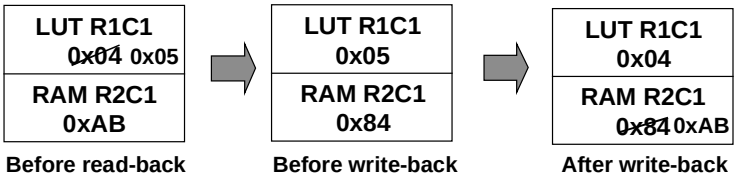

Figure 4-2: Example of the memory coherence problem.

Before the readback of the configuration data frames in the first column, let us assume that a configuration transient occurs in the FPGA and changes the value stored in LUT R1C1 from *0x04* to *0x05*. Also, the RAM R2C1 stores a user memory content *0xAB*

prior to the readback operation of the frame. These assumptions result in frames containing *0x05* in LUT R1C1 and *0xAB* in RAM R2C1 that are read back for the configuration transient recovery process.

The error in LUT R1C1 (*0x05*) in the frames read back can be detected and corrected in the subsequent verification circuitry, and a writeback operation of the faulty frames is required to complete the error recovery. The values of RAM R2C1 (*0xAB*) in the frames read back are masked for verification, and they are used in constructing the frames for writeback if no memory coherence protocol is used. As a result, the values in the first column after the configuration writeback operation become *0x04* in LUT R1C1 and *0xAB* in RAM R2C1, regardless of the system-level user operations between configuration readback and writeback.

To reduce the execution time overhead of the error recovery process, it is desired that the system-level user operations be only stalled during the writeback process of the frames. However, if the system-level user operations are not stalled between readback and writeback, the values in RAM R2C1 may change during this period. In the example in Fig. 4-2, suppose the system-level user operations update the values in RAM R2C1 as *0x84* before the writeback of the frames begins. If the writeback process is ignorant of the update, the old value *0xAB* in RAM R2C1 will be written back again. In this case, the two concurrent processes have different perspectives of RAM R2C1, and memory coherence is not preserved.

4.3.2 Previous Approach

[Kelem 00] suggested placing LUTs configured as user memory modules in different frames from LUTs configured as fixed logic functions. If the frames that contain LUTs configured as user memory modules do not contain any data that control the interconnect states, this method avoids the problem because the configuration error correction process can only change the frames that do not contain user memory modules. As a result, no write conflict to user memory modules between the configuration transient recovery and system-level user operations can happen. However, such a method imposes an extra constraint for placement and routing when users map their designs on the FPGAs. Such a constraint reduces the routing capability of FPGAs and increases path delays due to the separation of logic functions and local RAM or FIFO modules.

In the next sections, we present two protocols for solving the memory coherence problem on-line. The protocols include the stall-when-write strategy, and the dirty-bit strategy.

4.3.3 Stall-When-Write Strategy

An on-line solution without adding constraints on the placement and routing is to prevent system-level user operations from modifying user memory contents stored in the CLB column that is under configuration error detection and correction. This can be done by stalling the system-level user operations once a *write-enable* is signaled to a user memory module that is implemented in LUTs in the column under configuration error detection and correction. In this way, a CLB column remains unchanged during the configuration transient error detection for the column, and memory coherence in the column can thus be preserved.

Figure 4-3 shows the state diagram of this *stall-when-write* strategy for error detection and correction in configuration data. Initially, the system is in the "*normal*" state, which means that the configuration data readback operation runs without stalling system-level user operations. The system enters a "*stall*" state when a write operation (WR) is issued by system-level user operations to memory modules in LUTs in the column under configuration error detection and correction (Col). System-level user operations, including the write operation, are paused during the *stall* state. Memory coherence is thus preserved because the column under configuration error detection and correction is unchanged between the readback and writeback of its frames.

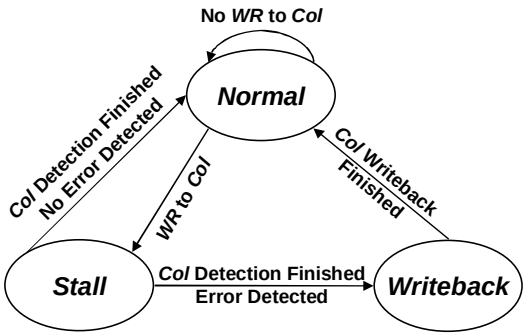

Figure 4-3: State diagram of the stall-when-write strategy.

The system leaves the *stall* state when the frames in the column under configuration error detection and correction are verified. If no error is detected, the

system returns to the *normal* state, and system-level user operations continue. If an error is detected, the system enters a "*writeback*" state for correcting errors in configuration data. In the *writeback* state, system-level user operations are still stalled until the writeback process of the faulty frames in the column finishes and the system returns to the *normal* state.

The drawback of the stall-when-write strategy is the extra time stalled when the system-level user operations write to memory modules in an error-free column that is read back for error detection. In this case, system-level user operations are stalled even though configuration writeback is not performed for the error-free column. This could result in large execution time overhead in user operations during the transient recovery process, especially when a large percentage of CLB columns contain memory modules, and the probability of memory writes in each cycle is high.

4.3.4 Dirty-bit Strategy

Practically, the number of frames that require a writeback operation in one configuration error recovery period for the entire reconfigurable hardware is small. This is because the SEU occurrence rate is in the order of one per hour or smaller, and the latency of configuring the entire FPGA is in the order of 10 to 100 ms in current technology [Carmichael 99]. Therefore, the first step towards minimizing the time overhead in system-level user operations is to avoid stalling memory writes that are issued to error-free columns under configuration error detection.

To achieve this objective, one can always insert a second readback period of a faulty frame before it is written back, and the system-level user operations are only stalled in the second readback and the writeback states. In this "*readback-again*" strategy, the user memory contents in the frame written back are always updated because of the second readback period, and system-level user operations are not stalled for memory writes to error-free columns that are read back for configuration error detection.

However, this readback-again approach still creates an unnecessary overhead by stalling the system-level user operations for the second readback of a faulty frame when the user memory contents in this frame are not updated since the first readback. This effect is more obvious when the number of memory writes per cycle in system-level user operations is small.

The extra overhead in the readback-again approach can be further reduced by adding a *dirty bit* to indicate the user memory status in the configuration transient recovery scheme. The dirty bit is set when the user memory contents stored in LUTs of the CLB column under configuration error detection are updated by system-level user operations since the most current readback. By adding a dirty bit, the second readback stage for a faulty column that requires configuration writeback is inserted only when the dirty bit is set. Therefore, stalls in system-level user operations due to the second readback stage are minimized.

The implementation of the dirty-bit memory coherence technique using the Dual-FPGA system in Fig. 4-1 is as follows. In Fig. 4-1, let us assume that the FPGA1 is the *checking device* and FPGA2 is the *device under check*. In this case, FPGA1 reads the configuration data frames from FPGA2 and start checking FPGA2. Also, FPGA1 monitors the memory addresses of write operations in the system-level user application processes of FPGA2 and checks if the addresses are in the column under configuration error detection and correction. In this way, because the checking device (FPGA1) has the information of memory activities of the device under check (FPGA2), a finite state machine (FSM) can be constructed in the checking device to control the memory coherence in the device under check according to the dirty-bit protocol.

The state diagram of the FSM for implementing the *dirty-bit* memory coherence technique in the configuration transient recovery process is shown in Fig. 4-4. During configuration readback and error detection, the system is in the "*Readback Normal*" state if system-level user operations have not modified user memory contents in the column under configuration error detection since the readback of the column. The system enters the "*Dirty*" state once system-level user operations modify user memory contents in the column under configuration error detection. System-level user operations are not stalled in either state.

If no errors are detected in the configuration frame, the system enters a "*Start Next Frame*" state to start reading back and detecting the next configuration frame. Otherwise, a "*Writeback*" state is necessary for correcting the faulty frame, and a "*Readback Stall*" state is inserted before the "*Writeback*" state in order to obtain the updated user memory contents only if the system is originally in the "*Dirty*" state.

System-level user operations are stalled in the "*Readback Stall*" state and the "*Writeback*" state.

In practice, a "*Wait*" state is inserted when an error is detected in a non-dirty frame in order to account for the race between the detection of the error and the memory write in user operations. Such race occurs because of the latency between the onset of the *write-enable* signal in user operations and the detection of this signal in the dirty-bit control FSM in the configuration transient recovery process. Detailed discussion can be found in Appendix C.

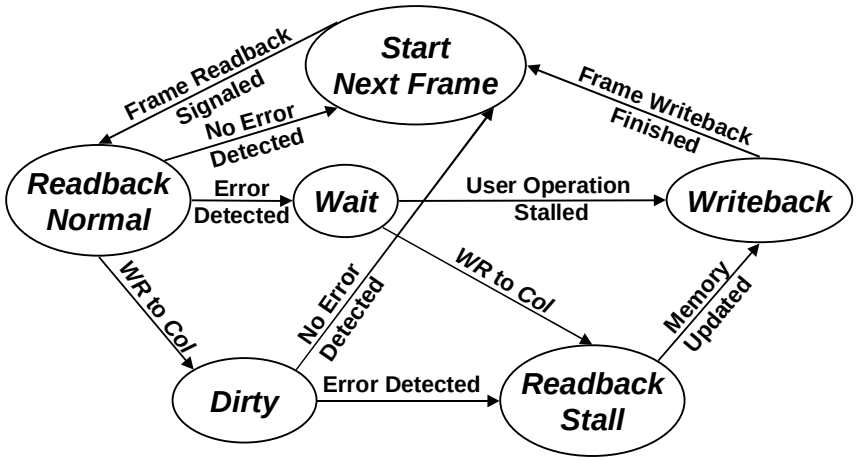

Figure 4-4: State diagram of the dirty-bit memory coherence technique.

4.3.5 Results

We analyzed and simulated the execution time overhead in the system-level user operations because of the stalls in the stall-when-write strategy and the dirty-bit strategy for memory coherence. In this section, we briefly present some important results. We assume a single faulty frame in an FPGA and measure the percentage of stalls in system-level user operations throughout the configuration transient recovery process of the entire FPGA. Detailed analyses and simulation descriptions can be found in Appendix C.

Figure 4-5 shows the simulation results of the time overhead in the two memory coherence strategies under the following environmental conditions: the readback latency and the writeback latency for a configuration frame are 30 cycles each, the FPGA has 36 CLB columns with 48 frames per column, 50% of the columns contain LUTs that are configured as user memory modules, and the number of memory write commands per cycle in the system-level user operations varies from 0 to 1. From Fig. 4-5, it is clear that

the time overhead in system-level user operations in the dirty-bit technique is under 0.1%, which is significantly better than the stall-when-write strategy.

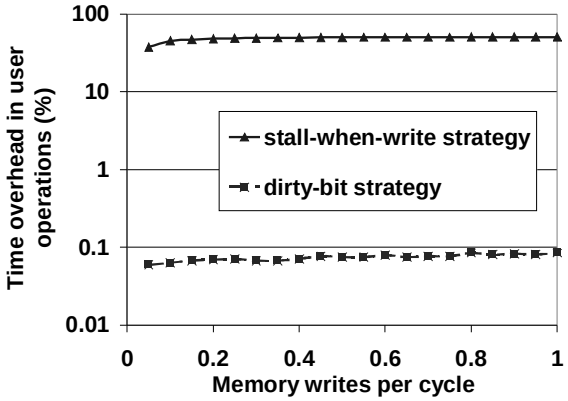

Figure 4-5: Time overhead comparison for memory coherence techniques.

Figure 4-6 show the simulation results of time overhead in system-level user operations in the dirty-bit technique with respect to different environmental parameters. Besides the number of memory writes per cycle in user operations, different environmental parameters in Fig. 4-6 are the percentage of CLB columns containing LUTs that are configured as user memory modules and the readback and writeback latency for one configuration frame. The resulting time overhead in system-level user operations are around 0.1%, and this overhead percentage changes little for different design environments.

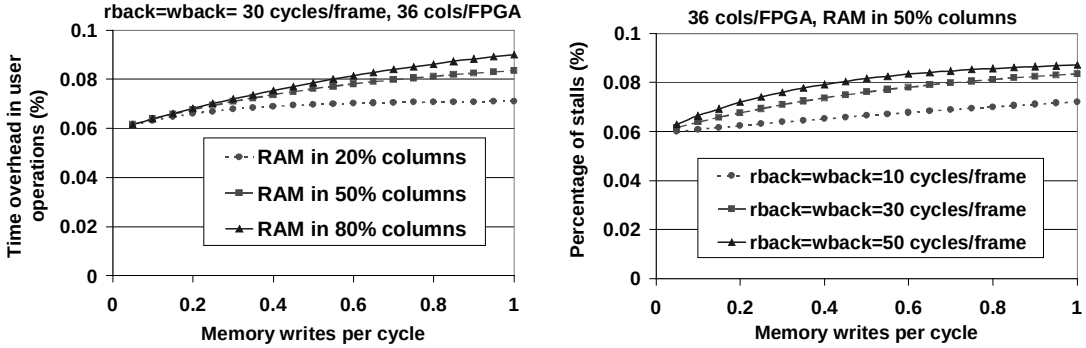

Figure 4-6: Time overhead in user operations for the dirty-bit technique.

We implemented the dirty-bit technique for the codeword buffer in the LZ encoder with inverse comparison CED described in Sec. 3.3 in order to estimate the area overhead. The codeword buffer is a dual-port FIFO and has 64 entries. Each entry in the buffer is 24-bit wide. Without the dirty-bit technique, we mapped the buffer in 153 CLB

slices, which span in four consecutive CLB columns, in the Xilinx Virtex V1000 FPGA. The resulting area overhead for implementing the dirty-bit technique is 37 CLB slices. This corresponds to 24% area overhead compared to the original buffer size, which is about only 1% extra CLB utilization in Xilinx V1000 FPGAs.

4.4 Summary

Transient errors in reconfigurable hardware can be classified into two categories: system transients and configuration transients. Errors caused by system transients can be corrected by traditional system-level recovery approaches, such as rollback recovery or forward recovery. Errors caused by configuration transients can be corrected by configuration readback and writeback processes. To minimize time overhead in user applications, the configuration transient recovery process is executed concurrently with system-level user operations, including normal operations in the application, CED operations, and system transient recovery operations.

However, because configuration frames may contain user memory data, a memory coherence issue arises when the configuration transient recovery process is executed concurrently with system-level user operations. To solve this problem, on-line memory coherence strategies, including the stall-when-write strategy and the dirty-bit technique, are presented without adding constraints on the placement and route of the user memory modules. Analyses and simulations show that the dirty-bit technique significantly outperforms the stall-when-write strategy when a small number of faulty configuration frames is present. The dirty-bit technique has a stable, small performance overhead under different environmental parameters.

Chapter 5
Reconfiguration for Fault Tolerance and Repair

Permanent faults, or *hard faults*, describe failures that are continuous and stable [Siewiorek 92]. A permanent fault can be caused by a physical defect missed at manufacturing test or due to wear-out during operation, or by a mistake in the design of the system. Because permanent faults persist in the system, transient error recovery techniques described in the previous chapter cannot restore normal operations without repair of the system.

As described in Chapter 1, reconfigurable hardware can achieve fault tolerance with fine-grained redundancy. Reconfiguration of the hardware, such that the faulty part is avoided and replaced, effectively repairs the system. Fast run-time reconfiguration is desired in order to reduce system downtime due to the repair operations.

In this chapter, we present reconfiguration techniques for fault tolerance in reconfigurable hardware. In particular, we summarize two column-based precompiled configuration techniques for minimizing post-fault-location system downtime due to reconfiguration and storage overhead for configuration data. Detailed discussions can be found in Appendix D.

We assume that the faulty part in the hardware has been diagnosed by some fault location techniques prior to reconfiguration. Fault location techniques are discussed in the next chapter.

5.1 Previous Work

Previous research on reconfiguration for fault tolerance in reconfigurable hardware can be broadly classified into two categories: (1) fast re-mapping and rerouting techniques for dynamic run-time generation of alternative configurations after error detection and fault location [Howard 94, Culbertson 97, Emmert 97, Emmert 98, Hanchek 98, Dutt 99, Mahapatra 99, Lakamraju 00], and (2) precompiled configuration approaches that create alternative configurations during the design phase and load the appropriate configuration after error detection and fault location [Mange 98, Lach 98, Lach 99].

5.1.1 Fast Re-mapping and Rerouting Techniques

Emmert and Bhatia proposed a minimax grid matching technique to re-map the functional units that are originally placed in faulty CLBs [Emmert 97]. They also proposed an incremental routing technique to reroute the corresponding signals [Emmert 98]. Lakamraju and Tessier proposed a localized swapping technique to tolerate faults within a CLB by replacing a faulty LUT (or flip-flop) with a local, reserved LUT (or flip-flop) in the same CLB [Lakamraju 00]. Dutt et. al. proposed node covering techniques that reserve spare CLBs and wire segments in order to facilitate the search for the optimal replacement for faulty parts [Dutt 99][Hanchek 98] [Mahapatra 99]. Also, a defect-tolerant, custom computing system has been built in Hewlett-Packard laboratory using run-time re-mapping and rerouting techniques in FPGAs [Culbertson 97].

The major problem with these techniques is the long latency for repair. Even though alternative configurations are generated by changing part of the original configuration, partial re-mapping and rerouting may still need minutes to hours to complete. Compared to the time required for downloading configuration bit-streams to an FPGA, which is of the order of hundreds of microseconds to several milliseconds, run-time re-mapping and rerouting represent a bottleneck for reducing system downtime. Besides, precise fault location is required before the re-mapping and rerouting operations begin. This is not always feasible and increases the system downtime for repair of failures.

5.1.2 Precompiled Configuration Techniques

To reduce the system downtime for repair, Lach et. al. [Lach 98, 99] proposed a tile-based precompiled configuration approach that moves the generation of alternative configurations to the design phase. The mapped circuitry is partitioned into tiles that represent functional blocks in the system. Alternative configurations for each tile are pre-generated and stored in the system. Also, Mange et. al. [Mange 98] and Emmert et. al. [Emmert 00] use precompiled configuration approaches based on shifting configuration columns.

The drawback of the precompiled configuration approach is the significant storage overhead in the system for all possible alternative configurations. Although the tile-partitioning approach in [Lach 98] reduces the storage requirement to some extent,

such storage overhead is still several times of the original configuration size. Moreover, previous precompiled configuration techniques do not suggest methods of how to create correlation and similarities in precompiled configuration data. This makes it difficult to reduce configuration storage by data compression techniques.

5.2 Column-Based Precompiled Configuration Techniques

In order to reduce the storage overhead due to precompiled configurations, data compression techniques that reduce redundancy in the information can be used. Most data compression techniques reduce data size by encoding correlated information with shorter codewords. Consequently, for a good compression ratio, the configuration data should be highly correlated.

For the data within one configuration version, correlation among configuration bit-streams depends on the application mapped in reconfigurable hardware. Some applications may show regular patterns in their configuration data, but others may have random bit-streams within each configuration. Therefore, high correlation is not always obtained within one configuration. However, for the data among different configuration versions, high correlation can be created intentionally by proper selections of re-mapping, rerouting, and configuration data partitioning methods when different configuration versions are constructed during the design phase.

As mentioned in Chapter 2, the reconfigurable hardware architecture has identical circuitry, routing resources, and configuration architecture in every CLB column. Therefore, our precompiled configuration approach is designed to create intentional similarities based on CLB columns in different configuration versions.

In order to create similar CLB columns in different configuration versions, we use a column-based shifting strategy similar to that in [Mange 98] and [Emmert 00], and design a new configuration encoding scheme to efficiently exploit such similarities among configurations for reducing storage overhead. According to the part of reconfigurable hardware used in the original configuration and alternative configuration versions, we can classify our column-based precompiled configuration technique into two schemes: the *overlapping* scheme and the *non-overlapping* scheme.

5.2.1 The Overlapping Scheme

The key concept of the overlapping precompiled configuration scheme can be illustrated by an example shown in Fig. 5-1. In Fig. 5-1(a), suppose that the original fault-free configuration, or the *base configuration*, is mapped in four consecutive CLB columns (column 1 to column 4). The *column-based functional modules* that are mapped in each of the four columns implement functions *A*, *B*, *C*, and *D*, respectively.

Let us consider the case for tolerating faults within any single CLB column. To this objective, we reserve one column outside of the mapped area (column 5) in the base configuration as backup resources for alternative configurations. All CLBs and switch matrices in the reserved column are unused. Also, the configuration of each column-based functional module in the mapped area is stored separately as a basis to construct alternative configurations.

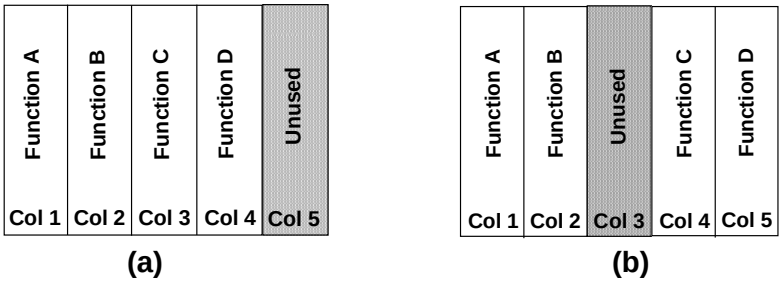

Figure 5-1: The overlapping scheme. (a) Base. (b) An alternative configuration.

In this case, four alternative configurations are required in order to guarantee 1-column fault tolerance in the FPGA. In each alternative configuration, one of the mapped columns (column 1 to 4) in the base configuration is intentionally unused. All functional modules originally mapped in CLB columns with smaller column indices than the intentionally unused column remain in the same places in the alternative configuration. The other functional modules are shifted rightwards by one column in the alternative configuration to avoid the intentionally unused column.

For example, Figure 5-1(b) shows one of the alternative configurations, where column 3 is intentionally unused. Functional modules mapped in column 1 and column 2 in the base configuration (function *A* and *B*) remain in the same places. Functional modules mapped in column 3 and column 4 in the base configuration (function *C* and *D*) are shifted rightwards to column 4 and column 5, respectively, to avoid using column 3.

In this overlapping scheme, the above re-mapping procedure in each alternative configuration creates several corresponding column sets. A *corresponding column set* is defined as a set of CLB columns in which certain functions are mapped in different configuration versions. Elements in a corresponding column set are referred to as *corresponding columns* in different configurations. For example, column 3 in Fig. 5-1(a) and column 4 in Fig. 5-1(b) belong to the same corresponding column set because they both implement function *C*, and they are corresponding columns of each other.

In Fig. 5-1(b), the re-mapping of the functional modules forms two *mapped regions* that are separated by the intentionally unused column. In this way, only *inter-region* signals that connect functional units in different mapped regions need to be rerouted due to the shifted functional modules. *Intra-region* signals that connect functional units within the same mapped region, however, can be routed in the same way as their counterparts in the corresponding base configuration column. This is because every CLB column has the same programmable logic and routing resources. Equivalently, routing for intra-region signals can be obtained directly by shifting the states of switches in corresponding base configuration columns using the same method described in the re-mapping procedure for column-based functional modules.

The detailed re-mapping and rerouting method for the overlapping scheme is described in Appendix D. In general, for *m*-column fault tolerance in the overlapping scheme (i.e., the *number of tolerable faulty columns* is *m*), we need to reserve *m* unused columns for backup in addition to the mapped area in the base configuration. If *the number of column-based functional modules in the base configuration* is *k* (i.e., *k* CLB columns are used to map the original circuit in the base configuration), the overlapping scheme with *m*-column fault tolerance requires (*k*+*m*) CLB columns in the FPGA. In each configuration, *m* out of (*k*+*m*) CLB columns are intentionally unused. The total number of configurations required is thus $C(k+m, m) = (k+m)! / (m!k!)$, with one version being the base configuration. Therefore, one needs to construct $[C(k+m, m) - 1]$ alternative configurations in order to achieve *m*-column tolerance in a *k*-column circuit.

As described in Appendix D, horizontal wires that connect CLBs of multiple blocks apart can be used in rerouting inter-region signals. One drawback of this scheme is that the maximum number of horizontal routes used in each column is limited by the

number of horizontal multiple-block wires available to reroute inter-region signals. Nevertheless, the utilization of routing resources for vertical wires is not limited by this rerouting scheme because all vertical connections are intra-region and need not be rerouted. Hence, to satisfy the horizontal routing constraint, the base configuration should be constructed such that most of the signals flow in the vertical direction.

The advantage of this scheme is that different configurations are very similar because each alternative configuration is created by shifting part of the base configuration in units of columns. More similarity in configuration data leads to a good compression ratio, and thus small storage overhead, when data compression techniques are applied. This property will be further examined in Sec. 5.3 when we discuss the configuration data compression technique

5.2.2 The Non-overlapping Scheme

If the target circuitry is small enough to fit within half of the FPGA, a simple way to construct alternative configurations is to shift the entire mapped circuitry to originally unused regions. This is the *non-overlapping* precompiled configuration scheme, where there is no overlap between the base configuration and alternative configurations.

In order to tolerate up to m faulty columns in the FPGA, the total number of alternative non-overlapping configurations required in addition to the base configuration is m. In this case, the base configuration has to be mapped within $1/(m+1)$ of the entire FPGA columns. Therefore, one drawback is the limitation in the size of the mapped region in the base configuration.

Still, the non-overlapping scheme can be feasible in situations where various applications are implemented in an FPGA in a time-multiplexed manner to reduce cost. In such cases, the FPGA is chosen to accommodate the largest circuit size among all target applications. Because of variations in circuit sizes, there may be small application circuits in which the size constraint in the non-overlapping scheme is satisfied.

Compared to the overlapping scheme, the non-overlapping scheme has smaller storage overhead for alternative configurations. This is caused by two factors. First, there are only m alternative configuration required for m-column fault tolerance, which is fewer than the $[C(k+m, m) - 1]$ configurations in the overlapping scheme.

Second, the non-overlapping scheme results in more similarity among different configuration versions because the relative positions among the mapped column-based functional modules are preserved. In this way, the only difference between the corresponding columns in different configurations is from the connections to primary inputs and outputs of the device. More similarity between configurations result in a better compression ratio, which will be examined in Sec. 5.3.

5.3 Storage Reduction Techniques for Configuration Data

5.3.1 Integrated Differential and Run-length Coding

In our column-based precompiled configuration schemes, the only difference among corresponding columns in different configurations is the inter-region reroutes and the wires to the primary I/O's of the device. In this case, a *difference vector*, which is computed by a bit-wise XOR between a column in an alternative configuration version and its corresponding column in the base configuration, contains strings of long, consecutive 0's and scattered 1's. Therefore, *run-length coding*, which uses the length of leading 0's in the input string to represent the string, is expected to be very effective in reducing the required storage size of difference vectors for alternative configuration columns.

We use *Golomb codes with group size g* [Golomb 66] to encode difference vectors between corresponding columns in the base configuration and alternative configurations. We do not need to store the mapping relationship of corresponding columns (e.g., the information of "Which column in the base configuration is the corresponding column of a certain alternative configuration column?"). This is because the intentionally unused columns are already specified in each alternative configuration, and we can thus calculate the amount of rightward shifting for each base configuration column and derive the mapping information of corresponding columns.

The detailed encoding algorithm for Golomb code with group size g can be found in Appendix D, and an example with $g = 4$ is shown in Table 5-1. Generally, such an encoding scheme transforms a $(z+1)$-bit string of z leading 0's followed by a 1 into a codeword of two parts: (1) $\lfloor z / g \rfloor$ leading 1's followed by a 0, and (2) the $\log_2 g$-bit,

binary representation of (z mod g). When the run-length of 0's in the source string is significantly larger than the group size g, the compression ratio is approximately $1/g$.

The compression ratio improves with increasing run-length of 0's in the source string. Because of intentional similarities created among different configurations, the column-based precompiled configurations are expected to have long run-lengths in difference vectors, and thus, a good compression ratio. Using run-length coding, the non-overlapping scheme achieves better data compression than the overlapping scheme due to greater similarity in the corresponding CLB columns. Experimental results are described in the next section.

In addition to a good compression ratio, another factor that makes the proposed differential and run-length coding suitable for encoding configuration data is the simple decoding process. The detailed procedure for decoding is described in Appendix D. The implementation of the decoding process in hardware requires only a counter to compute the run-length and XOR circuitry to reconstruct alternative configuration columns from difference vectors.

Table 5-1: Example of Golomb code with group size = 4.

Run-length z	Source String S_z	Codeword
0	1	000
1	01	001
2	001	010
3	0001	011
4	00001	1000
5	000001	1001
6	0000001	1010
7	00000001	1011
8	000000001	11000
9	0000000001	11001
10	00000000001	11010
11	000000000001	11011

Note that corresponding columns can be configured exactly in the same way for different configuration versions. For example, in the case of Fig. 5-1, the CLB column that holds the function A can have the same settings for the base configuration and any alternative configuration where no inter-region reroute is required for this function.

Therefore, for each alternative configuration, we store the pointers to the encoded difference vectors instead of the actual data of such vectors. In this way, the actual data of each encoded difference vector are stored only once, and different configuration versions can share the same difference vector without redundant storage. In addition, for each alternative configuration, we store the column indices of intentionally unused columns that are required to locate the corresponding base configuration columns.

5.3.2 Experimental Results

To demonstrate the storage overhead improvement for alternative configurations, we simulated the overlapping and non-overlapping column-based precompiled configuration schemes with 1-column tolerance capability (m = 1) to a subset of the MCNC benchmark circuits[1]. Because the benchmark circuits are relatively small compared to the capacity of current-generation FPGAs, we used the smallest FPGA in Xilinx Virtex-E series, XCV50E [Xilinx 01], in our experiments. This type of FPGA has a 16x24 CLB array in each device.

The number of CLB columns (parameter k) used in the base configuration for each benchmark circuit is shown in Table 5-2. In the base configuration, we mapped the entire circuit and placed the I/O's in the left part of the chip compactly. In this way, we can reduce the number of alternative configurations required in the overlapping scheme and minimize the routings in the horizontal direction.

In our experiment, the configuration storage overhead is calculated relative to the portion of the configuration data required for the used columns in the base configuration, instead of the overall configuration bit-stream for the device. This is to avoid over-optimistic results because of the small size of the benchmark circuits relative to the capacity of the FPGA. In this way, experimental results for these small benchmark circuits can still be good estimates of configuration storage overhead percentages for larger circuits.

Because the number of tolerable faulty columns in the experiments is one, k alternative configurations for the overlapping scheme and one alternative configuration

[1] In our experiment, we rerouted inter-region signals in each alternative configuration manually because of the lack of support for setting rerouting path constraints in current commercial place-and-route tools. Therefore, we choose small MCNC benchmark circuits in the experiment for easier control in rerouting.

for the non-overlapping scheme are required, respectively. Therefore, without data compression, the storage overhead for configuration data relative to the size of the base configuration is ($k \times 100\%$) in the overlapping scheme and 100% in the non-overlapping scheme.

Table 5-2: Size of benchmark circuits.

Circuit	Number of CLB columns used in the base configuration (k)
c499	11
duke2	10
planet1	6
sand1	5

To generate alternative configurations, we manipulated the Xilinx Design Language (XDL) files [Xilinx 01] of base configurations according to the column-based shifting methods. XDL files describe the physical mapping of functional units and the routing of nets in FPGAs in text format. The resulting XDL files for both base configurations and alternative configurations can be translated into configuration bit-streams, which are processed according to the configuration architecture in [Xilinx 00] in order to extract the configuration data of corresponding columns. The extracted configuration data for each column is then encoded as difference vectors and compressed using Golomb codes with different group sizes.

Figure 5-2 and 5-3 show the resulting storage overhead for the benchmark circuits due to the encoded alternative configurations in both precompiled configuration schemes, respectively. From Fig. 5-2 and Fig. 5-3, a group size of 128 in the Golomb code minimizes the storage overhead of alternative configuration versions in both precompiled configuration schemes. For the overlapping scheme, the resulting minimum storage overhead of alternative configurations is in the range of 15-35% for the benchmark circuits. For the non-overlapping scheme, the minimum storage overhead of alternative configurations is around 2-6% only.

As described previously, the small configuration storage overhead in the non-overlapping scheme comes at the expense of more FPGA area reserved in each configuration version. Compared to the multiple-time configuration storage overhead without compression in previous approaches, our schemes achieve one to two orders of magnitude improvement in storage requirement for alternative configurations.

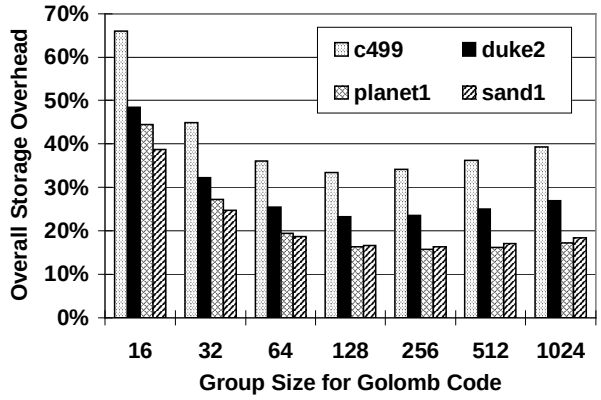

Figure 5-2: Storage overhead for the overlapping scheme.

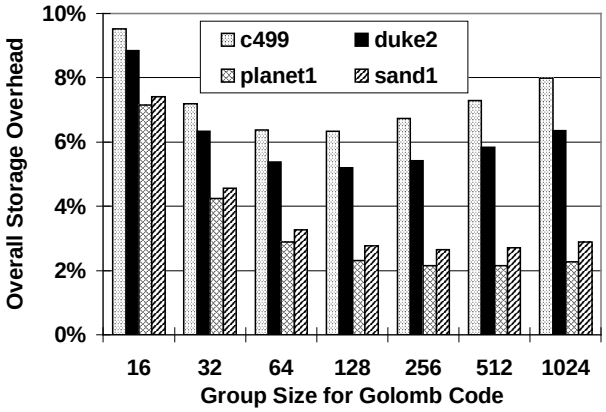

Figure 5-3: Storage overhead for the non-overlapping scheme.

In Fig. 5-4, we compare our integrated differential and run-length coding scheme with the commonly used *gzip* data compression utility in the *Unix* operating system for compressing alternative configurations in the overlapping scheme. The data for gzip compression are acquired by compressing each individual configuration and calculating the total alternative configuration data size after compression. In Fig. 5-4, it is shown that our scheme achieves three to five-times improvement in compressing alternative configuration data. This result further supports our claim in Sec. 5.2 that a good compression ratio can be achieved by exploiting the intentional similarity among alternative configurations.

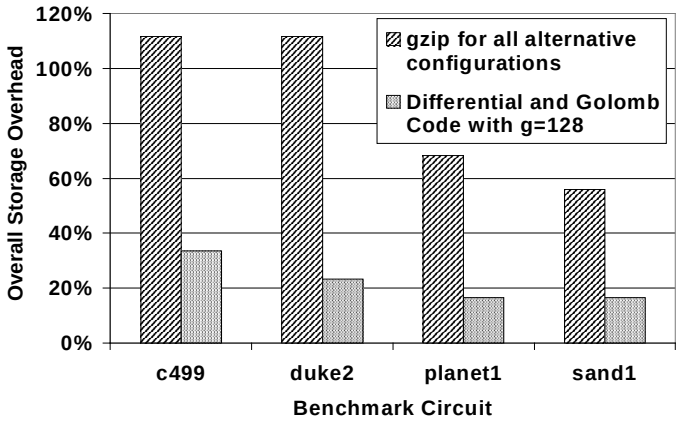

Figure 5-4: Comparison of gzip and the integrated differential/Golomb code.

5.4 Performance Impact

Shifting CLB columns in alternative configurations can change the path delays of the circuit. For vertical paths and intra-region connections, path delays are unchanged because these paths are not rerouted. For inter-region paths, however, there can be delay changes due to rerouting.

In order to evaluate the performance impact of alternative configurations, we computed the maximum combinational path delay in each configuration for the MCNC benchmark circuits in Table 5.2. The results, as reported by the timing analyzer tool in Xilinx Alliance 3.1i software [Xilinx 01], are shown in Fig. 5-5. Each curve represents the results for a benchmark circuit. Compared to the base configuration, the worst-case critical path delay overhead in the alternative configurations range from 11% to 18%. Also, for some alternative configurations where the critical path is not changed due to rerouting, there is no performance degradation after reconfiguration.

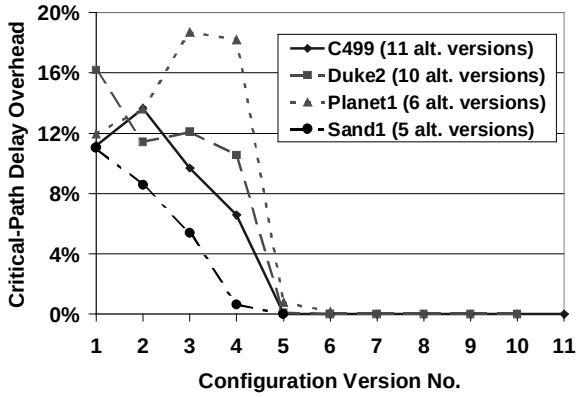

Figure 5-5: Critical-path delay overhead in alternative configurations.

5.5 Dependability Improvement

The dependability improvements of the proposed schemes can be analyzed using the parameter Mean Time to Failure (MTTF). *MTTF* is defined as *the expected time of the first failure in the system, given successful startup at time zero* [Siewiorek 92]. In this section, we present the dependability improvement results. Detailed analysis can be found in Appendix D.

Figure 5-6 shows the *normalized MTTF* for the overlapping and non-overlapping schemes for different numbers of tolerable faulty columns, *m*. The MTTF's for both schemes are normalized to the MTTF of the base configuration without fault tolerance techniques. Note that the result for the overlapping scheme is dependent on the circuit size *k*, while the result for the non-overlapping scheme is not. In Fig. 5-6, we choose three different circuit sizes (k = 10, 20, and 50) for the overlapping scheme. As a comparison, current-generation FPGAs, such as the Xilinx Virtex-E family, have choices ranging from 24 to 156 CLB columns in a chip.

In Fig. 5-6, it is clear that both schemes can achieve significant MTTF improvement. With the same number of tolerable faulty columns, the overlapping scheme has a better MTTF than the non-overlapping scheme. Intuitively, this can be explained in two ways. First, in order to guarantee *m*-column fault tolerance for a *k*-column circuit, the non-overlapping scheme requires more CLB columns (totally $(m+1)k$ columns) to implement than the overlapping scheme (totally $(k+m)$ columns). A greater number of CLB columns used in the non-overlapping scheme results in a larger area and becomes more susceptible to faults.

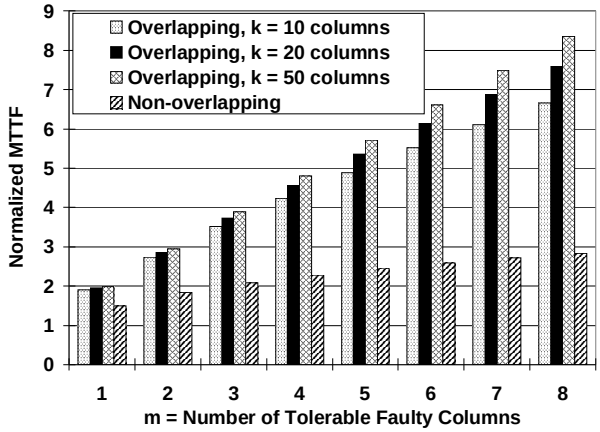

Figure 5-6: Normalized MTTF.

Second, given the same number of columns, N_{col}, in the reconfigurable hardware for implementing a k-column circuit using both schemes, the overlapping scheme is guaranteed to tolerate ($N_{col} - k$) faulty columns, whereas the non-overlapping scheme is guaranteed to tolerate only ($\lfloor N_{col} / k \rfloor$ - 1) faulty columns. The difference is more noticeable when the circuit size, k, is large.

Figure 5-7 shows the normalized MTTF increment for each additional column of tolerance in both schemes. The *normalized MTTF increment* represents the incremental MTTF improvement resulting from increasing the fault-tolerance capability by one more column in each scheme. The result follows the law of diminishing return, which indicates that the maximal gain in MTTF improvement is obtained when the system is changed from a no-recovery scheme to a scheme with 1-column tolerance capability.

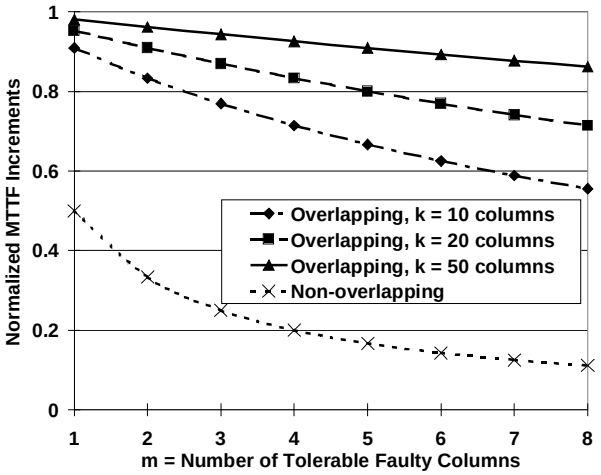

Figure 5-7: Normalized MTTF increments.

5.6 Summary

In this chapter, we presented two column-based precompiled configuration techniques, the overlapping scheme and the non-overlapping scheme. By creating alternative configurations that avoid certain parts of the original mapped area in the reconfigurable hardware during the design phase, the precompiled configuration approach improves the dependability significantly and permits fast reconfiguration for reducing system downtime. Because similarity among alternative configurations is intentionally created, the storage overhead for alternative configurations in both schemes

is reduced by orders of magnitude using differential and run-length coding. The critical-path delay overhead in alternative configurations is also measured.

The overlapping scheme has a better MTTF improvement, while the non-overlapping scheme achieves smaller storage overhead for alternative configurations but needs more CLB columns for implementation.

Chapter 6
Run-Time Fault Location Techniques

Run-time fault location in reconfigurable hardware is important because the resulting diagnostic information can be used to determine how to reconfigure the hardware for tolerating permanent faults. In order to minimize system downtime and increase availability, a fault location technique with very short diagnostic latency is desired.

The choice of run-time fault location techniques for reconfigurable hardware is highly dependent upon the reconfiguration scheme used for tolerating permanent faults in the system. For example, if the dynamic re-mapping technique in [Emmert 97, Emmert 98] or the tile-based precompiled configuration scheme in [Lach 98, Lach 99] is used for tolerating faulty CLBs, a fault location technique with diagnostic resolution of one CLB is required. Alternatively, if the localized swapping technique in [Lakamraju 00] is used, it is necessary to locate the faulty resource within a CLB.

In this chapter, we briefly describe an integrated technique for fast run-time fault location and fault tolerance in reconfigurable hardware. Based on the CED techniques in Chapter 1 and the column-based precompiled configuration techniques presented in Chapter 5, our approach can achieve significant availability improvement by minimizing the number of reconfigurations required for fault location and recovery. Detailed discussions can be found in Appendix E.

6.1 Previous Techniques

Most of the papers on FPGA fault detection and location [Jordan 93, Lombardi 96, Stroud 97, Stroud 98, Renovell 98] focus on production test of FPGAs. Application dependent fault-location approaches have been described in [Mitra 98, Das 99]. While the diagnostic resolution of these approaches is very fine-grained (e.g., faulty CLBs), the number of reconfigurations (and hence, the fault-location time) can be large.

Abramovici et. al. proposed a roving STAR approach for fault detection and location during system operation [Abramovici 99, Emmert 00]. They suggested using CED techniques for detection of transient failures. For detection of permanent faults, two

rows and two columns of the FPGA are reserved as the self-testing area (STAR). The STAR areas are not used for user applications and are tested while the rest of the system is in operation. After the testing of a STAR is complete, the STAR is moved to a new location by downloading a new configuration, and the process iterates until the entire FPGA is completely tested. Complete testing of a STAR in a Lucent ORCA 2C15A FPGA [Lucent 01] takes about 31 seconds, and the system clock has to be stopped for about three to four seconds for copying the state before the STAR moves to a new location [Emmert 00].

One concern with the roving STAR approach is the performance and availability degradation of the system because of the moving of the STAR, even when failures are not present. Another concern with the roving STAR technique is the error latency. It has been reported in [Emmert 00] that it takes approximately six minutes before the STAR has roved the entire ORCA 2C15A FPGA, which has a programmable logic array of 20 columns and 20 rows. The capacity of logic gates in such FPGAs is about 80% of the smallest Xilinx Virtex FPGAs. Hence, unless CED techniques are used, it may take a long time before the error is detected in an FPGA. The above problems can be overcome if fine-grained CED techniques are used combined with fault location techniques that depend on checker responses. Using the STAR approach without utilizing checker data can be very expensive in terms of fault-location time and hence, system downtime.

6.2 The Blind Reconfiguration Technique

In the column-based precompiled configuration approach of Chapter 5, alternative configurations for repair are pre-generated and stored in the system before a permanent fault occurs. Thus, one simple way to avoid the fault while repairing the system is to try all possible configurations alternately until a configuration that successfully avoids the fault is loaded. For each configuration, CED schemes for the application circuitry are used to ensure data integrity and determine if the attempt is successful. In a sense, this "*blind*" reconfiguration scheme replaces high-complexity fault location techniques with CED schemes at run-time.

There are several ways to generate patterns for testing each reconfiguration attempt. First, if the rollback recovery technique described in Chapter 3 is used in the

system for system transient recovery, one can use the same method to retry the input sequence that fails in the original computation. Although the test using rollback retry is closely integrated with the transient recovery scheme, such a test sequence is not complete. There may be faults escaping the test because their effects do not propagate to the output observed by the CED checker during the retry in an alternative configuration attempt. This effect is further discussed in Sec. 6.2.1.

The second method is to use *pseudo-exhaustive BIST* (PE-BIST) patterns to test each sub-circuit in the target application exhaustively [McCluskey 81]. Such patterns provide very high fault coverage without relying on explicit fault models. Also, they do not require large memory to store the test patterns because a minimal length PE-BIST pattern can be generated using a simple test pattern generator [McCluskey 82]. The test pattern generator can be implemented in the controller of the other FPGA in the Dual-FPGA scheme.

The third method is to use a *linear feedback shift register* (LFSR) to apply *pseudo-random* test patterns to the target application circuitry [Abramovici 90]. Like PE-BIST, this approach does not require memory overhead to store test patterns, and the test patterns can be easily generated by the controller of the other FPGA in the Dual-FPGA architecture. The fourth method is to use the *functional verification pattern* of the target application in the reconfigurable hardware. Such a pattern verifies the correct functioning in the application, but it requires memory overhead to store the patterns.

For this blind reconfiguration approach, the major area overhead compared to the original application circuitry includes the extra storage space for precompiled configurations and the area for the CED scheme. However, both parts of the overhead are inherent either in the reconfiguration for fault tolerance and repair scheme or in the error detection scheme that is necessary for constructing a reconfigurable, dependable computing system. Therefore, this fault location technique does not cause major area overhead in a system that already has some CED scheme for data integrity and the precompiled configuration technique for fault tolerance.

In order to reduce the diagnostic time, one should choose a CED scheme with small error detection latency. For example, duplex CED that compares the results from duplicated modules in every cycle is good for minimizing the latency. On average, if

there are *k* alternative configurations available in a *k*-column circuit with 1-column fault tolerance in the overlapping precompiled configuration scheme, we need to try *k*/2 configurations assuming a single fault in the system. Therefore, for small circuit size, this blind reconfiguration approach has the potential of reducing system downtime caused by fault location compared to previous techniques.

Compared to the roving STAR approach in [Abramovici 99], this blind reconfiguration scheme guarantees data integrity and does not impose performance and availability degradation in fault-free operations. The control of the blind reconfiguration process is also simpler, and thus more feasible for dependable systems with autonomous recovery based on reconfigurable hardware. However, there are some issues related to the blind reconfiguration scheme, and we discuss these issues in the following subsections. The issues include the effect of error detection capability in the CED scheme, the precision of the fault location result, and the order of configuration attempts.

6.2.1 Effect of Error Detection Capability in CED

In the blind reconfiguration approach, the existing CED scheme is used to determine if a reconfiguration attempt is successful. Because the CED scheme is also used during operation before and after reconfiguration, data integrity is always preserved even though some faults may escape a test pattern sequence during a reconfiguration attempt, providing faults present only in the originally used part of the reconfigurable hardware.

However, during the normal operation of the original configuration, permanent faults may occur in the reserved part of the reconfigurable hardware and become *dormant* faults. In this case, when a permanent fault occurs in the used part in the original configuration, such a single fault in the original mapped circuitry causes multiple faults in some alternative configuration attempts. Figure 6-1 illustrates this effect of dormant faults in the original configuration. The shaded area is unused, and faulty areas are marked with “x”. In this case, because of a dormant fault at (row 2, column 3), a single fault in the original mapped region at (row 1, column 2) causes multiple faults when the alternative configuration in Fig. 6-1(b) is attempted.

Note that this situation occurs when there are multiple permanent faults in the hardware. To alleviate the effect of dormant faults, CED schemes with high coverage of

multiple faults should be used. For example, *diverse duplex* systems with different implementations of the same logic function are good candidates because they provide better protection against multiple faults than other application-independent CED techniques [Mitra 00a].

Figure 6-1: Effect of dormant faults. (a) Original. (b) Alternative configuration. (Shaded area represents unused elements.)

6.2.2 Fault Location Precision

One important feature of the blind configuration attempts is that the result does not identify a precise fault location in the hardware. Instead, it finds a successful alternative configuration rapidly to resume the normal operation of the target application. The actual fault may not occur in intentionally unused columns in a successful configuration attempt.

Figure 6-2 shows an example of an unidentified fault location. Because the configuration data is shifted in units of columns, the faulty block (row 2, column 2) is unused in the alternative configuration in Fig. 6-2(b) where column 1 is intentionally unused. If we assume that a fault in an unused block does not affect the normal operation, this alternative configuration attempt will be considered successful even though the actual fault is not in column 1.

Figure 6-2: Unidentified fault location. (a) Original. (b) Alternative configuration. (Shaded area represents unused elements.)

Because the actual fault location is not necessarily in intentionally unused columns in a successful configuration attempt, we should try all precompiled configurations except for the ones that fail in avoiding current faults when the next permanent fault occurs. For example, in Fig. 6-2, other configurations that use column 1 should not be completely excluded in the attempts when the next permanent fault occurs. This is to avoid the degradation of fault-tolerant capability (measured by the number of tolerable faulty columns in the reconfigurable hardware) in the column-based precompiled configuration scheme.

Note that with the use of the proposed run-time fault location technique, the system can postpone complicated, off-line diagnostic processes until it is idle. In this way, even though the total system downtime due to the detailed diagnosis is unchanged, the system has more flexibility in scheduling the maintenance while preserving the most availability when the load is active.

Once the knowledge of precise fault locations is obtained from the scheduled diagnosis, adaptive reconfiguration attempts that include only configurations without using resources in these locations can be applied when the next fault occurs on-line. This can reduce the number of configuration attempts, and thus reduce the overall downtime to recover the system from the next occurrence of faults.

6.2.3 Order of Configuration Attempts

The most crucial block to guarantee data integrity in a CED scheme is the checker. Therefore, the order of configuration attempts should be carefully arranged to ensure correct functioning of the CED checker.

Figure 6-3 shows an example of the impact of reconfiguration order on data integrity. Suppose the checker is mapped in the A3 block, and the faulty block is B3 in the original configuration. If the first configuration attempt avoids column 1, the checker is shifted to the faulty block and the checker output may be stuck at zero (error-free). In this way, the CED scheme fails to function correctly, and data integrity is not preserved in subsequent operations.

To avoid the corruption in the checker output, *self-checking checkers* should be used [McCluskey 90]. Such checkers guarantee data integrity for any single fault at the checker outputs. Another solution to guarantee the correct functioning in the checker is

to try configurations without shifting the checker column (denoted as $Conf_{same\_checker}$) first. The system tries configurations that shift the checker column (denoted as $Conf_{shift\_checker}$) only when none of $Conf_{same\_checker}$ is successful.

Figure 6-3: Effect of reconfiguration order. (a) Original. (b) Alternative configuration.

Because the checker remains in the same position for $Conf_{same\_checker}$, the detection capability of the checker is identical to that of the normal operations. Also, the destination column for the checker in $Conf_{shift\_checker}$ (e.g., column 2 in Fig. 6-3) is intentionally unused in one of the configurations in $Conf_{same\_checker}$. If the destination column for the checker is the only faulty column, one of the configurations in $Conf_{same\_checker}$ can avoid the fault and becomes a successful attempt. The problem illustrated in Fig. 6-3 can thus be avoided by such ordering of configuration attempts.

6.3 Minimization of Reconfigurations

6.3.1 Distributed CED Checkers

There are different levels of granularity at which CED can be performed. For reconfigurable systems, since it is possible to repair the system rather than replacing the faulty chip or the faulty board, it is reasonable to implement CED at a fine granularity level of *sub-circuits* in the system. For example, Fig. 6-4 shows the architecture of a duplex CED scheme with distributed checkers in each stage of a pipelined system. Other system-level architectures with fine-grained CED have been described in [Mitra 00a].

In Fig. 6-4, the functional modules and registers in each stage of the pipeline are duplicated, and a local, distributed checker in each stage is used to compare the

duplicated outputs and to signal an error if a mismatch occurs. Distributed error signals can be combined to form the global error signal of the system (not shown in Fig. 6-4).

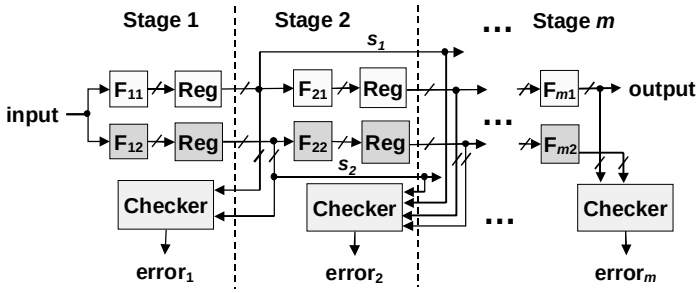

Figure 6-4: Duplex CED with distributed checkers in a pipelined system.

Because pipeline registers are used to separate different stages in the system, errors signals from distributed CED checkers can localize faults within part of the system. If the CED checker that checks the *i*-th stage outputs does not signal an error in a certain cycle, correctness of computations in the *i*-th stage in this cycle can be assumed. In this way, we can reduce the number of suspect faulty columns in the reconfigurable hardware and the number of configuration attempts.

Signals that propagate across more than two sub-circuits (e.g., signal s_1 and s_2 in Fig. 6-4) are also checked in intermediate sub-circuits if they are routed by PIPs in such intermediate sub-circuits. In this way, an open in the middle of a long, global signal can be localized according to error signals reported by the checkers of intermediate sub-circuits.

The outputs of the distributed checkers in each sub-circuit can be stored in flip-flops, which can be connected in a scan chain. The idea of using a scan chain to scan out the checker outputs is also used in the IBM mainframes to locate the faulty chip or board [Kraft 81, IBM 01]. When the system signals an error, the contents of the flip-flops storing the checker outputs can be scanned out for further processing by the controller in the other FPGA of the Dual-FPGA architecture. The scanned data can be read out through a dedicated Scan Out pin or through the boundary scan port generally available in FPGA chips. If the number of checkers is low, dedicated I/O pins can be used to directly observe the checker outputs.

The controller stores the information about the CLB columns that are occupied by each sub-circuit in the system. Such information is directly available from CAD tools,

such as Xilinx Alliance software [Xilinx 01], at the design phase through the floorplan of each sub-circuit and checker in the FPGA. After the controller scans out the checker data, it can execute the following simple routine to find the set of suspect faulty columns:

```
Suspect = ∅
For each checker output do
    If the checker produces an error signal
        Suspect = Suspect ∪ {Columns occupied by the sub-circuit that is
            checked by the checker}
    Endif
Endfor
```

If the suspect set contains only one CLB column, the configuration to be loaded is the one that does not use the suspect column. If the suspect set contains multiple CLB columns, the configurations to be loaded are restricted to those that do not use at least one of the suspect CLB columns.

Suppose that there are m sub-circuits in the system, each of which has a localized, distributed CED checker. For this distributed checker scheme, the worst-case number of configuration attempts for avoiding single fault that does not occur on wires across multiple sub-circuits is $max(k_1, k_2, \ldots, k_m)$, where k_i is the number of CLB columns occupied by the i-th sub-circuit. The worst-case number of configuration attempts for avoiding single fault that occurs on a wire across multiple sub-circuits is the summation of k_i's in every sub-circuit along the path of the wire.

Note that since the faulty column suspects are specified by distributed checkers, our technique can also be integrated with the roving STAR approach in [Abramovici 99] in order to find the precise fault location. The only difference is that, instead of roving the STAR across the entire FPGA, we only need to rove the STAR across the suspect columns. The resulting number of reconfigurations required in the roving STAR approach is thus reduced when integrating with the distributed CED checkers in our approach.

Generally, for fine-grained partitioning of sub-circuits, the number of configuration attempts is smaller than systems with coarse-grained sub-circuits. This is because each sub-circuit occupies fewer columns in the hardware for systems partitioned

with finer granularity. However, there is a possible tradeoff with area overhead because of the increasing number of distributed checkers in the system because of fine-grained partitioning. Detailed discussions of this issue can be found in Appendix E.

6.3.2 Modified Column-based Precompiled Configuration Scheme

By floorplanning the design, each sub-circuit and its corresponding CED checker can be confined within certain columns in the hardware. In this way, we can minimize the number of configuration attempts by using a modified column-based precompiled configuration scheme, which is illustrated in Fig. 6-5.

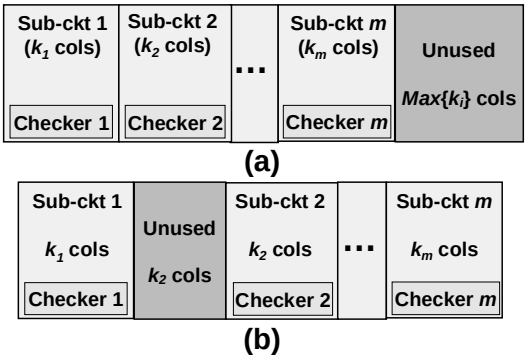

Figure 6-5: Modified column-based precompiled configuration. (a) Base. (b) Alternative configuration when checker 2 reports errors.

In Fig. 6-5, alternative configurations are created by shifting the logic mappings in units of sub-circuits, instead of shifting in units of CLB columns. Each alternative configuration avoids the entire mapped region of a sub-circuit in the original configuration by such sub-circuit-based shifting. Because the suspect faulty columns are formed based on the distributed checkers in each sub-circuit, the number of configurations to be loaded is minimized by such sub-circuit-based shifting strategy.

For single fault that does not occur on wires across multiple sub-circuits, the suspect faulty columns are confined within one sub-circuit. In this case, only one reconfiguration is necessary for the entire fault location and repair process. For single fault that occurs on wires across *n* sub-circuits, the worst-case number of configuration attempts is *n*.

To guarantee single-fault tolerance, the modified column-based precompiled configuration scheme needs to reserve $max(k_1, k_2, \ldots, k_m)$ CLB columns in the original configuration. Compared to the scheme using the original overlapping column-based

precompiled configuration approach, this enhanced approach minimizes the number of configuration attempts with the price of extra CLB columns that are reserved as backup.

Because alternative configurations are created by shifting multiple columns, the same coding scheme as the original column-based precompiled configuration approach described in Sec. 5.3 can be used for compressing configuration data and achieving significant storage reduction. For an *m*-sub-circuit system, the total number of alternative configurations required for tolerating single fault is *m*. Compared to the overlapping column-based precompiled configuration scheme where $(k_1+k_2+\ldots+k_m)$ alternative configurations are required for guaranteeing single-fault tolerance, the modified scheme needs fewer configurations and thus a smaller configuration storage overhead.

Note that when a sub-circuit occupies a large number of columns, an alternative configuration that avoids the whole region of such sub-circuit may not be found because of the routing constraints and the total number of columns available in the reconfigurable hardware. Therefore, the modified column-based precompiled configuration approach is more suitable for systems that can be partitioned into small sub-circuits.

6.4 Case Study: LZ Compression Encoder in FPGAs

To estimate the impact on area overhead due to the fine-grained partitioning and distributed checkers in Sec. 6.3, we re-examined the case study used in Chapter 3: LZ compression in FPGAs. The hardware architecture of such application is shown in Fig. 6-6, which is redrawn from Fig. 3-4 in Chapter 3.

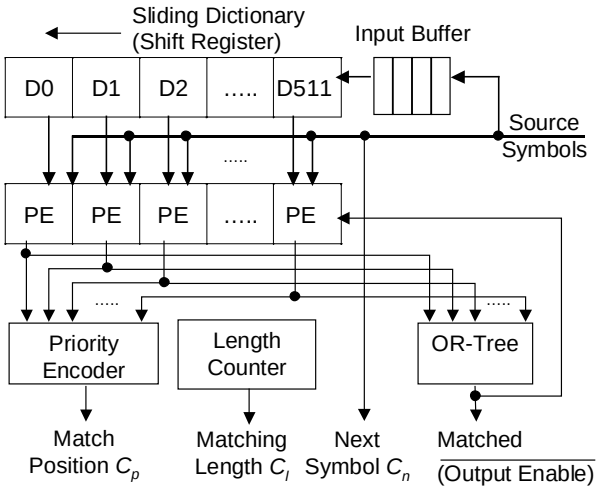

Figure 6-6: Architecture of LZ encoder.

In Chapter 3, the inverse comparison CED scheme is used for detecting errors by checking if the encoded output can be decoded to correctly match the source data. However, because the check is only performed at the output codewords, it is difficult to partition the LZ encoder and to insert distributed checkers in fine granularity if the inverse comparison CED is used. Therefore, in this case study where the entire LZ encoder is the target system, we use the duplex CED scheme for estimating the impact of area overhead due to the fine-grained partitioning and distributed checkers. In other systems where the LZ encoder is a sub-circuit, the inverse comparison CED can be used.

Our designs are mapped in the Xilinx Virtex XCV1000 FPGA [Xilinx 01], which has a CLB array of 64 rows and 96 columns. The area and clock frequency results are reported by Xilinx Alliance 3.1i place-and-route tool. The number of sub-circuits in the system is selected between 9 and 17 in order to balance the size of each sub-circuit. Detailed discussions on the choice of partitioning methods for this case study can be found in Appendix E.

Table 6-1 shows the resulting area and clock frequency for simplex (single module without error detection), duplex without partitioning, duplex with nine sub-circuits, and duplex with 17 sub-circuits. Note that although the area overhead in terms of CLB utilization increases with the number of sub-circuits, the number of columns used does not change significantly in different duplex schemes. This is because the place-and-route software generally results in scattered empty CLBs in order to facilitate routing. Therefore, the area overhead due to distributed checkers and fine-grained partitioning is not very significant compared to a duplex scheme without partitioning. Degradation of maximum clock rate because of partitioning is within 10% in this case study.

Table 6-1: Area overhead and clock rate for different schemes.

Scheme	CLB utilization (slices)	CLB columns used	Max. clock rate (MHz)
1. Simplex	4863	41	33.2
2. Duplex without partition	9743	84	32.7
3. Duplex with 9 sub-circuits	9794	85	31.5
4. Duplex with 17 sub-circuits	9962	85	30.0

Table 6-2 summarizes the partitioning results for duplex schemes with 9 and 17 sub-circuits. Because there are global signals connecting the controller sub-circuit and all the other sub-circuits in the systolic array, the worst-case suspect set spans the entire used region in the FPGA. However, such global signals represent only about 0.2% of all signals in the system in both schemes, and only faults in such signals can produce a suspect faulty set of size greater than two sub-circuits. Therefore, with a very high probability, the entire fault location and recovery process can be completed in only one or two reconfigurations using the modified column-based precompiled configuration scheme.

Table 6-2: Comparison of two partitioning schemes.

Scheme	No. of checkers	Max. no. of columns per sub-ckt	Max. no. of columns in the suspect set	Pct. of signals across more than 2 sub-ckts
Duplex, 9 sub-ckts	9	10	85	0.19%
Duplex, 17 sub-ckts	17	5	85	0.22%

For the duplex scheme with 9 sub-circuits, 10 additional columns are required as backup for guaranteeing 1-column fault tolerance using the modified column-based precompiled configuration scheme. For the duplex scheme with 17 sub-circuits, only 5 additional columns are required to achieve the same fault tolerance capability. Since both schemes are mapped into the same number of columns, the duplex scheme with 17 sub-circuits is preferred if the modified column-based precompiled configuration scheme is used.

6.5 Summary

We present a blind reconfiguration technique, which integrates CED schemes with the column-based precompiled configuration approach used for fault tolerance, for solving run-time fault location and recovery problem in the reconfigurable hardware rapidly. Using distributed CED checkers in each sub-circuit of a system, faulty column suspects can be obtained, and the number of configuration attempts for fault location and recovery can be reduced. With a modified column-based precompiled configuration

scheme that shifts the configuration in units of sub-circuits, the number of configuration attempts can be further reduced.

A case study of LZ compression in FPGAs show that our approach can complete the fault location and recovery process in one or two reconfigurations with a very high probability. For duplex CED schemes, extra area overhead and variations of clock frequencies due to partitioning and distributed checkers are small.

Chapter 7
Concluding Remarks

This dissertation summarizes my contributions to dependable computing techniques for applications using reconfigurable hardware. With the inherent fine-grained redundancy of resources and the re-programmable feature, reconfigurable hardware has great potential for constructing cost-effective systems with high dependability. Various run-time supports required for achieving this objective have been studied thoroughly, including concurrent error detection (CED) techniques, transient error recovery schemes, fault location techniques, and reconfiguration strategies for tolerating permanent faults.

For concurrent error detection, the inverse comparison CED technique guarantees data integrity for applications with unique inverse. Such a technique is very attractive when the inverse operation is simpler than the forward process because both area overhead and time overhead can be very small.

For transient error recovery in reconfigurable hardware, two kinds of techniques should be executed in order to restore normal operations from different types of transient faults. For system transients where configuration data is not changed, rollback recovery or forward recovery techniques can be used to restore normal operations. Two rollback recovery techniques, reload-retry and direct-retry, have been developed for the LZ compression application. For configuration transients where configuration data is changed, configuration readback and writeback operations can be used to correct the error. In order to minimize the performance degradation in user operations, the configuration transient recovery process is executed either as a background scrubbing operation or initiated concurrently with the system transient recovery process.

The concurrent processes in transient error recovery schemes in reconfigurable hardware raises a memory coherence issue when some of the Look-Up Tables (LUTs) in Configurable Logic Blocks (CLBs) are configured as user memory modules. Two on-line memory coherence techniques, stall-when-write strategy and the dirty-bit technique, are developed. Simulation results show that systems with the dirty-bit technique have

lower time overhead in user operations when a small number of faulty configuration frames is present. The dirty-bit technique has a stable, small performance overhead under different environmental parameters, and the area overhead is small.

For tolerating permanent faults, column-based precompiled configuration techniques achieve significant dependability improvement with minimum post-fault-location downtime. Because alternative configurations are constructed by column-based shifting, an integrated differential and run-length coding technique can be used to reduce the storage overhead for precompiled configuration data by 1-2 orders of magnitude.

Based on the integration of the column-based precompiled configuration technique and CED schemes with distributed checkers, a blind reconfiguration approach is developed which reduces the total system downtime for the entire permanent fault location and repair process. Using this approach, a case study shows that the entire fault location and repair process can be completed in one or two reconfigurations with a very high probability.

There are several subjects for future investigation. For column-based pre-compiled configuration schemes, although similarities in alternative configurations cause significant storage reduction after data compression, effects of Common Mode Failures (CMFs) in similar configuration patterns and the resulting degradation in fault-tolerant capability would need to be characterized. Development of proper fault models and fault simulation methodologies for reconfigurable hardware would be also helpful for measuring the effectiveness of various dependable computing techniques.

References

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

[Abramovici 99] Abramovici, M., C. Stroud, C. Hamilton, et. al., "Using Roving STARs for Online Testing and Diagnosis of FPGAs in Fault-Tolerant Applications," *Proc. International Test Conference*, pp. 973-982, 1999.

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

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

[Berger 61] Berger, J. M., "A Note on Error Detection Codes for Asymmetric Channels," *Information and Control*, vol. 4, pp. 68-73, 1961.

[Bernstein 88] Bernstein, P. A., "Sequoia: A Fault-Tolerant Tightly Coupled Multiprocessor for Transaction Processing," *IEEE Computer*, vol. 21, no. 2, pp. 37-45, 1988.

[Boley 95] Boley, D., G. H. Golub, S. Makar, N. Saxena, and E. J. McCluskey, "Floating Point Fault Tolerance with Backward Error Assertions", *IEEE Trans. on Computers*, vol. 44, no. 2, pp. 302-311, 1995.

[Bose 85] Bose, B. and D. J. Lin, "Systematic Unidirectional Error-Detecting Codes," *IEEE Trans. on Computers*, pp. 1026-1032, Nov. 1985.

[Bowen 92] Bowen, N., and D. K. Pradhan, "Virtual Checkpoints: Architecture and Performance," *IEEE Trans. on Computers*, vol. 41, pp. 516-525, May 1992.

[Calgary 90] Calgary Text Compression Corpus, available from Univ. of Calgary, Canada. *ftp://ftp.cpsc.ucalgary.ca*.

[Carmichael 99] Carmichael, C., E. Fuller, P. Blain, and M. Caffrey, "SEU Mitigation Techniques for Virtex FPGAs in Space Applications," *2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies International Conference (MAPLD '99)*, Sept. 1999.

[Chang 98] Chang, D., and M. Marek-Sadowska, "Partitioning Sequential Circuits on Dynamically Reconfigurable FPGAs," *Proc. ACM International Symp. on Field-Programmable Gate Arrays*, pp. 161-167, 1998.

[Cisco 01] Cisco Systems Inc., *http://www.cisco.com*, 2001.

[Culbertson 97] Culbertson, W. B., R. Amerson, R. J. Carter, P. Kuekes, et. al., "Defect Tolerance on the Teramac Custom Computer," *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, pp. 116-123, 1997.

[Dai 99] Dai, C., et. al., "Alpha-SER Modeling & Simulation for Sub-0.25μm CMOS Technology," *Proc. Symp. VLSI Tech.*, pp. 81-82, 1999.

[Das 99] Das, D., and N. A. Touba, "A Low Cost Approach for Detecting, Locating, and Avoiding Interconnect Faults in FPGA-Based Reconfigurable Systems," *Proc. IEEE International Conference on VLSI Design*, pp. 266-269, 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 International Conference on Computer-Aided Design*, pp. 173-176, 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, 1997.

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

[Emmert 00] Emmert, J., C. Stroud, B. Skaggs, and M. Abramovici, "Dynamic Fault Tolerance in FPGAs via Partial Reconfiguration," *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 165-174, 2000.

[Golomb 66] Golomb, S. W., "Run-length Encoding," *IEEE Trans. on Information Theory*, vol. IT-12, pp. 399-401, 1966.

[Hanchek 98] Hanchek, F., and S. Dutt, "Methods for Tolerating Cell and Interconnect Faults in FPGAs," *IEEE Trans. on Computers*, Vol. 47, No. 1, pp. 15-32, 1998.

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

[Huang 00a] Huang, W.-J., N. Saxena, and E. J. McCluskey, "A Reliable LZ Data Compressor on Reconfigurable Coprocessors," *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 249-258, 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, 2000.

[Huang 01a] Huang, W.-J., and E. J. McCluskey, "A Memory Coherence Technique for Online Transient Error Recovery of FPGA Configurations," *Proc. Ninth ACM International Symposium on Field-Programmable Gate Arrays*, pp. 183-192, 2001.

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

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

[IBM 01] International Business Machines, Inc., *http://www.ibm.com*, 2001.

[Jewett 91] Jewett, D., "A Fault-Tolerant Unix Platform," *Proc. International Symposium on Fault-Tolerant Computing*, pp. 512-519, 1991.

[Johnson 89] Johnson, B. W., *Design and Analysis of Fault Tolerant Digital Systems*, Addison-Wesley, 1989.

[Jordan 93] Jordan, C., and W.P. Marnane, "Incoming Inspection of FPGAs," *European Test Conference*, pp. 371-377, 1993.

[Jou 88] Jou, J-Y, and J. A. Abraham, "Fault-Tolerant FFT Networks," *IEEE Trans. Computers*, vol. 37, no. 5, pp. 548-561, May 1988.

[Jung 98] Jung, B., and W. P. Burleson, "Efficient VLSI for Lempel-Ziv Compression in Wireless Data Communication Networks," *IEEE Trans. on VLSI Systems*, vol. 6, no. 3, pp.475-483, 1998.

[Kelem 00] Kelem, S., "XAPP151: Virtex Configuration Architecture Advanced Users' Guide," Xilinx Application Note, *http://www.xilinx.com*, 2000.

[Kraft 81] Kraft, G. D. and W. N. Toy, *Microprogrammed Control and Reliable Design of Small Computers*, 1981.

[Lach 98] Lach, J., W. H. Mangione-Smith, and M. Potkonjak, "Efficiently Supporting Fault-Tolerance in FPGAs", *Proc. ACM International Symp. on Field-Programmable Gate Arrays*, pp. 105-115, 1998.

[Lach 99] Lach, J., W. H. Mangione-Smith, and M. Potkonjak, "Algorithms for Efficient Runtime Faulty Recovery on Diverse FPGA Architectures", *Proc. IEEE International Symp. on Defect and Fault Tolerance in VLSI Systems*, pp. 386-394, 1999.

[Lakamraju 00] Lakamraju, V., and R. Tessier, "Tolerating Operational Faults in Cluster-Based FPGAs," *Proc. ACM Int'l Symp. on Field Programmable Gate Arrays*, pp. 187-194, 2000.

[Lin 90] Lin, T.-T. Y., and D. P. Siewiorek, "Error Log Analysis: Statistical Modeling and Heuristic Trend Analysis," *IEEE Trans. Reliability*, vol. 39, no. 4, pp. 419-432, 1990.

[Lombardi 96] Lombardi, F., D. Ashen, X. Chen, and W.K. Huang, "Diagnosing Programmable Interconnect Systems for FPGAs," *Proc. ACM International Symp. on Field Programmable Gate Arays*, pp. 100-106, 1996.

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

[Lucent 01] Lucent Technologies, Inc., *http://www.lucent.com*, 2001.

[Mahapatra 99] Mahapatra, N. R., and S. Dutt, "Efficient Network-Flow Based Techniques for Dynamic Fault Reconfiguration in FPGAs", *Proc. International Symposium on Fault-Tolerant Computing*, pp. 122-129, 1999.

[Mange 98] Mange, D., E. Sanchez, A. Stauffer, G. Tempesti, P. Marchal, and C. Piguet, "Embryonics: A New Methodology for Designing Field-Programmable Gate Arrays with Self-repair and Self-replicating Properties," *IEEE Trans. on VLSI Systems*, vol. 6, no. 3, pp. 387-399, 1998.

[May 79a] May, T. C., and M. H. Woods, "Alpha-Particle-Induced Soft Errors in Dynamic Memories," *IEEE Trans. Electron Devices*, vol. ED-26, no. 1, Jan. 1979.

[May 79b] May, T. C., "Soft Errors in VLSI – Present and Future," *IEEE Trans. Comp. Hyb., Manuf. Technol.*, vol. CHMT-2, pp. 377-387, Dec. 1979.

[McCluskey 81] McCluskey, E.J., and S. Bozorgui-Nesbat, "Design for Autonomous Test," *IEEE Trans. Computers*, pp. 866-875, Nov. 1981.

[McCluskey 82] McCluskey, E.J., "Verification Testing," *Proc. Design Automation Conference*, pp. 495-500, 1982.

[McCluskey 90] McCluskey, E. J., "Design techniques for Testable Embedded Error Checkers," *IEEE Computer*, vol. 23, no. 7, pp. 84-88, July 1990.

[Mitra 98] Mitra, S., P. P. Shirvani, and E.J. McCluskey, "Fault Location in FPGA-Based Reconfigurable Systems," *IEEE Intl. High Level Design Validation and Test Workshop*, 1998.

[Mitra 00a] Mitra, S. and E. J. McCluskey, "Which Concurrent Error Detection Scheme to Choose?," *Proc. International Test Conference*, pp. 985-994, 2000.

[Mitra 00b] 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 Symp. on Dependable Computing, Fast Abstracts*, pp. 15-16, 2000.

[Nagaraj 99] Nagaraj, N.S., et. al., "Performance and Reliability Verification of C6201/C6701 Digital Signal Processors," *Proc. IEEE Int'l Conf. Computer Design*, pp. 521-525, 1999.

[Nelson 96] Nelson, M., and J. L. Gailly, *The Data Compression Book*, 2nd edition, M&T Books, 1996.

[Nicolaidis 93] Nicolaidis, M., "Efficient Implementations of Self-Checking Adders and ALUs," *Proc. Intl. Symp. Fault-Tolerant Computing*, pp. 586-595, 1993.

[Nicolaidis 97] Nicolaidis, M., R. O. Duarte, S. Manich and J. Figueras, "Fault-secure Parity Prediction Arithmetic Operators," *IEEE Design and Test of Computers*, vol. 14, no. 2, pp. 60-71, 1997.

[Nortel 01] Nortel Networks Inc., *http://www.nortelnetworks.com*, 2001.

[Oh 01] Oh, N., P. P. Shirvani, and E. J. McCluskey, "Error Detection by Duplicated Instructions in Super-scalar Processors," to appear in *IEEE Transactions on Reliability*, Sep. 2001.

[Patel 82] Patel, J.H. et al. "Concurrent Error Detection in ALUs by REcomputing with Shifted Operands," *IEEE Transactions on Computers*, vol.C-31, no.7, pp. 589-595, July 1982.

[Pradhan 94] Pradhan, D. K., and N. H. Vaidya, "Roll Forward Checkpointing Scheme: A Novel Fault-Tolerant Architecture," *IEEE Trans. on Computers*, vol. 43, no. 10, pp. 1163-1174, 1994.

[Pradhan 96] Pradhan, D. K., *Fault-Tolerant Computer System Design*, Prentice Hall, 1996.

[Renovell 98] Renovell, M., J.M. Portal, J. Figueras, and Y. Zorian, "Testing the Interconnect of RAM-Based FPGAs," *IEEE Design and Test of Computers*, pp. 45-50, Jan. 1998.

[Reynolds 78] Reynolds, D. and G. Metze, "Fault Detection Capabilities of Alternating Logic," *IEEE Trans. on Computers*, vol. C-27, pp.1093-1098, Dec. 1978.

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

[Saxena 00] Saxena, N. R., S. Fernandez-Gomez, W.-J. Huang, S. Mitra, S.-Y Yu, and E. J. McCluskey, "Dependable Computing and On-Line Testing in Adaptive and Configurable Systems," *IEEE Design and Test of Computers*, vol.17, no. 1, pp. 29-41, 2000.

[Sedmak 78] Sedmak, R. M. and H. L. Liebergot, "Fault-Tolerance of a General-Purpose Computer Implemented by Very Large Scale Integration," *Proc. International Symposium on Fault-Tolerant Computing*, pp. 137-143, 1978.

[Seifert 01] Seifert, N., et. al., "Historical Trend in Alpha-Particle Induced Soft Error Rates of the Alpha Microprocessor," *IEEE Int'l Reliability Physics Symp.*, 2001, to appear.

[Sellers 68] Sellers, F., M-Y Hsiao and L. W. Bearnson, *Error Detection Logic for Digital Computers*, McGraw-Hill Book Company, 1968.

[Serlin 84] Serlin, O., "Fault-Tolerant Systems in Commercial Applications," *IEEE Computer*, pp. 19-30, Aug. 1984.

[Shedletsky 78] Shedletsky, J. J. et al., "Error correction by alternate-data retry," *IEEE Trans. on Computers*, Vol. C-27, no.2, pp. 106-112, Feb.1978.

[Shirvani 00] Shirvani, P.P., N. Oh, E.J. McCluskey, D. Wood and K.S. Wood, "Software-Implemented Hardware Fault Tolerance Experiments; COTS in Space," *Proc. International Conference on Dependable Systems and Networks (FTCS-30 and DCCA-8)*, Fast Abstracts, pp. B56-7, 2000.

[Siewiorek 92] Siewiorek, D. P., and R. S. Swarz, *Reliable Computer Systems: Design and Evaluation*, 2nd Edition, Digital Press, 1992.

[Spainhower 99] 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. 1999.

[Strom 88] Strom, R. E., D. F. Bacon, and S. A. Yemini, "Volatile Logging in *n*-Fault-Tolerant Distributed Systems," *Proc. Int'l Symposium on Fault-Tolerant Computing*, pp. 42-50, 1988.

[Stroud 97] Stroud, C., E. Lee, and M. Abramovici, "BIST-Based Diagnostics of FPGA Logic Blocks," *Proc. International Test Conference*, pp. 539-547, 1997.

[Stroud 98] Stroud, C., S. Wijesuriya, C. Hamilton, and M. Abramovici, "Built-In Self-Test of FPGA Interconnect," *Proc. International Test Conference*, pp. 404-411, 1998.

[Takeda 80] Takeda, K. et al., "Logic Design of Fault-Tolerant Arithmetic Units Based on the Data Complementation Strategy," *10th International Symp. on Fault-Tolerant Computing*, pp. 348-350, 1980.

[Touba 97] Touba, N. A. and E. J. McCluskey, "Logic Synthesis of Multilevel Circuits with Concurrent Error Detection," *IEEE Trans. Computer-Aided Design*, vol. 16, pp. 783-789, July, 1997.

[Trimberger 98] Trimberger, S., "Scheduling Designs into a Time-Multiplexed FPGA," *Proc. ACM International Symp. on Field Programmable Gate Arrays*, pp. 153-160, 1998.

[Wakerly 78] Wakerly, J., *Error Detecting Codes, Self-checking Circuits and Applications*, 1978.

[Woods 86] Woods, M. H., "MOS VLSI Reliability and Yield Trends," *Proceedings IEEE*, vol. 74, no. 12, pp. 1715-1729, Dec. 1986.

[Wu 90] Wu, K.-L., W. K. Fuchs, and J. H. Patel, "Error Recovery in Shared Memory Multiprocessors Using Private Caches," *IEEE Trans. on Parallel and Distributed Systems*, vol. 1, no. 2, pp.231-240, 1990.

[Xilinx 01] Xilinx Inc., *http://www.xilinx.com*, 2000.

[Zeng 99] Zeng, C., N. R. Saxena and E. J. McCluskey, "Finite State Machine Synthesis with Concurrent Error Detection," *Proc. International Test Conference*, pp. 672-680, 1999.

[Ziegler 96] Ziegler, J. F., et. al., *IBM Journal of Research Development*, vol. 40, no. 1, (all articles), Jan. 1996.

[Ziv 77] Ziv, J. and A. Lempel, "A Universal Algorithm for Sequential data Compression," *IEEE Trans. Information Theory*, vol. IT-23, no. 3, pp. 337-343, 1977.

[Ziv 78] Ziv, J. and A. Lempel, "Compression of Individual Sequence via Variable-Rate Coding," *IEEE Trans. Information Theory*, vol. IT-24, no. 5, pp. 530-536, 1978.