

AFRL-IF-WP-TR-2002-1505

**ON-LINE TESTING AND
RECONFIGURATION OF FIELD
PROGRAMMABLE GATE ARRAYS
(FPGAs) FOR FAULT-TOLERANT (FT)
APPLICATIONS IN ADAPTIVE
COMPUTING SYSTEMS (ACS)**



Miron Abramovici

**Lucent Technologies, Inc.
Circuits & Systems Research Laboratory
Agere Systems
600 Mountain Avenue
Murray Hill, NJ 07974**

John M. Emmert and Charles E. Stroud

**Department of Electrical & Computer Engineering
University of North Carolina at Charlotte
9201 University City Blvd.
Charlotte, NC 28223**

APRIL 2002

Final Report for 01 May 1998 – 30 April 2002

Approved for public release; distribution is unlimited.

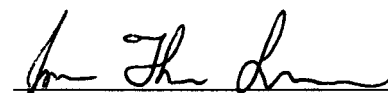
**INFORMATION DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7334**

NOTICE


USING GOVERNMENT DRAWINGS, SPECIFICATIONS, OR OTHER DATA INCLUDED IN THIS DOCUMENT FOR ANY PURPOSE OTHER THAN GOVERNMENT PROCUREMENT DOES NOT IN ANY WAY OBLIGATE THE US GOVERNMENT. THE FACT THAT THE GOVERNMENT FORMULATED OR SUPPLIED THE DRAWINGS, SPECIFICATIONS, OR OTHER DATA DOES NOT LICENSE THE HOLDER OR ANY OTHER PERSON OR CORPORATION; OR CONVEY ANY RIGHTS OR PERMISSION TO MANUFACTURE, USE, OR SELL ANY PATENTED INVENTION THAT MAY RELATE TO THEM.

THIS REPORT HAS BEEN REVIEWED BY THE OFFICE OF PUBLIC AFFAIRS (ASC/PA) AND IS RELEASABLE TO THE NATIONAL TECHNICAL INFORMATION SERVICE (NTIS). AT NTIS, IT WILL BE AVAILABLE TO THE GENERAL PUBLIC, INCLUDING FOREIGN NATIONS.

THIS TECHNICAL REPORT HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION.



LT JASON LAWSON
Project Engineer



KENNETH LITTLEJOHN, Team Leader
Embedded Info Sys Engineering Branch
Information Technology Division
Information Directorate



JAMES S. WILLIAMSON, Chief
Embedded Info Sys Engineering Branch
Information Technology Division
Information Directorate

Do not return copies of this report unless contractual obligations or notice on a specific document require its return.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**

1. REPORT DATE (DD-MM-YY) April 2002		2. REPORT TYPE Final		3. DATES COVERED (From - To) 05/01/1998 – 04/30/2002	
4. TITLE AND SUBTITLE ON-LINE TESTING AND RECONFIGURATION OF FIELD PROGRAMMABLE GATE ARRAYS (FPGAs) FOR FAULT-TOLERANT (FT) APPLICATIONS IN ADAPTIVE COMPUTING SYSTEMS (ACS)				5a. CONTRACT NUMBER F33615-98-C-1318	
				5b. GRANT NUMBER	
				5c. PROGRAM ELEMENT NUMBER 69199F	
6. AUTHOR(S) Miron Abramovici (Lucent Technologies, Inc.) John M. Emmert (University of North Carolina at Charlotte) Charles E. Stroud (University of North Carolina at Charlotte)				5d. PROJECT NUMBER ARPI	
				5e. TASK NUMBER FT	
				5f. WORK UNIT NUMBER 03	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Lucent Technologies, Inc. Circuits & Systems Research Laboratory Agere Systems 600 Mountain Avenue Murray Hill, NJ 07974				8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Information Directorate Air Force Research Laboratory Air Force Materiel Command Wright-Patterson Air Force Base, OH 45433-7334				10. SPONSORING/MONITORING AGENCY ACRONYM(S) AFRL/IFTA	
				11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S) AFRL-IF-WP-TR-2002-1505	
12. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.					
13. SUPPLEMENTARY NOTES Report contains color.					
14. ABSTRACT (Maximum 200 Words) Adaptive computing systems (ACS) rely on reconfigurable hardware to adapt the system operation to changes in the external environment, and to extend mission capability by implementing new functions on the same hardware platform. This results in increased functional density and reduced power consumption – features very important in many domains, such as space missions or mobile devices. Field-programmable gate arrays (FPGAs) featuring incremental dynamic run-time reconfiguration (RTR) offer additional benefits by allowing the system to continue to execute uninterrupted, while portions of the FPGA are reconfigured for new logic functions. ACS are often deployed in harsh and/or hostile remote environments, and they are subject to strict high-reliability and high-availability requirements. Cosmic radiation may perturb the operation of a defect-free FPGA, and marginal defects not causing failures in manufacturing testing (such as a short initially having very high resistance) may become active with the aging of the device, or because of environmental factors. Since direct human intervention for maintenance and repair is impossible in such environments, fault-tolerant (FT) techniques resulting in graceful degradation must be used to achieve the desired mission life even in the presence of faults. However, traditional FT design, based on replicated modular redundancy and voting, is extremely expensive given the space, weight, and power constraints of ACS.					
15. SUBJECT TERMS Formal Methods, FPGAs, Fault Tolerance, Adaptive Computing Systems, Reconfiguration, Run-Time, Testing, Diagnosis					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT: SAR	18. NUMBER OF PAGES 84	19a. NAME OF RESPONSIBLE PERSON (Monitor) 1st Lt. Jason Lawson 19b. TELEPHONE NUMBER (Include Area Code) (937) 255-6548 x3586
a. REPORT Unclassified	b. ABSTRACT Unclassified	c. THIS PAGE Unclassified			

Table of Contents

<u>Section</u>	<u>Page</u>
Table of Contents	iii
List of Figures	iv
List of Tables	vi
1. Introduction	1
2. Main Contributions	5
3. The Roving STARS	7
3.1. The Test and Reconfiguration Controller	7
3.2. The Roving STARS Structure	7
3.3. The Roving Mechanism	8
4. On-Line Testing and Diagnosis	11
4.1. Basic BIST Structure	11
4.2. Testing Programmable Logic Blocks.....	12
4.3. Divide-and-Conquer Adaptive Diagnosis.....	17
4.4. Locating Faulty PLBs Using BISTER Results	19
4.5. Diagnosis within a Faulty PLB	26
4.6. Testing Interconnect.....	29
4.7. Interconnect Diagnosis.....	39
4.8. Fault Injection Emulation	41
5. Fault-Tolerant Techniques	45
5.1. Basic Features	45
5.2. Related Research.....	46
5.3. Multilevel Fault Tolerance Approach.....	49
5.4. Fault Tolerance for PLBs.....	50
5.5. Fault Tolerance for Interconnect.....	55
5.6. Fault Tolerance Summary	61
6. Summary and Conclusions	63
7. Participants, Patents, and Publications	64
8. References	69
List of Acronyms	73

List of Figures

<u>Figure</u>	<u>Page</u>
Figure 1. Roving Area Under Test Across the Chip	1
Figure 2. Horizontal and Vertical STARS	7
Figure 3. Roving Steps	9
Figure 4. Basic BISTER Tile	11
Figure 5. Integrated ORA/Scan cell	12
Figure 6. Typical PLB Structure	12
Figure 7. Configuration Multiplexer	13
Figure 8. 3 by 2 BISTER Tile Rotations	14
Figure 9. 4 by 2 BISTER Tile Rotations	14
Figure 10. BISTER Tile Rotations (Two Faulty BUTs)	15
Figure 11. BISTER Tile Rotations (One Faulty BUT)	15
Figure 12. Layout of a BISTER Tile	17
Figure 13. Example of Adaptive PLB Diagnosis	18
Figure 14. Example of Inconclusive Diagnosis	18
Figure 15. Combined test sessions a) 3 by 2 tile b) 4 by 2 tile	19
Figure 16. Figure for Equation (1)	20
Figure 17. Figure for Theorem 2	22
Figure 18. Figure for Lemma 6	22
Figure 19. Figure for Lemma 7	23
Figure 20. Figure for Example 1	24
Figure 21. Figure for Example 3	25
Figure 22. Figure for Example 4	25
Figure 23. Diagnosis of Actual Faulty PLB	26
Figure 24. ORA with Greater Diagnostic Resolution	27
Figure 25. PUB Diagnosis.	28
Figure 26. CIP Structure and Types	29
Figure 27. Single PLB View of Routing Resources	30
Figure 28. Basic BIST Structure for Interconnect	32
Figure 29. Bus Realignment	32
Figure 30. Integrated ORA/scan cell	33
Figure 31. Testing global crosspoint CIPs	33
Figure 32. Roving Order for Testing Interconnect	34
Figure 33. BISTER for Vertical Busses	35
Figure 34. Example of Test Phases for Local Routing	36
Figure 35. Testing Crosspoint CIPs between Direct and Global Busses	36
Figure 36. Example of MUX CIP Test Phases	37
Figure 37. Testing global crosspoint CIPs	37
Figure 38. Testing Results from Faulty ORCA 2C15s	39
Figure 39. Diagnostic Configurations	39
Figure 40. Wire Segments and Faults	40
Figure 41. TREC and fault injection emulations GUI	44
Figure 42. Example of Four Logic Cells Time Sharing One PLB.....	50

Figure 43. Number of Faults Tolerated with and without PUBs	52
Figure 44. Spare Cell Allocation Patterns	54
Figure 45. Illustration for Fault Compatibility	55
Figure 46. Multiple Fault Interaction	56
Figure 47. Compatible Multiple Fault	56
Figure 48. Where Is the Short?	57
Figure 49. Signal Compatibility	58
Figure 50. Taking Advantage of an Open Segment	58
Figure 51. Example Stuck-Open CIP	59
Figure 52. Bypassing an Open Segment	60

List of Tables

<u>Table</u>	<u>Page</u>
Table 1.Performance Penalty with STARs	10
Table 2:Summary of BIST Phases for ORCA 2C and 2CA Logic Blocks.....	16
Table 3:Summary of Additional Diagnostic Phases for ORCA 2C and 2CA.....	27
Table 4:Summary of interconnect BIST test sessions	34
Table 5:Summary of interconnect BIST test sessions and phases.....	38
Table 6:Summary of Interconnect BIST Test Results on Faulty Chips	38
Table 7.Performance Penalty with Spare PLBs	54
Table 8.Single-Fault Compatibility Conditions	56

1. Introduction

Adaptive computing systems (ACS) rely on reconfigurable hardware to adapt the system operation to changes in the external environment, and to extend mission capability by implementing new functions on the same hardware platform. This results in increased functional density and reduced power consumption – features very important in many domains, such as space missions or mobile devices. Field-programmable gate arrays (FPGAs) featuring incremental dynamic runtime reconfiguration (RTR) offer additional benefits by allowing the system to continue to execute uninterrupted, while portions of the FPGA are reconfigured for new logic functions. ACS are often deployed in harsh and/or hostile remote environments, and they are subject to strict high-reliability and high-availability requirements. Cosmic radiation may perturb the operation of a defect-free FPGA, and marginal defects not causing failures in manufacturing testing (such as a short initially having very high resistance) may become active with the aging of the device, or because of environmental factors. Since direct human intervention for maintenance and repair is impossible in such environments, fault-tolerant (FT) techniques resulting in graceful degradation must be used to achieve the desired mission life even in the presence of faults. However, traditional FT design, based on replicated modular redundancy and voting, is extremely expensive given the space, weight, and power constraints of ACS.

While FPGA testing has been the focus of many research and development efforts [2, 18, 20, 21, 25, 26, 27, 28, 33, 34, 41, 44, 45, 46, 52, 53, 55, 56, 57, 60], most of this previous work deals with off-line testing, which is not applicable in missions relying on continuous system operation. The only attempt at on-line FPGA testing has been the pioneering work of Shnidman et al. [48]. The main idea, originally proposed by Shombert and Siewiorek in the context of testing systolic arrays [49], is to have only a relatively small portion of the chip off-line and being tested, while the rest of the device is on-line and continues its normal operation. Testing of the entire FPGA is accomplished by repeatedly moving different sections of the system logic in the most recently tested part, thus allowing a new area of the chip to be tested. Figure 1 illustrates this process, called roving spares in [49] and fault scanning in [48]. For FPGAs, roving relies on RTR, which is the ability to dynamically reconfigure part of an FPGA without disturbing the operation performed in the rest of the device. Compared with on-line testing based on modular redundancy, this approach has much smaller overhead. Since a fault occurring in the working area will be detected the next time when the fault site will be under test, the fault latency (the interval between the occurrence of a fault and its detection) is bounded by the interval required to test the entire FPGA.

Despite these advantages, the method proposed in [48] has several serious limitations. Although today most FPGAs use dedicated wire segments for inter-block communication, this method can work only for bus-based FPGAs. Even with this limitation, it is not applicable to any existing



Figure 1. Roving Area Under Test Across the Chip

FPGA architecture, since it requires a modified design of the FPGA blocks. Moreover, these modifications introduce a substantial overhead, since the entire configuration memory is duplicated. This method does not address faults in the programmable interconnect network, which occupies most of the area of an FPGA, and it is also limited by the simplifying assumption that the configuration memory and flip-flops (FFs) are fault-free.

Most FT approaches for FPGAs target yield improvement by replacing faults detected in manufacturing test with spare resources to achieve a fully functional chip [7, 9, 10, 11, 12, 19, 22, 23, 24, 29, 31, 36, 40, 42, 47, 58]. But most of these methods are not applicable for on-line fault tolerance, because they can deal with only a limited number of faults, or only with specific faults, or use repair techniques not available in remote environments, or require excessive redundancy, or need excessive computation time to determine how to bypass faults. The method of Lach et al. [31] divides the FPGA into tiles, reserves a spare cell in every tile, and precomputes replacement configurations for every possible faulty cell in a tile. This removes the need for expensive on-line computations and enables fast bypassing of detected faults. However, this method cannot deal with more than one fault per tile. In [32], Lach et al. extend their method to consider interconnect faults. Similar to their method for handling logic faults, several replacement configurations are precomputed for programmable signals that do not extend beyond the tile, so that each configuration leaves some routing resources unused. Then when a fault occurs, they try to find a configuration that does not use the faulty resource; this method, however, is not able to tolerate all possible faults or a group of faults in the same tile. For signals that do extend between tiles, they propose two alternatives. In some cases the tile size can be enlarged to encompass the signals. This works well for hierarchical architectures like complex programmable logic devices (CPLDs) and for dedicated, linked interconnect paths like fast carries. For signals that are routed through interconnects that extend across several tiles, they leave a long segment vacant, to be used to replace any parallel segment in the same area. But after one faulty segment is replaced by the long spare, no additional faults can be tolerated in that region.

The embryonics approach, developed by Mange et al. [37, 38], applies evolutionary concepts from biology to achieve self-repair in a new type of fine-grained reconfigurable array. While their results are fascinating, this approach is not applicable to existing FPGAs.

All the FT methods discussed above are static, as the spare resources they provide – logic cells and/or interconnect – are preassigned in the FPGA. Too many spares cause a large overhead, while too few spares may result in not having sufficient resources available in areas affected by fault clusters. In contrast, the dynamic method proposed by Mahapatra and Dutt [36] allocates interconnect resources to bypass faults only after faults have been located. If the new required segments conflict with the current usage of the routing tracks, the layout is incrementally modified to make room for the new segments. However, this method is dynamic only with respect to the interconnect resources; spare cells are still statically allocated, which may result in long chains of layout changes between the fault sites and spare cells.

Our approach is specifically targeted toward faults that appear during the lifetime of the chip and affect the logic in a STAR when that area is under test. However, transient faults may affect the working area at any time. For transient faults, our system will use a low-overhead method such as concurrent error detection (CED) [62]. CED and roving STARs are complementary techniques, as CED does not address dormant faults and does not provide for diagnosis. At the system level, we

use a checkpointing and rollback strategy. A rollback restores the state of the system to the last state saved before the fault occurred (checkpoint), and it should be initiated whenever CED detects a fault. The STARs can continue roving during the rollback period, which is viewed as normal system activity. (Faster fault location can be achieved if the CED mechanism can indicate the unit likely to be faulty and roving starts within that unit.) However, when a fault is detected by a STAR, rollback is not needed because either it is already in progress (if CED has already encountered the same fault), or the fault has not affected the recent system operation and hence it must be a dormant one. If CED has detected a fault, but the next STAR roving round does not confirm its existence, the fault must have been a transient one from which the system has recovered by rollback.

THIS PAGE WAS INTENTIONALLY LEFT BLANK

2. Main Contributions

We have developed the first integrated approach to on-line testing, diagnosis, and fault tolerance for FPGAs, where all three subjects are concurrently treated in a unified and consistent manner. Our approach inherits the basic feature of [49] and [48]; namely, we have self-testing areas (STARs) of the FPGA that are roving around the chip. However, our roving STARs approach provides many significant features not available in prior work. Our approach is applicable to any FPGA that supports RTR, and it does not require any modifications to the FPGA architecture.

To satisfy the high-reliability and high-availability requirements of ACS hardware, our approach guarantees complete testing of both logic and interconnect, and does not require any part of the chip to be fault free. Complete testing means that all the FPGA resources, including logic and interconnect currently not employed by the application logic, will be tested in every possible mode of operation. This strategy could be considered an overkill: why don't we test only the used resources, and only in their current mode of operation? The main reason is that during the normal operation of an adaptive system, the FPGA will be used with different configurations at different times, and our strategy makes certain that no segment of the FPGA will be assigned a function that may be incorrectly performed because of its dormant (not yet detected) faults. Also, the resources in the currently unused portions of the FPGA are spares to be used for fault tolerance. But since failures are as likely to affect the spares as the working area, there is no reason to assume that the spares will be fault free; thus the spares may also develop dormant faults, unless they are tested in the same way as the working area. Recently, it has been shown that on-line testing of dormant faults is essential in achieving high reliability in safety-critical applications [52].

Our approach is a type of built-in self-test (BIST), where both the test-pattern generation and the analysis of the output responses are done within the circuit under test. This is much more difficult than external testing, since we cannot assume that the FPGA resources used in test-pattern generators (TPGs) or in output response analyzers (ORAs) are fault free. Nevertheless, our logic BIST techniques will detect any single or multiple fault in a cell, and any possible combination of multiple faulty cells; our interconnect BIST techniques will detect any single or multiple fault affecting the interconnect network. By exhaustively testing both the programmable logic blocks (PLBs) and the programmable interconnect, our tests implicitly detect all possible stuck faults affecting the configuration memory.

We have developed diagnostic techniques that achieve the highest possible resolution, locating any group of faulty PLBs, any group of interconnect faults, and also identifying defective modules inside the faulty cells. (Note that one cannot distinguish between a configuration bit being stuck and the corresponding fault in the resource controlled by the stuck bit.) The diagnostic goal of conventional FT techniques for FPGAs is to identify faulty resources, which are then bypassed and replaced by spares. The high accuracy of our diagnosis allows us to introduce a new form of fault tolerance, based on reusing faulty resources whenever possible. A partially usable block (PUB) is a faulty PLB with identified failing mode(s) of operation. A PUB may be used as a fault-free cell in the working part of the system, provided that its faults do not affect its intended operation. Similarly, we allow the system logic to safely reuse faulty wire segments. Reusing defective hardware resources increases the effective spare capacity and leads to more graceful degradation and longer mission life.

In addition, our roving STARS approach has important advantages in FT applications. In most previous FT work, faults are detected in the working logic, and they must be located and bypassed very quickly to restart the normal operation as soon as possible. In contrast, in our approach the detected faults do not affect the working logic, so the system operation does not have to be interrupted for fault diagnosis and for computing fault-bypassing reconfigurations.

We also introduce the idea of an adaptive system clock. One side-effect of FT reconfigurations is a possible increase of the delays of some critical paths of the circuit. The conventional solution to this problem is to set the clock period with sufficient slack to work for any possible FT reconfiguration. But this cost in performance degradation is paid all the time, even if the worst-case faults will never occur. We use a programmable clock whose initial frequency is set to the maximum value allowed in the fault-free circuit, and we adjust it only when required as a result of FT reconfigurations [14]. In this way the ACS performance will be gracefully degraded, and only when needed in response to the occurrence of new faults.

We have successfully implemented this approach using a commercially available FPGA – specifically, the ORCA 2C and 2CA series FPGA [16]. However, we emphasize that our approach is general, as it may be applied to any FPGA that uses segmented interconnect and features RTR. The remainder of this report is organized as follows. Section 3 presents the structure and the operation of the roving STARS. Section 4 describes the BIST and BIST-based diagnosis of logic and interconnect resources in the FPGA. Section 5 outlines the main concepts of our new FT techniques for faults in logic and in interconnect. Section 6 concludes the report. Section 7 lists the participants in the project, the patents obtained or applied for, and publications. Section 8 contains references.

3. The Roving STARs

3.1 The Test and Reconfiguration Controller

Since ACS normal system operation involves reconfiguring FPGAs for different functions, the reconfiguration process is controlled by a module external to FPGAs; typically, this is an embedded microprocessor having some memory to store configurations. We extend the tasks of this processor to also control the test, diagnosis, and FT functions, including their associated reconfigurations. We refer to this processor as the test and reconfiguration controller (TREC). TREC accesses an FPGA using its boundary-scan mechanism [51], so that this access is transparent to the normal function of the chip. TREC also employs the boundary-scan interface to partially reconfigure the FPGA, to control the BIST, and to retrieve the test results. TREC uses RTR to rove the STARs across the chip and to reconfigure them for different test and diagnosis operations. TREC also controls the system clock(s) and has the capability to stop the system operation for short intervals to allow for safe relocation of the system logic in the last tested area.

If faults are detected during BIST, TREC analyzes the results to locate the faults. If the faults could not be precisely diagnosed, TREC will run additional diagnostic configurations until the desired resolution is achieved. TREC also keeps track of the status of FPGA resources, and determines if the defective hardware located in the current STAR affects the operation of the next slice of the working area to be relocated over the STAR position. If the faults affect only spare resources, or if the faulty resources may be safely reused to implement the required function, TREC proceeds with the relocation. If the faulty resources may not be reused, then TREC determines configuration changes to the working area so that the defective resources are bypassed and replaced by fault-free ones. For faults affecting its own operation, TREC employs microprocessor-specific FT techniques that will not be discussed in this report.

3.2 The Roving STARs Structure

Figure 2 illustrates the floor plan of an FPGA at some stage of the roving process; the FPGA has a working area and a spare area consisting of a vertical STAR (V-STAR) with two columns and an horizontal STAR (H-STAR) with two rows. Depending on the positions of the two STARs, the working area may be contiguous or split in two of four regions. Most of the time only one STAR is active (testing) at any time. A STAR tests both the PLBs and the interconnect in its area. A PLB may use local interconnect to communicate with its adjacent cells, and global interconnect for

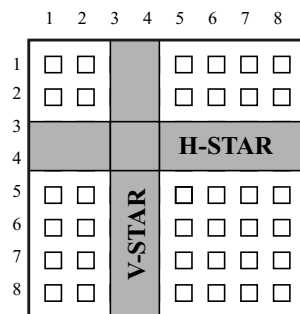


Figure 2. Horizontal and Vertical STARs

connections with other cells or input/output (I/O) blocks. The latter category includes long lines, that extend horizontally along an entire row or vertically along an entire column. While only one STAR would be sufficient to test all the PLBs in the FPGA, testing of both vertical and horizontal global busses and of the connections between them requires separate vertical and horizontal STARs. Having two roving STARs also provides additional benefits for diagnosis and fault tolerance that will be discussed in the next sections. Testing the entire FPGA takes one full horizontal sweep of V-STAR and one full vertical sweep of H-STAR. The direction of roving changes after every full sweep: left→right, up→down, right→left, and down→up. For an $N \times N$ FPGA, a full sweep of one STAR requires $N/2$ positions, thus the total number of STAR movements is N . An important consequence is that the fault latency increases linearly with the size of the FPGA.

To preclude the usage of the spare resources by the FPGA design tools, the spare PLBs in both STARs, the vertical routing tracks in V-STAR, and the horizontal routing tracks in H-STAR are designated as reserved. Connections among the working PLBs are allowed to use only horizontal wire segments through V-STAR, and only vertical segments through H-STAR; the same rule also applies to connections with the I/O blocks of the FPGA. For an $N \times N$ FPGA, the area overhead taken by the two STARs is $OV(N) = 1 - (N-2)^2/N^2$. The overhead decreases with N . For example, for $N=20$, OV is 19 percent, but for $N=40$, OV is only 10 percent. Another part of the overhead consists of routing resources for a four-wire bus around the perimeter of the array, needed for the links between the boundary-scan test access port (TAP) controller and BIST circuitry within the STARs.

3.3 The Roving Mechanism

Once a STAR has been tested, the STAR roving process continues by relocating the working area adjacent to the STAR over the current STAR position, and by reconfiguring the just-released working area as a new STAR. Figure 3 illustrates several aspects of this procedure by following a section of one row during several steps. Figure 3a shows the initial state, where A, D, E, and F are working PLBs, separated by B and C which are part of V-STAR. Assume all test activity has stopped without detecting any faults, while the normal operation continues. TREC performs the following operations: (1) Configure B and C with the functions of D and E; (2) Stop the system clock; (3) If necessary, copy the state of D and E into B and C; (4) Reconfigure to disconnect the nets connected to D and E and connect them to B and C; (5) Restart the system clock; (6) Configure the new STAR and restart testing. The critical steps are (3) and (4), which must be as fast as possible to minimize the period when the system clock is suspended; these steps are explained next.

Step (3) – copying the state values from the old to the new locations of the relocated working PLBs – is not needed for PLBs that are used only as combinational functions. Otherwise, if the relocated functions include registers and/or RAM, the implementation of this step depends on the features of the target FPGA. For an FPGA that allows direct reading and writing of registers by treating them as addressable memory locations, the task is straightforward. Although a little more complicated, solutions based on partial configuration readback, available on ORCA series 3C [16], are also easy to implement. For FPGAs without such features, a general solution is to use an additional configuration that creates temporary transfer paths through the STAR from the source PLBs to the destination PLBs and issue a single clock (derived from the test clock TCK) to copy

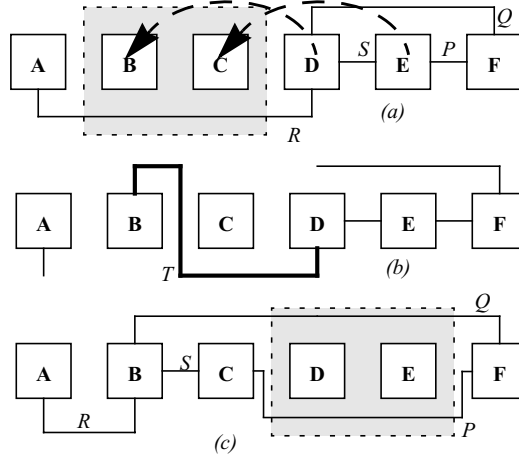


Figure 3. Roving Steps

the values. Figure 3b illustrates a path T created to transfer values from D to B (we assumed that E implements a combinational function). In addition to a transfer path, copying RAM contents requires the creation of a transfer controller to sequence both the source and the destination RAMs concurrently through their address space and to issue the proper read and write signals for the desired transfer. The transfer controller can be placed in the STAR not involved in the current roving step (H-STAR in our example).

Step (4) maintains the correct interconnections among working PLBs; the results of this step can be seen in Figure 3c. Net Q , originally between F and D, has been extended to connect to B. In contrast, net R has been shrunk since now it connects adjacent PLBs. Net P , originally a direct connection between E and F, has been routed via two vertical segments and a horizontal one. Net S has been moved with the PLBs it connects.

These changes are done by an incremental rerouting of the affected area of the chip, using the same netlist, changing the placement of the relocated logic, marking the vertical tracks in the new STAR position as unusable, and saving the resulting incremental configurations. For a full horizontal swap we need to repeat this process $N/2$ times. We are using a standard router for this job. Since computing these roving configurations is not done on-line, but as a preprocessing step, the cost of the N incremental routing jobs is acceptable. A more efficient incremental routing procedure is described by Dutt et al. in [10], but this requires a special-purpose router.

Inserting a STAR in the working area introduces a delay overhead in the signals that now have to traverse the STAR, typically corresponding to a two-cell long wire segment. Although relatively small, this overhead must be compensated by a slack in the operating clock frequency. In the worst case, a signal in the application circuit's critical path may be split by one or two STAR widths. To investigate this performance penalty, we used seven benchmark circuits. First we mapped the circuit as a non-FT FPGA design using commercial computer-aided design (CAD) tools, with options set for performance optimization. Then we mapped the same set of circuits to the same FPGA with two inserted STARs. The results presented in Table 1 are based on implementing the seven benchmarks with ORCA 2C15A, and show the number of PLBs, the initial maximum delay, the maximum delay in the presence of STARs, and the percent increase. The

maximum delays were computed for the 10 positions of the V-STAR. For six circuits, the delay increase caused by the insertion of the STARS is between 0 and 16.3 percent.

Table 1. Performance Penalty with STARS

Circuit	# PLBs	Initial Delay (sec)	Delay w/ STARS (sec)	% diff
Huffman	40	87.5	98.9	11.5
Fibonacci	70	130.7	130.8	0.0
Wallace tree multiplier	65	144.3	149.7	3.6
Digital Single Sideband Modulator	229	75.1	76.6	2.0
Hilbert	192	73.2	87.5	16.3
Random number generator	134	49.4	56.4	11.5
Mono-FFT	62	117.8	120.8	2.5

4. On-Line Testing and Diagnosis

4.1 Basic BIST Structure

A STAR contains several disjoint tiles that are tested concurrently. Figure 4 shows the basic BIST structure used in a tile, called a BISTER, which inherits some of the concepts used in our off-line FPGA BIST methods for PLBs [56, 55, 2] and programmable interconnect [57]. A BISTER contains a TPG applying test patterns to two identically configured blocks under test (BUTs) in case of logic BIST, or two sets of wires under test (WUTs) in the case of routing BIST. The outputs of the BUTs or WUTs are compared by an ORA. The ORA latches and reports mismatches as test failures. Direct comparison avoids possible aliasing errors introduced by compressing responses into signatures [1]. *BIST Start/Reset* and *Pass/Fail* are two of the interface signals with the TREC via the boundary-scan access mechanism. *Start/Reset* is used to initiate the BIST sequence and to reset the TPG and ORA. The test result *Pass/Fail* is captured in a FF which is part of a scan register [18]. The other two inputs from the TREC/boundary-scan interface include *TCK* for clocking the BIST circuitry and a control input for scanning out the *Pass/Fail* results from each BISTER. The first configuration of a BISTER checks the proper operation of the scan register, inducing mismatches by comparing BUTs or WUTs with different configurations. This also protects against the case of all ORA FFs being stuck at the “Pass” value.

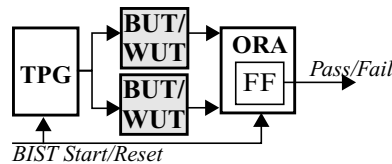


Figure 4. Basic BISTER

The TREC accesses an FPGA using its boundary-scan mechanism [51], so that this access is transparent to the normal function of the chip (most FPGAs support boundary scan). TREC employs the boundary-scan interface to reconfigure the STARs for different test and diagnosis operations, to initiate the BISTERS, and to scan out the test results. BISTERS work with the boundary-scan clock *TCK*, and the configuration clock is also derived from *TCK*. TREC also controls the system clock(s) and has the capability to stop the system operation for short intervals to allow for safe relocation of the system logic in the last tested area. If faults are detected, TREC starts the diagnosis process (to be discussed later).

Figure 5 illustrates a typical ORA that compares four pairs of outputs coming from the two BUTs (or WUTs). The feedback latches the first mismatch into the FF. Higher diagnostic resolution could be obtained by constructing an independent ORA (comparison and latching FF) for each pair of outputs to be compared. However, this would increase the *Pass/Fail* result data to be retrieved by a factor of four. While higher diagnostic resolution is important for the FT techniques, there is a tradeoff between diagnostic resolution and the total testing time which, in turn, determines the fault latency. Therefore, to minimize fault latency, we break the fault detection and fault diagnosis into two separate processes when necessary. We apply diagnostics procedures only after we have detected faults in the STAR. Accordingly, we break our discussion of the fault

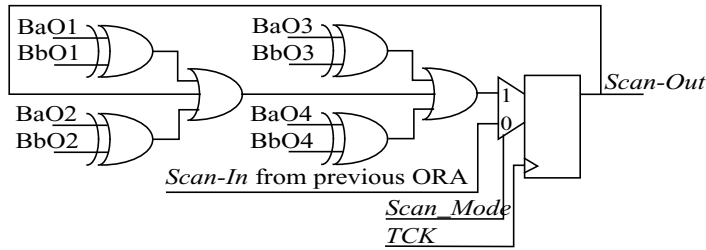


Figure 5. Integrated ORA/Scan cell

detection and diagnostic processes into the following separate subsections for both logic and interconnect.

4.2 Testing Programmable Logic Blocks

Figure 6 shows a typical structure of a PLB, composed of a memory module, a register, and a combinational output module. This structure is found in most current RAM-based FPGAs. The memory block can implement combinational lookup tables (LUTs) or a RAM with various modes of operation (synchronous, asynchronous, single-port, dual-port, etc.). A PLB also contains special-purpose logic for arithmetic functions (counters, adders, multipliers, etc.) The register can be configured as FFs or latches with programmable clock-enable, preset/clear, and data selector functions.

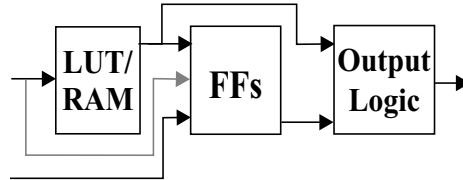


Figure 6. Typical PLB Structure

The BUTs are repeatedly configured to be pseudoexhaustively tested in every mode of operation; this means that every module in a BUT is exhaustively tested [39]. The configuration of the TPG also changes when a new BUT configuration requires different patterns. For example, to test combinational functions implemented by a LUT with n inputs, the TPG is a counter that generates all possible 2^n vectors, while to test the RAM, the TPG is a state machine generating standard RAM test sequences [59], which are known to be exhaustive for the fault models specific to RAMs, such as pattern-sensitive faults. The ORA needs to be reconfigured when the new mode of operation of BUT involves a different number of outputs.

A configuration multiplexer (MUX) is a commonly used hardware mechanism that selects subcircuits for various modes of operation. A configuration MUX is controlled by configuration memory bits to select one input to be connected to its output. In Figure 7a, assume that we set the configuration bit S to 0 to connect $V0$ to X . Then the subcircuit producing $V1$ disappears from the circuit model seen by the user. This is correct from a design viewpoint, because the value $V1$ can no longer affect X in the current configuration. But from a testing viewpoint, in any test for the MUX, we need to set $V0$ and $V1$ to complementary values. In general, for a MUX with k inputs, if V is the value of the selected input, all the other $k-1$ inputs should be set to value \bar{V} .

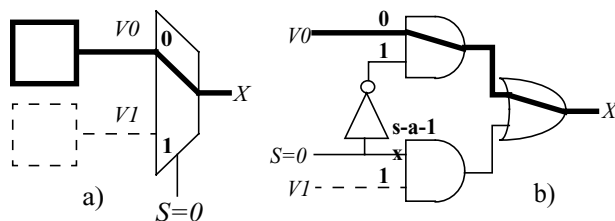


Figure 7. Configuration Multiplexer

The problem arises because FPGA CAD tools generate the configuration bitstream based on the user model, which will never include the functionally inactive subcircuits (called invisible logic in [56]). Thus in Figure 7a, when $S=0$, $V0$ will be set to both 0 and 1, but $V1$ cannot change. Similarly, the user logic cannot control $V0$ in any configuration where $S=1$. The result is that the testing of the MUX may not be complete. For example, the s-a-1 fault in the gate-level MUX model in Figure 7b is detected only when $S=0$, $V0=0$, and $V1=1$. But this pattern may never be applied if $V1$ cannot be controlled when $S=0$.

Our solution relies on separately configuring the invisible logic so that it will generate the proper values needed for the inactive MUX inputs. Then we overlay the resulting configuration files over the main configuration file with the active logic, and we merge them without changing any MUX setting done in the main configuration. This process is conceptually simple, but its implementation requires knowledge of the FPGA configuration stream structure.

Note that in most previous work dealing with testing FPGAs, the problem of testing a configuration MUX is either not addressed or it is solved functionally, by connecting every input in turn to the output, and providing both 0 and 1 values to the selected input. However, the invisible logic driving the inactive inputs is completely ignored. Hence prior claims of complete testing may not be valid since the testing of every configuration MUX in the FPGA is likely to be incomplete. Methods that did provide complete tests for configuration multiplexers, such as [27, 55], used models that did not remove the invisible logic. But such models cannot be used with the existing FPGA CAD tools to generate configuration files.

Since a BISTER provides complete testing only for its BUTs, we have to reconfigure every BISTER several times so that every PLB will be a BUT in at least one configuration. The BUTs are repeatedly configured to be tested in all their modes of operation. The modes of operation of a PLB may be determined only from the information available in the FPGA data book, without having a detailed knowledge of its implementation. The test time for a BISTER, and thus total roving time, is dominated by the reconfiguration time, which is much larger than the BIST time. Hence to reduce the latency of our procedure, we try to minimize the number of PLBs in a BISTER. The number of PLBs for an ORA and for a TPG depend on the target PLB architecture. Usually the number of outputs in a PLB is smaller than its number of inputs. Since a TPG must provide exhaustive patterns for BUTs, we will need more than one PLB to construct the TPG. For illustration purposes, we will assume that a TPG needs three PLBs and an ORA only one. (Our analysis, however, is independent of the number of PLBs used in implementation.)

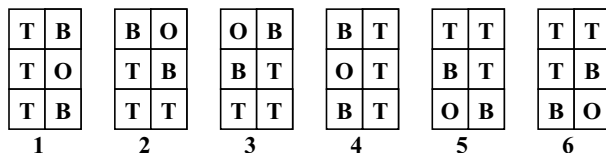


Figure 8. 3 by 2 BISTER Tile Rotations

Figure 8 illustrates six floor plans of a 3 by 2 BISTER tile, where T, B, and O denote, respectively, a TPG cell, a BUT, and an ORA cell. The goal of the six configurations is to systematically rotate the functions of the PLBs, so that eventually every PLB in the tile is completely tested twice, each time being compared with a different BUT. This rotating strategy assures that every single faulty PLB and almost all combinations of multiple faulty PLBs (except for a few pathological cases described in [2]) are guaranteed to be detected. If the number of PLBs in a STAR is not a multiple of the number of PLBs in a tile, then the leftover PLBs that could not make up a BISTER will be grouped with some of the PLBs already tested in the adjacent tile, so that every PLB in the STAR will eventually be part of a BISTER.

Figure 9 illustrates the floor plan of the 4 by 2 BISTER tile used in our implementation with the ORCA 2C series FPGA and its associated eight test configurations (S indicates a spare PLB). The role of the spare cells is to provide additional routing resources to overcome routing congestion problems. We arrange the number of spare PLBs so that a tile will have an even number of PLBs, symmetrically distributed between the two columns of the V-STAR. Similar to the 3 by 2 BISTER tile, the eight configurations shown in Figure 9 systematically rotate the functions of the PLBs, so that every PLB in the tile is completely tested twice, each time being compared with a different BUT. To overcome routing congestion problems, spare PLBs may be incorporated into the BISTER tile to provide additional routing resources. While these are not pure rotations, since they maintain the 3 TPG cells in the same column, they do achieve the same property as the rotations of the 3X2 tile. This rotating strategy allows us to prove the following properties regarding the fault detection capability of a BISTER [4]:

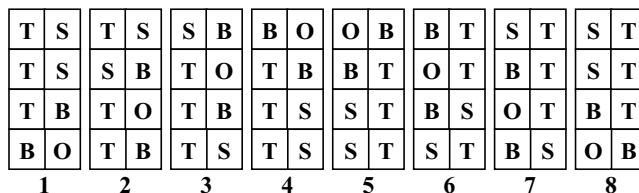


Figure 9. 4 by 2 BISTER Tile Rotations

Claim 1: Any single faulty PLB is guaranteed to be detected in at least two BISTER configurations.

Proof: The faulty PLB is a BUT in two BISTER configurations, where its exhaustive inputs patterns are produced by a fault-free TPG, and its outputs are compared with a fault-free BUT by a fault-free ORA. Hence no fault (single or multiple) detected in the BUT can escape detection in these two configurations.

Claim 2: Except for a few pathological cases, any pair of faulty PLBs is guaranteed to be detected in at least one BISTER configuration.

Proof: Since any single faulty PLB is detected, we must have a circular masking relation [1], where one faulty PLB masks the detection of the faults in the other and vice versa. This masking should occur in any BISTER configuration where a single faulty PLB would be detected. First we note that any masking relation between the two faulty PLBs disappears whenever one of them becomes a spare. Because a TPG or an ORA containing a faulty PLB may still work correctly, we analyze only configurations when at least one of the faulty PLBs is a BUT and the other is not a spare.

Case 1: Both faulty PLBs are BUTs in one BISTER configuration. Then the TPG and the ORA are fault-free, and the only way the faulty pair can escape detection is to have functionally equivalent faults, since then their identical outputs will not mismatch. Assume that the faulty PLBs are BUTs in the configuration 1 in Figure 10. Then in configurations 3 and 5, one of them is a BUT (compared with a fault-free BUT by a fault-free ORA), and the other is a TPG cell. The only way the faulty pair can escape detection is for the faulty TPG to skip exactly the patterns that detect the faulty BUT. We say that this is a pathological case, because it has an extremely low probability of occurrence: we need two faulty PLBs, and they must have functionally equivalent faults, and when a faulty PLB is part of the TPG, the TPG must skip those patterns that detect the faulty BUT.

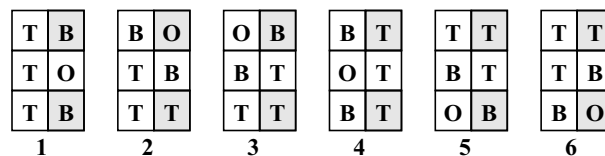


Figure 10. BISTER Tile Rotations (Two Faulty BUTs)

Case 2: At most one faulty PLB is a BUT in any BISTER configuration. Assume, without loss of generality, that the faulty cells are in the first row of the BISTER, as shown in Figure 11. Let X be the faulty BUT and Y the faulty TPG cell in configuration 1. To have circular masking between X and Y , all of the following conditions must be true: 1) in configuration 1, the TPG must skip those patterns that detect X ; 2) in configuration 4, the TPG must skip those patterns that detect Y ; 3) in configuration 2, the ORA X must not record the mismatch between Y and the fault-free BUT; and 4) in configuration 3, the ORA Y must not record the mismatch between X and the fault-free BUT. Again, this is a pathological situation because we need the AND of four conditions, which are very unlikely by themselves.

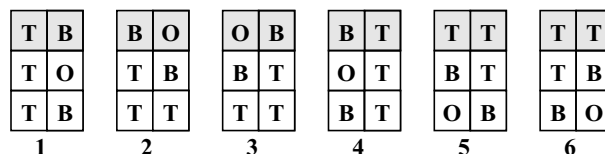


Figure 11. BISTER Tile Rotations (One Faulty BUT)

Clearly, similar arguments can be made for more than two faulty PLBs. Hence we can conclude that:

Claim 3: In practice, any combination of faulty PLBs is detected in at least one BISTER configuration.

Table 2 summarizes the 14 test phases we have developed to completely test a PLB in our implementation of the roving STARs approach using the ORCA 2C and 2CA series FPGAs [16]. The last five phases apply only to the 2CA series PLB. The FF modes can be tested at the same time with five of the LUT modes to reduce fault latency (as illustrated in Table 2), or we can test them separately to enhance diagnostic resolution. We chose to minimize fault latency in our implementation at the expense of diagnostic resolution. Thus complete testing of a PLB requires 14 BISTER configurations. One additional BISTER configuration is added to each set of test phases to test the integrated ORA/scan chain for a total of 15 test phases per rotation of the BISTER.

Table 2: Summary of BIST Phases for ORCA 2C and 2CA Logic Blocks

Phase No.	FF/Latch Modes & Options					LUT Mode	No. Outs	Logic Tested		
	FF/Latch	Set/Reset	Clock	Clock Enable	FF Data			LUT	FFs	MUX
1	-	-	-	-	-	Asynchronous RAM	4	√		√
2	-	-	-	-	-	Adder/subtractor	5	√		√
3	-	-	-	-	-	5-variable MUX	4	√		√
4	-	-	-	-	-	5-variable XOR	4	√		√
5	FF	async. reset	falling edge	active low	LUT output	Count up	5	√	√	√
6	FF	async. set	falling edge	enabled	PLB input	Count up/down	5	√	√	√
7	Latch	sync. set	active low	active high	LUT output	Count down	5	√	√	√
8	FF	sync. reset	rising edge	active low	PLB input	4-variable	4	√	√	√
9	Latch	-	active high	active low	dynamic select	4-variable	4		√	√
10	-	-	-	-	-	Multiplier	5	√		
11	-	-	-	-	-	Greater/equal to Comp	5	√		
12	-	-	-	-	-	Not equal to Comp	5	√		
13	-	-	-	-	-	Synchronous RAM	4	√		
14	-	-	-	-	-	Dual port RAM	4	√		

The operational modes of the PLBs require between 9 and 12 inputs be supplied with test patterns. Since a 4-bit counter can be easily implemented in a single PLB, we use three PLBs for a TPG. An ORA consisting of a 4-bit comparator and a FF to latch mismatches is implemented by a single PLB. Figure 12 shows the actual layout of a BISTER tile in the H-STAR. Two spare PLBs (denoted by S/O and SP), configured as identity functions, have been added to avoid the use of global vertical routing resources. The layout also shows the boundary-scan TAP controller (denoted by BS TAP), which is not part of the BISTER tile [18]. Since the tile has 8 PLBs, we will have 8 rotations of this layout (see Figure 9), and every BISTER will be reconfigured 15 times for a total of 120 configurations per STAR to completely test all of the PLBs within that STAR.

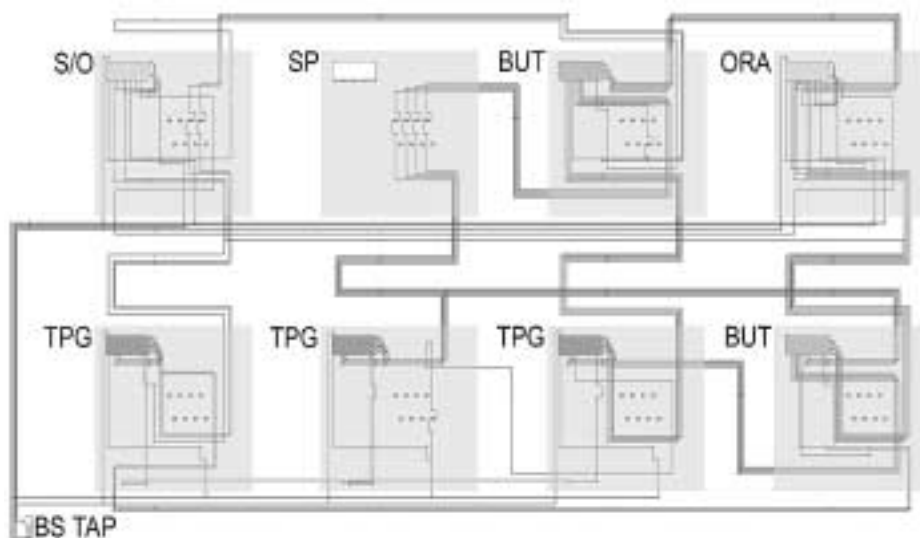


Figure 12. Layout of a BISTER Tile

Partial reconfiguration reduces the time required for downloading the BISTER tile configurations (and therefore the fault latency), and it also significantly reduces the amount of storage required for the BISTER tile configurations. For example, the complete configuration memory for the 2C15A consists of about 225 Kbits. Partial configuration of the complete STAR requires about 23 Kbits. However, by performing partial configuration for only those portions of the STAR that must be changed from one BISTER tile configuration to the next requires only between 3 Kbits and 10 Kbits, depending on how much of the STAR is being reconfigured; changing the mode of operation of the BUTs requires the smallest number of bits, while rotating the BISTER tile to begin testing a new set of BUTs requires the largest number of bits. As a result, only about 7 percent of the files are of the larger size (10 K), while 93 percent of the files are the smaller size (3 K), for a total of about 420 K bits of storage required for the PLB BIST configurations.

4.3 Divide-and-Conquer Adaptive Diagnosis

The goal of our original approach was to have a unique procedure able to locate any combination of multiple faulty PLBs in a BISTER without making any restrictive assumption about the multiplicity of the fault. We start by considering all the PLBs in a failing BISTER as suspects. Figure 13 illustrates this adaptive diagnosis procedure. The V-STAR has a failing BISTER with six suspected PLBs. First they are split in two sets – $\{A,B\}$ and $\{C,D,E,F\}$ – which are included in separate BISTERS whose other PLBs were shown to be fault free. Next we assume that the upper BISTER passes all its tests, while the lower one fails. The remaining suspects are again split in two sets – $\{C,D\}$ and $\{E,F\}$ – and included in separate BISTERS. Next we assume that the E and F are found to be fault free. But now the remaining suspects – C and D – reside in the same row and may not be included in separate BISTERS. To complete the diagnosis, we rove H-STAR to bring it over the row with C and D , which now may be tested in separate horizontal BISTERS.

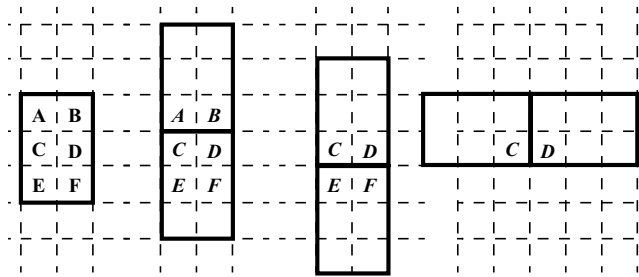


Figure 13. Example of Adaptive PLB Diagnosis

The time for the diagnosis procedure is determined by the number of partial reconfigurations needed to locate all the faulty PLBs. Even in the best case, when we have a single faulty PLB, we need at least three different STAR configurations, and for each one of them, the full set of six or eight rotations. In addition, several roving steps may be needed to bring the second STAR to cover the remaining suspected PLBs, where each roving step is done by another partial reconfiguration. Long run-time for diagnosis is not critical, because the normal system operation continues without interruption during this period [15]. However, it may result in delaying the detection of a new fault occurring during the execution of the diagnosis procedure.

A second problem with this procedure results from the need to group suspected PLBs with adjacent known-good ones. During the life of a fault-tolerant system, the set of fault-free spares gets gradually smaller as more faults occur, and fault-free PLBs may no longer exist in an area where they are needed to help in diagnosing suspects. In such a situation the procedure would be no longer applicable.

To illustrate a third shortcoming of the design-and-conquer technique, consider the situation shown in Figure 14, where we have the suspects A , B , C , and D . Recall that in [2] H-STAR and V-STAR move by two PLBs each roving step and thus each STAR can only be positioned in one of $N/2$ two-cell-wide positions in the FPGA. Then in V-STAR we can place $\{A, B\}$ and $\{C, D\}$ in separate BISTERS, and in H-STAR we can separate between $\{A, C\}$ and $\{B, D\}$. But if all these four BISTERS fail, we cannot separate the suspects. Indeed both diagonal pairs of PLBs – $\{A, D\}$ and $\{B, C\}$ – would cause failures in all four BISTERS. Similarly, all sets of three faulty PLBs and the group of all four faulty PLBs in the 2 by 2 area located at the intersection of the V-STAR and H-STAR are indistinguishable. This problem could be solved by allowing STARs to have one-PLB-wide roving steps for diagnosis, but this solution would double the number of configurations to be stored by TREC, and significantly complicate the control programs.

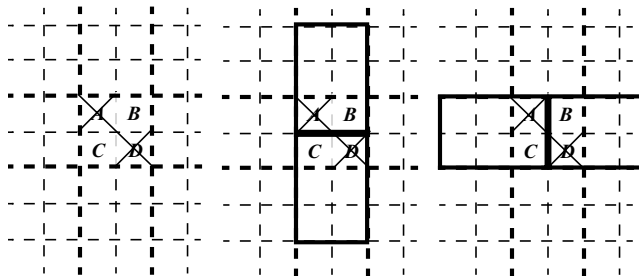


Figure 14. Example of Inconclusive Diagnosis

4.4 Locating Faulty PLBs Using BISTER Results

In every test configuration, TREC records the set of failures obtained at the ORA outputs. Our improved diagnosis method consists of two stages. First, it tries to locate faulty PLBs based on analyzing the test results, and in many cases, this stage is sufficient to achieve maximum diagnostic resolution. The frequent case of a single faulty PLB falls in this category. For cases of multiple faulty cells not completely resolved by the first stage, a second stage employs the divide-and-conquer strategy based on additional reconfigurations [4, 3].

In any fault location procedure, maximum diagnostic resolution is achieved when faults are isolated within an equivalence class containing all the faults that produce the observed response. If every equivalence class has only one fault, we say that the fault is uniquely diagnosed, that is, there is no other fault which can produce the same response. In our case, a fault is one faulty PLB that may have several internal faults, and the response is the set of failing test configurations (also called test phases) detected at the outputs of the ORAs. We begin by assuming a single faulty PLB in the FPGA, then we analyze the case of multiple faulty PLBs, and we also discuss locating faults inside a defective PLB.

We can observe that the PLBs in a BISTER tile can be partitioned into two disjoint sets, so that every set contains the PLBs that are pairwise compared when configured as BUTs. For example, for the 3 by 2 tile, the BUTs in configurations 1, 3, and 5 are compared only among themselves and they are never compared to the BUTs in phases 2, 4, and 6. A similar partition exists for the 4 by 2 tile as well. We can represent the relations between BUTs that are pairwise compared and the ORAs that observe them by the graphs shown in Figure 15, where a BUT is denoted by a node B_i , and the ORA that observes BUTs B_i and B_j is denoted by O_{ij} . We can view such a graph as representing a combined test session, in which half of the BUTs in the tile are concurrently tested. Each tile is completely tested in two combined test sessions, where the BUTs in one session are ORAs in the other one, and vice versa.

Theorem 1: Any single faulty PLB is guaranteed to be uniquely diagnosed.

Proof: When the faulty PLB is configured as a BUT, it is detected at its two adjacent ORAs, and no other BUT is detected at the same two ORAs. Note that when the faulty PLB is configured as a TPG cell, no error is generated even when the fault modifies the TPG patterns, because the two

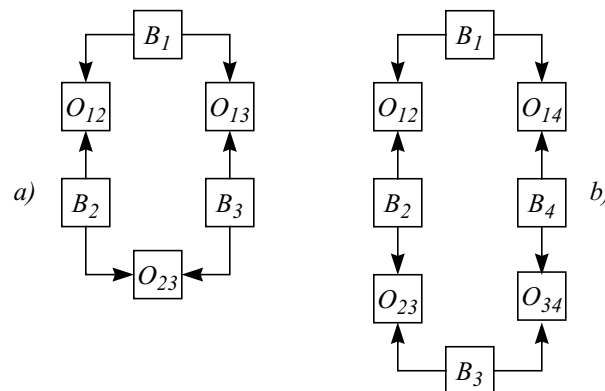


Figure 15. Combined test sessions a) 3 by 2 tile b) 4 by 2 tile

BUTs still receive the same patterns. When configured as an ORA, the faulty PLB may generate an error, but this will be in addition to the failures observed in the combined session when it is a BUT, and these results are sufficient to distinguish it from any other single faulty PLB.

As a consistency check, we can verify that the sets of failures obtained at the two ORAs are identical.

The following results deal with the location of a group G of faulty PLBs that is detected in at least one test session. To uniquely diagnose G means to identify all its faulty blocks such that no other group (including subsets of G) can produce the same result. Although unique diagnosis is not always possible, there are many situations when it can be guaranteed.

We will analyze the interactions among faulty PLBs under the following assumptions:

Assumption 1: A TPG with faulty PLBs does not skip the patterns that detect faults in a BUT.

Assumption 2: A faulty PLB is not detected when it is configured as an ORA.

Based on these assumptions, the set of failing phases obtained at ORA O_{ij} is given by the following:

$$FO_{ij} = (FB_i \cup FB_j) - Feq_{ij} \quad (1)$$

where FB_k is the set of failing phases of BUT B_k , and Feq_{ij} is the set of failing phases of both B_i and B_j that have identical responses (and thus do not cause mismatches at O_{ij}). In Figure 16, the area of FO_{ij} is marked by diagonal lines. Note that FO_{ij} is empty (\emptyset) when both B_i and B_j are fault free, or when the faults in B_i and B_j are equivalent (since then $FB_i = FB_j = Feq_{ij}$). Also note that there exists one situation when FO_{ij} is the same, no matter if only one or both of the BUTs observed at O_{ij} are faulty; this occurs if the two BUTs have the same sets of failing phases ($FB_i = FB_j$), but their faulty responses are never the same ($Feq_{ij} = \emptyset$).

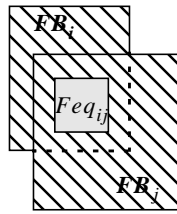


Figure 16. Figure for Equation (1)

Knowing the set of failing phases observed at O_{ij} and the complete set of failing phases of one of the two faulty BUTs, we can determine the set of failing phases where the two BUTs have identical responses by the following:

$$Feq_{ij} = FB_i - FO_{ij} = FB_j - FO_{ij} \quad (2)$$

Based on FO_{ij} and one of the two sets, we can also compute a lower bound on the other set by

$$FB_i \supseteq (FO_{ij} - FB_j) \cup Feq_{ij} \quad (3)$$

or by

$$FB_j \supseteq (FO_{ij} - FB_i) \cup Feq_{ij} \quad (4)$$

Note that Equation (3) becomes an equality when FO_{ij} and FB_j are disjoint:

$$FB_i = FO_{ij} \cup Feq_{ij} \quad (5)$$

This occurs when $FB_i \subseteq FB_j$ and $Feq_{ij} = FB_i$.

For the diagnosis procedure, we will make one more assumption:

Assumption 3: No more than two faulty BUTs have identical responses in the same failing phase.

To justify this assumption, recall that in every configuration the TPG applies exhaustive tests for that mode of operation. If Assumption 3 is not satisfied, it means that we would have three or more BUTs with equivalent faults for that mode of operation, which is quite an unlikely event.

The following results will be used by our diagnosis procedure.

Lemma 1: If none of the two ORAs observing BUT B fails in phase p , then B does not fail phase p .

Proof: Assume, by contradiction, that B fails phase p . Based on Assumption 1, the vectors detecting its faults are generated by the TPG. Then the other two BUTs observed by the two ORAs must have equivalent faults to the fault of B in phase p , because p is not reported by either of the two ORAs. But then we would have three BUTs with equivalent faults in the same phase, which contradicts Assumption 3. Therefore, B does not fail phase p .

Lemma 2: If the two ORAs observing the same BUT report no failures, then their common BUT is fault free.

Proof: Based on Lemma 1, the common BUT of the two ORAs does not fail any phase.

Lemma 3: Any BUT failure appears at least at two ORAs.

Proof: Assume, by contradiction, that phase p is reported only at one ORA. Let B be the BUT failing p , and O be the other ORA observing B . Since O does not report p as a failure, the other BUT (say C) observed by O must have identical response in p . But the other ORA observing C does not report p as a failure either, so we have more than two BUTs with equivalent faults in p , which contradicts Assumption 3. Therefore, p must appear at least two ORAs.

Lemma 4: If we have failures only at two ORAs, they must have identical failures.

Proof: Based on Lemma 3, any failure must appear at least twice, and in this case we have only two failing ORAs. Hence their failures must be identical.

Theorem 2: (For the 4 by 2 tile) If only two ORAs observing the same BUT report failure in phase p , their common BUT fails in phase p .

Proof: In Figure 17, without loss of generality, assume that O_{12} and O_{14} are the two failing ORAs reporting p as a failure. Assume, by contradiction, that their common BUT B_1 does not fail phase p . Then both B_2 and B_4 must fail p . But since no other ORA reports p , then B_3 must also fail in phase p , and its response must be identical to that of B_2 and B_4 . Hence we would have three BUTs with equivalent faults in phase p , which contradicts Assumption 3. Therefore, B_1 must fail phase p .

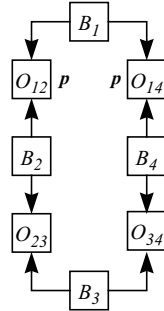


Figure 17. Figure for Theorem 2

Lemma 5: If two ORAs observing the same BUT report no failures, then all three BUTs observed by them are fault free.

Proof: Based on Lemma 2, the common BUT of the two ORAs is fault free. So each one of the other BUTs is compared with a fault-free one by an assumed fault-free ORA, and no failure is reported. Hence the other two BUTs are also fault free.

Lemma 6: (For the 4 by 2 tile) If only two ORAs without a common BUT fail phase p , then at least one pair of BUTs between the two ORAs are faulty and have identical response in phase p .

Proof: In Figure 18, without loss of generality, assume that O_{12} and O_{34} are the two ORAs without a common BUT failing phase p . At O_{12} , p may be caused by B_1 or B_2 . Let us assume that B_1 fails p . Because p is not observed at O_{14} , B_4 must also fail p with an identical response. Similarly, had we assumed that B_2 fails p , we would have concluded that B_2 and B_3 have equivalent faults in phase p . Note that all four BUTs may be faulty as well, but in this case the equivalent faults in the two pairs will be different.

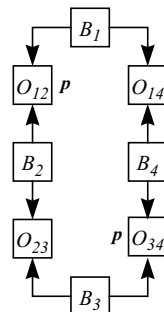


Figure 18. Figure for Lemma 6

Note that under the condition of Lemma 6, we cannot reach a unique diagnosis. However, we can use the reduced set of suspects as a starting point for the divide-and-conquer technique to test some of the suspected PLBs in separate BISTER tiles. A subsequent example will show how the combined technique can uniquely diagnose the faulty PLBs.

Lemma 7: Assume that two out of three ORAs fail, such that the middle one has no failures, and the other two have disjoint failures. Then the two BUTs observed by the nonfailing ORA are fault free and the other two BUTs are faulty.

Proof: In Figure 19, without loss of generality, assume that O_{34} has no failures, while O_{14} and O_{23} have disjoint failures. Then we want to prove that B_3 and B_4 are fault free and B_1 and B_2 are faulty.

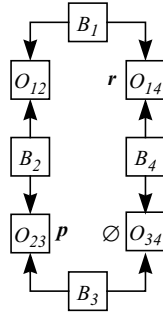


Figure 19. Figure for Lemma 7

Assume, by contradiction, that one of the two BUTs observed by the nonfailing ORA (say, B_3) fails phase p . Since O_{34} reports no failures, B_4 must have an equivalent fault in p . At most one of O_{14} and O_{23} may fail p , because their failures are disjoint. Then we have to analyze two cases:

Case 1: One of the failing ORAs (say, O_{23}) shows p as a failure. Since p is not observed at O_{14} , while B_4 fails p , B_1 must also have an equivalent fault in p . In this case we would have three BUTs with equivalent faults in p , which would contradict Assumption 3.

Case 2: None of the failing ORAs reports p as a failure. But this would mean that both B_1 and B_2 must also have equivalent faults in p , since p does not appear at O_{14} or O_{23} . In this case we would have four BUTs with equivalent faults in p , which would contradict Assumption 3.

Therefore, both B_3 and B_4 are fault free. Then B_2 is the source of the failures recorded at O_{23} , while those observed at O_{14} originate at B_1 .

As a consistency check, we can verify that the failures recorded at the ORA between the faulty BUTs are the union of the disjoint sets of failures obtained at the other two failing ORAs (for the example above, $FO_{12} = FO_{14} \cup FO_{23}$).

The following examples deal with the results of only one combined test session. Under the assumption that a faulty PLB is detected only when configured as a BUT, the two sessions can be independently analyzed, and a different group of faulty BUTs may be identified in each session.

Example 1: Figure 20 shows the set of failing phases obtained at the four ORAs in one combined test session for a 4 by 2 BISTER tile. Because failures in test phase 1 are reported only at O_{23} and O_{34} , then B_3 is faulty and fails in phase 1 (by Theorem 2). In the same manner, B_4 is identified as failing phase 5, B_1 is identified as failing phase 9, and B_2 is identified as failing phase 7. Since failing phase 8 is reported at three ORAs, Theorem 2 does not apply and we cannot determine which BUTs fail phase 8 (should be at least two). Note that we have identified all four BUTs as faulty.

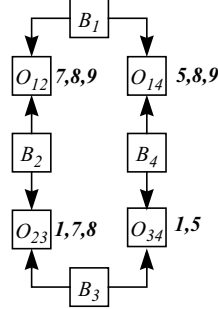


Figure 20. Figure for Example 1

If our goal is also to identify the exact set of failing phases for every faulty PLB (to be able to reuse its fault-free modes of operation as a partially usable block), we can use the divide-and-conquer technique to obtain additional information. Let us assume that we test B_3 in a separate BISTER with known fault-free PLBs and we find out that $FB_3 = \{1, 8\}$. Then from Equation (2) we obtain $Feq_{34} = \{1, 8\} - \{1, 5\} = \{8\}$, and from Equation (4) we determine that B_4 must also fail phase 8. At this point we still do not know whether B_1 or B_2 is causing O_{12} to report failure in phase 8, so we again use the divide-and-conquer technique and test B_1 in a separate BISTER. Let us assume that we get $FB_1 = \{9\}$. Then from Equation (2) we obtain $Feq_{12} = \emptyset$, and from Equation (4) we determine that B_2 must also fail phase 8. Now we know the exact sets of failing phases for every BUT.

Note that the original divide-and-conquer technique by itself cannot diagnose this quadruple fault, because no matter how the original tile is divided, any BISTER will contain one faulty PLB, and the initial set of eight suspected PLBs will never be reduced. In contrast, our analysis technique found all of the four faulty PLBs without any reconfiguration, and relied on the information provided by two steps of divide-and-conquer only to determine the exact sets of failures for the faulty PLBs.

Example 2: Assume that we have $FO_{23} = FO_{34} = \{3, 5\}$ and no failures at O_{12} and O_{14} . Based on Lemma 5, we conclude that B_1 , B_2 , and B_4 are fault free, and from Theorem 2 we determine that B_3 is failing phases 3 and 5.

This example has illustrated the common pattern of a single faulty PLB.

Example 3: Assume that we have $FO_{23} = \{2, 5\}$, $FO_{14} = \{3, 4\}$, and no failures at O_{34} (see Figure 21). Because FO_{23} and FO_{14} are disjoint while O_{34} reports no failures, the conditions of Lemma 7 are satisfied, and we can conclude that B_3 and B_4 are fault free and B_1 and B_2 are faulty.

Moreover, from Equation (1) we find that $FB_1=\{2,5\}$ and $FB_2=\{3,4\}$. As a consistency check, we can verify that $FO_{12}=\{2,3,4,5\}$.

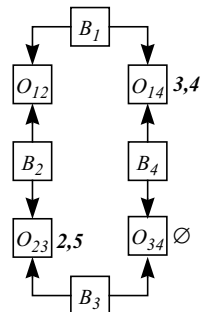


Figure 21. Figure for Example 3

Example 4: Assume that we have $FO_{23}=\{4\}$, $FO_{12}=\{3\}$, $FO_{34}=\{3,4\}$, and no failures at O_{14} (see Figure 22). Since only two ORAs with a common BUT fail phase 4, from Theorem 2 we determine that B_3 is faulty failing phase 4. Because phase 3 is a common failure for two ORAs without a common BUT, from Lemma 6 we know that at least one of the pairs $\{B_1, B_4\}$ or $\{B_2, B_3\}$ have identical failures in phase 3. This is not a unique diagnosis, so we select one of the four suspects, say B_4 , to be tested in a separate BISTER, and let us assume that we find that B_4 is fault free. From Lemma 6 it follows that B_1 is also fault free, and then B_2 and B_3 must be the faulty PLBs with identical responses in phase 3.

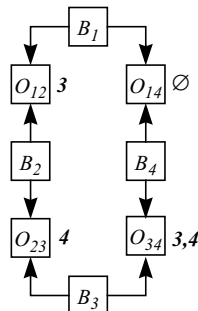


Figure 22. Figure for Example 4

Note how the combined use of failing BIST phases along with the divide-and-conquer technique allowed us to uniquely diagnose three faulty PLBs that cannot be diagnosed by either technique when used by itself.

The PLB test and diagnosis approach was applied to three known-faulty ORCA 2C15A FPGAs from Lucent Microelectronics that had been previously tested with the off-line approaches presented in [55] and [57]. One of the FPGAs was found to contain a logic fault consistent with the diagnosis result obtained in [55]. The results of the test phases for both combined test sessions are illustrated in Figure 23, where the failing test phases from Table 2 are given for each session. In Figure 23a, test phases 5 through 9 fail with failure indications given at two ORAs comparing a common BUT. Based on Theorem 2, we conclude that B_4 is faulty. In Figure 23b, we have failures only at one ORA, which, at face value, would indicate an inconsistent result as it contradicts Lemma 3. But in this case, we know from the first session that this ORA is faulty, and these

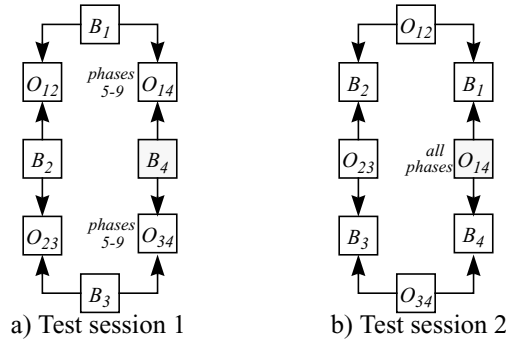


Figure 23. Diagnosis of Actual Faulty PLB

results show that Assumption 2 does not hold in this case. Thus the diagnosis of B_4 as the single faulty PLB is consistent.

4.5 Diagnosis within a Faulty PLB

During each test session, the TREC records all ORA failures. This allows us to identify the failing mode(s) of operation of the faulty PLB and its faulty internal module(s). For example, lines 1 through 9 in Table 2 summarize the test phases developed for the ORCA 2C series FPGA, while lines 10 through 14 describe the additional modes of operation tested for ORCA 2CA series. Phases 1 through 4 and 10 through 15 are used to test the different modes of operation of the LUT/RAM module of the PLB, while phases 5 through 9 are used to test the FFs and their modes of operation. Let us assume that a PLB fails only phases 5, 6, and 7. Since this faulty PLB passed the exhaustive tests for all its other modes of operation, it may be safely used for any function that does not involve counting (since phases 8 and 9 pass, the FFs work correctly as registers). This is the concept of a partially usable block [2, 15], where such a defective block is reused whenever possible in its fault-free modes of operation, to increase the effective spare capacity of the FPGA in FT applications. If phases 5 through 9 are the only ones that fail, the LUT module can still be used as a RAM or to perform any combinational logic function.

When the BIST results do not clearly indicate which component(s) of a faulty PLB is at fault, additional test phases are added to further isolate and identify the fault(s) and obtain even higher diagnostic resolution. Table 3 presents a list of additional diagnostic phases for the ORCA 2C15A FPGA. Phases 1 through 10 are used to test the output multiplexer, ensuring that each output from the LUTs and the registers can reach each output of the output multiplexer. Phases 11 through 14 can be used to isolate and test the registers in their FF mode of operation, while phases 15 through 18 can be used to test the registers in their LATCH mode of operation.

In Figure 5, the ORA FF stores the combined result of comparing four pairs of BUT outputs. Additional diagnostic resolution can be obtained by storing the result of comparing each pair of outputs separately. In this way we can determine which LUT output or which FF is faulty. Figure 24 illustrates this ORA implementation, which is feasible in FPGAs such as the Xilinx 4000 series [61] and ORCA 3C series [35]. The PLB architecture of other FPGAs, such as the ORCA 2C series [35], allows comparing only two pairs of BUT outputs per FF; in this case, we can locate the fault to only one out of two LUTs or FFs.

Table 3: Summary of Additional Diagnostic Phases for ORCA 2C and 2CA

Phase No.	FF/Latch Modes & Options					LUT Mode	No. Outs	Logic Tested		
	FF/Latch	Set/Reset	Clock	Clock Enable	FF Data			LUT	FFs	MUX
1-4	Latch	-	active high	enabled	PLB input	-	4			√
5-8	-	-	-	-	-	4-variable	4	√		√
9	FF	async. set	falling edge	active low	PLB input	-	4		√	
10	FF	async. reset	rising edge	active high	PLB input	-	4		√	
11	FF	sync. set	rising edge	active low	PLB input	-	4		√	
12	FF	sync. reset	falling edge	active high	PLB input	-	4		√	
13	Latch	async. set	active low	active low	PLB input	-	4		√	
14	Latch	async. reset	active high	active high	PLB input	-	4		√	
15	Latch	sync. set	active high	active low	PLB input	-	4		√	
16	Latch	sync. reset	active low	active high	PLB input	-	4		√	

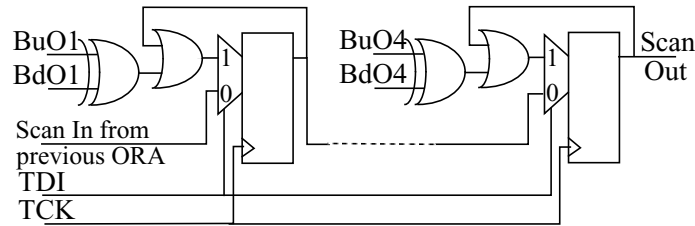


Figure 24. ORA with Greater Diagnostic Resolution

The additional diagnostic configurations shown in Table 3 were applied to the known-faulty ORCA 2C15A PLB illustrated in Figure 23 that had been previously tested with the approaches presented in [55] and [57]. Two of the FPGAs – Chip 1 and Chip 2 – exhibited routing faults consistent with the results in [57], while the third faulty FPGA – Chip 3 – was found to contain a logic fault consistent with the diagnosis result obtained in [55]. The results of the diagnostic phases indicate that the fault in this PLB only affects the four FFs, leaving the lookup tables and output multiplexer logic unaffected. Specifically, the fault appears to be located in or affecting a global resource to the FFs such as the clock input, global reset/preset, or clock enable signal. This defective PLB can be and has been successfully reused as a PUB to implement combinational logic functions in a FT application [15].

Diagnosis within a PLB to determine whether a LUT or a FF is faulty and which LUT(s) or FF(s) are faulty requires additional BIST configurations and testing sequences. Once the normal BIST and diagnostic configurations for PLBs has completed and the faulty PLBs are identified, special PUB diagnostic configurations are generated on the fly. The PUB diagnostic configurations are generated for the two or more rotations that will put the PLBs identified as faulty under test. The sliding BISTER tile used for the divide and conquer diagnosis procedure is then used to test the faulty PLBs. The actual PUB diagnostic configurations are different from those used to test the PLBs during the normal on-line BIST in that one set tests the LUTs in conjunction with the output switch box while another set tests the FFs in conjunction with the output switch box. There are eight LUT PUB diagnostic configurations, four of which are programmed with a four-input exclu-

sive-OR function while the other four are programmed with a four-input exclusive-NOR function. During each of the two sets of four LUT configurations for PUB diagnosis, the outputs of the LUTs are rotated through the four outputs of the switch box. In this way, if the error follows the rotation, the fault is known to be in a LUT, and the faulty LUT is identified as illustrated in the example of a faulty LUT in Figure 25.

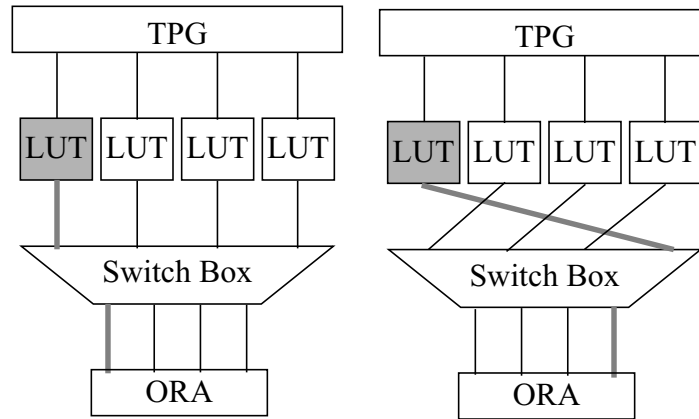


Figure 25. PUB Diagnosis.

If the error does not rotate, on the other hand, then the switch box is faulty and the faulty output is identified. The same rotation strategy is used for the FF PUB diagnostic configurations as can be seen in Figure 25 by substituting FF for LUT. Any single faulty LUT or FF can be identified and any combination of a single faulty LUT and single faulty FF can be identified with the current ORA design in the ORAC 2C FPGA. With the PLB architecture of more recent ORCA PLBs (like the 4C series FPGA), any combination of multiple faulty LUTs and FFs in the PLB can be identified. This same level of diagnostic resolution can be obtained with the ORCA 2C but at the expense of 24 PUB diagnostic configurations instead of 12 configurations.

Once the PUB diagnosis has completed for a given BISTER tile, the diagnostic procedure moves on to the next BISTER tile identified as having faulty PLBs until all PLBs that have been identified as faulty within a STAR have undergone PUB diagnosis. While this is the procedure in the current implementation, PUB diagnosis of multiple faulty BISTER tiles can be performed in parallel at the expense of slightly more complicated diagnostic software in the TREC. When PUB diagnosis has been completed within a STAR, the faulty logic resources within each PLB are passed to the logic FT software to be used to determine whether a given PLB can be utilized as a PUB by the next system function to occupy that position.

4.6 Testing Interconnect

The programmable interconnect network in most FPGAs consists of wire segments that can be connected via configuration interconnect points (CIPs). The basic CIP structure consists of a transmission gate controlled by a configuration memory bit (Figure 26a). There are three types of CIPs which we refer to as the crosspoint CIP (Figure 26b), the breakpoint CIP (Figure 26c), and the multiplexer CIP or MUX CIP (Figure 26d) [35]. While a crosspoint CIP connects wire segments located in disjoint planes (a horizontal segment with a vertical one), a breakpoint CIP connects two wire segments in the same plane. The multiplexer CIP is a group of 2^k crosspoint CIPs sharing a common output wire and controlled by k configuration bits, such that at any time only the input wire addressed by the configuration bits is connected to the output wire (in Figure 26d $k=1$). Wire segments within the programmable interconnect network are bounded by these CIPs; a given segment begins at a CIP and ends at another CIP.

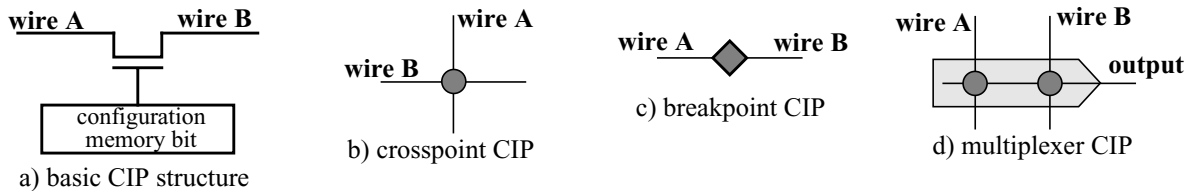


Figure 26. CIP Structure and Types

Wire segments within the FPGA are either global or local routing resources. While global routing resources are used to interconnect nonadjacent PLBs and programmable I/O cells, local routing resources are specific to a given PLB and are used to connect that PLB to global routing resources or adjacent PLBs. Figure 27a illustrates a simplified view of the global routing busses associated with a single PLB in an ORCA 2C series FPGA, which provides similar routing resources to that of the Xilinx 4000 series FPGA [61]. Horizontal and vertical busses are denoted by h and v , respectively. The suffixes $x1$, $x4$, xH , and xL indicate wire segments that extend across 1 PLB, 4 PLBs, half the PLB array, and the full length of PLB array, respectively, before encountering a breakpoint CIP. The direct busses provide connections between adjacent PLBs; we denote the four direct busses as dn , ds , de , and dw denoting directions north, south, east, and west, respectively. For every PLB there are two sets of vertical $x1$ busses and two sets of horizontal $x1$ busses, designated $vx1w$, $vx1e$, $hx1n$, and $hx1s$. Not visible in Figure 27 is the rotation undergone by the wires of the $x4$ and xL busses as they pass by each PLB in the array. Many CIPs are available to establish different connections among the wire segments. The diamond symbol of a breakpoint CIP on a 4-bit bus represents a group of 4 individual breakpoint CIPs. The circle denoting a crosspoint CIP at the intersection of a vertical 4-bit bus with an horizontal 4-bit bus represents a group of 4 individual crosspoint CIPs between corresponding wires in the two busses, as illustrated at the upper right side of Figure 27a. The square at the intersection of a 5-bit direct bus with a 4-bit $x1$ bus represents a more flexible matrix of crosspoint CIPs, illustrated at the upper left side of Figure 27a. The local routing associated with each PLB is illustrated in Figure 27b, which includes multiplexer CIPs at the inputs to the PLB and bidirectional buffers with connections to the vertical and horizontal xH and xL busses, as well as the four direct busses. Here the MUX CIPs select one of four busses for connection to the PLB inputs. Although the ORCA 2C15 FPGA

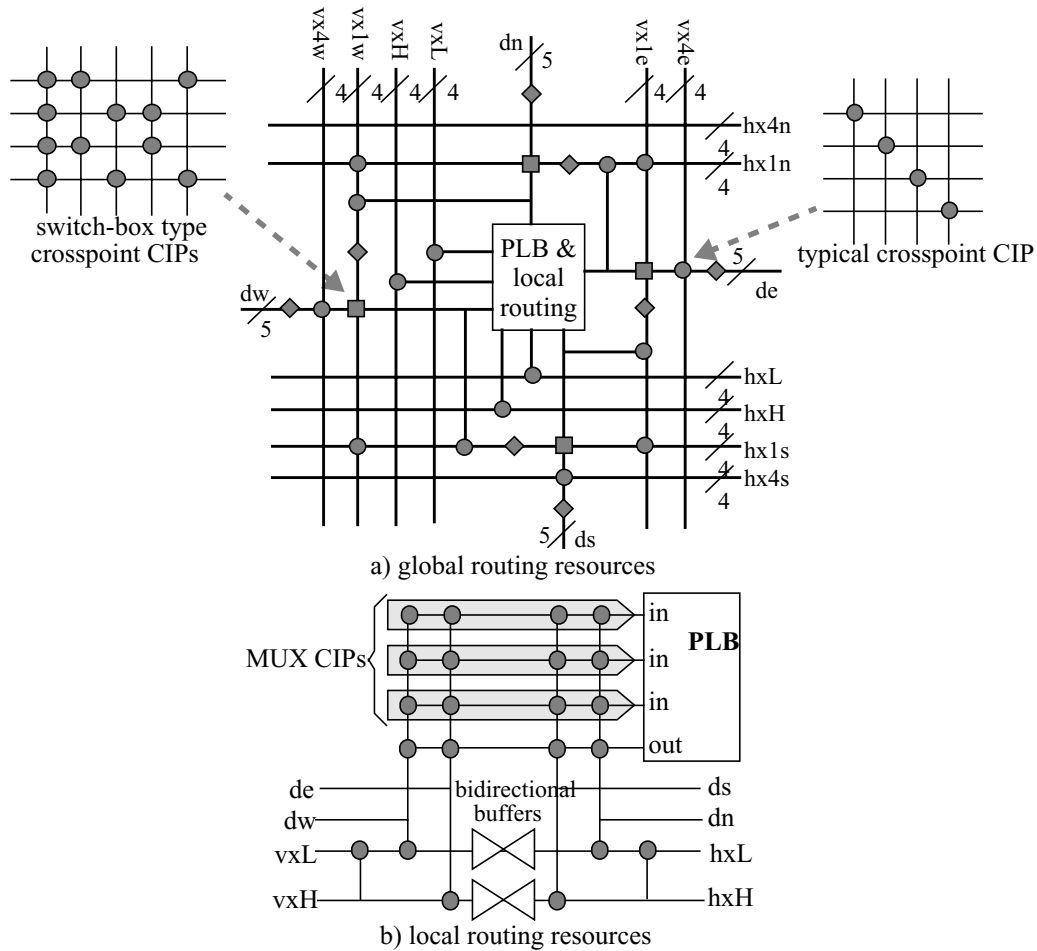


Figure 27. Single PLB View of Routing Resources

is a relatively small FPGA, it nevertheless contains over 25,000 wire segments and over 100,000 CIPs.

As mentioned above, the Xilinx 4000 series routing architecture is similar in many respects to that of the ORCA 2C series FPGA illustrated in Figure 27. For example, the Xilinx 4000 series FPGA has $x1$, $x2$, xL , and direct busses. The $x2$ busses also rotate at each PLB location. The array of CIPs illustrated at both the upper left and upper right corners of Figure 27a are present in the Xilinx FPGA. There are MUX CIPs in the Xilinx FPGA with the principle difference from the ORCA 2C series being that the MUX CIPs access any segment in a bus adjacent to the PLB in one direction (either north, south, east, or west), while OCRA 2C MUX CIPs access a subset of the segments in each north, south, east, and west bus. Xilinx 4000 series FPGAs also incorporate bidirectional buffers for high fanout signals or signals traveling long distances across the FPGA the same as in ORCA. Aside from the topological differences in their data book drawings of the interconnect structure, there is not much difference between ORCA and Xilinx interconnect structures. As a result, the BIST approach we describe here can be applied to either FPGA as well as any similar interconnect architecture.

4.6.1 Faults in FPGA Interconnect

The fault model we consider is typical for FPGA interconnect structures [45]: CIPs stuck-closed (stuck-on) and stuck-open (stuck-off), wires stuck at 0 or 1, open wires, and shorted wires. Detecting the CIP faults also detects stuck-at faults in the configuration memory bits that control the CIPs. For generality, we allow both wired-AND/wired-OR and dominant bridging fault models as possible behavior for shorts. A stuck-closed CIP creates a short between its two wires. We assume that no detailed layout information is available regarding adjacency relations between segments. Instead, we use only rough physical data (available in FPGA data books) to determine bunches of wires, where a bunch is a group of wires that *may* have pair-wise shorts, but not every wire is necessarily adjacent with every other wire in the bunch, and wires in different bunches are not adjacent. For example, all the vertical wires located between two adjacent PLB columns in Figure 27a could be treated as one bunch, even if not all shorts may be physically feasible. This treatment makes our method layout independent and allows us to ignore the bus rotations, which make the adjacency relations among the wires of the same bunch change along a row or a column of the FPGA. Although a short between a vertical and an horizontal segment is not physically possible, a crosspoint CIP stuck-on has an equivalent effect.

To detect the faults described above, the applied tests must check that every wire segment and CIP is able to transmit both a 0 and a 1, and that every pair of wire segments that may be shorted can transmit both (0,1) and (1,0) values. A test that applies 0 and 1 values at one end of a wire and expects the same values at the other end, detects any stuck-open fault in any type of closed CIP (crosspoint, breakpoint, or multiplexer) along the wire, and also any stuck-at fault affecting any wire segment. Stuck-on CIP faults require that opposite logic values be applied to both wire segments associated with an open CIP, so that a stuck-on fault will yield an incorrect logic value on one of the two wire segments. Since only the value on the wire segment which is part of a WUT is observed, all four combinations of values must be applied to the two wire segments separated by the open CIP, to cover both the AND and the OR types of short.

A multiplexer CIP requires one test configuration for each one of its inputs. Both 0 and 1 are applied to the selected input, while opposite values must be applied to the non-selected inputs. This test detects the stuck-off fault in the closed CIP that connects the selected input to the output wire, and the stuck-on faults in the open CIPs corresponding to the unselected inputs. This also provide a complete test for the selection logic of the multiplexer [57]. Testing MUX CIPs is similar to the invisible logic problem [56, 2]. The problem arises because FPGA CAD tools generate the configuration bitstream based on the user model, which will never include the functionally inactive subcircuits, or inactive inputs to MUX CIPs in this case. The result is that the testing of the MUX CIPs may not be complete. To ensure that all interconnect is properly tested, the routing must be done semimanually, without the typical FPGA CAD tools, such that opposite logic values can be routed to the inactive inputs of the MUX.

4.6.2 BISTER for FPGA Interconnect

The architecture of a single BISTER within a STAR is illustrated in Figure 28 and consists of a counter-based TPG, two groups of WUTs, and one or more comparison-based ORAs. A WUT may be composed of several wire segments connected by closed CIPs. To check local intercon-

nect, WUTs may also go through PLBs configured as identity functions (passing their inputs directly to their outputs). In the illustration in Figure 28, the WUTs are shown by bold lines, and the activated (closed) CIPs are shown in gray, while the open CIPs are white. The first group of WUTs connects the segments P , Q , R , and S , while the second group connects X , Y , Z , V , U , and W . Since Y and V connect to inputs of PLBs, these PLBs are configured as identity functions that just pass data from inputs to outputs. Note that without involving PLBs in configuring WUTs, one cannot achieve a complete interconnect test. The TPG applies exhaustive patterns to the WUTs, that is, all possible 2^n test patterns to every group of n WUTs. While walking patterns (walking a 1 through a field of 0s and a 0 through a field of 1s) are known to provide a complete set of tests [34], generating exhaustive patterns with a n -bit counter requires less PLBs than generating the two n -bit walking patterns, which are already contained in the exhaustive set. For large n , we divide the n WUTs into groups of $k < n$ wires, and apply exhaustive patterns to one group at a time, while setting the other $n-k$ wires to constant values, so that eventually all required pairs of values will be applied. The open CIPs which isolate the WUTs from the rest of the interconnect must be tested for stuck-closed faults. To achieve this, the TPG also controls any wire that may become shorted to some WUT (A , B , C , D , and E in Figure 28) such that when the TPG drives a 0(1) on the WUTs, it must also set A , B , C , D , and E to 1(0) at least once during the test of the WUTs.

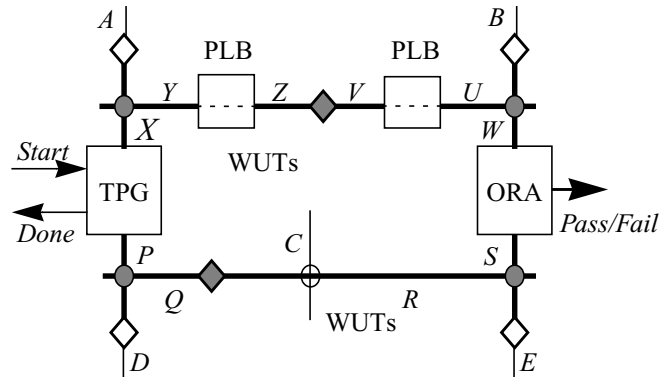


Figure 28. Basic BIST Structure for Interconnect

A swapper cell may be needed as part of the BISTER to align the compared signals before reaching the ORA. The need for the swapper cell arises due to the nature of the logic equations implemented in the LUTs of the ORA, routing limitations on the inputs to the PLB implementing the ORA, and bus rotations in the WUTs such as the $x4$ and xL busses in ORCA or the $x2$ busses in Xilinx 4000 FPGAs. As illustrated in Figure 29, the swapper is simply a combinational function

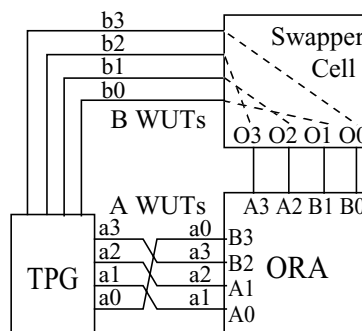


Figure 29. Bus Realignment

programmed to map input signals to output signals to achieve the desired alignment of the WUTs at the ORA inputs, so that signals $a1$ and $a2$ are compared with $b1$ and $b2$ in the LUT fed by the PLB inputs A0-A3 while signals $a0$ and $a3$ are compared with signals $b0$ and $b3$ in the LUT fed by the PLB inputs B0-B3.

We use comparison-based ORAs, which do not suffer from aliasing problems that affect methods based on compression, such as signature analysis. Figure 30 illustrates an ORA comparing wires from each set of WUTs being tested by the BISTER. The FF stores the result of the comparison, and the feedback loop latches the first mismatch in the FF. The *Pass/Fail* result FFs of all ORAs are connected to form the scan chain [18]. In every test phase, the scan chain is also tested, to assure the integrity of the test results. Here an ORA compares the patterns propagating on two groups of WUTs, which should be identical if their propagation is not affected by faults. The only condition that will make this scheme fail would be a multiple fault consisting of identical (or equivalent) faults in the two compared groups (for example, both groups having a short between the third and fourth wires). Although this is not very likely, we overcome the problem by testing each set of WUTs twice and comparing a given set of WUTs with two different sets of WUTs (a different set each time it is tested).

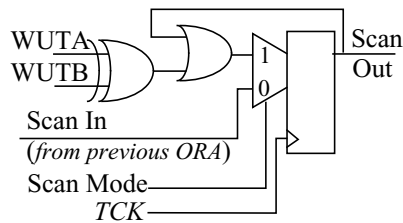


Figure 30. Integrated ORA/scan cell

Testing the crosspoint CIPs connecting global horizontal and vertical routing busses is a difficult problem, since most global horizontal routing resources in V-STAR, and similarly, most global vertical routing resources in H-STAR, may be used by the system signals connecting the two working areas separated by the STAR. Our solution is to construct a BISTER that spans both V-STAR and H-STAR, as illustrated in Figure 31. To test all these CIPs would require such a BISTER for every possible pair of positions of H-STAR and V-STAR, which means $O(N^2)$ configurations. To avoid such a significant increase in the fault latency, we adopt the following compromise: we test only a subset of these CIPs on every full horizontal sweep, but making sure that the subset is different on every sweep. That is, we keep the H-STAR fixed and we test only the subset of CIPs in the current intersection of H-STAR and V-STAR, but for every full horizon-

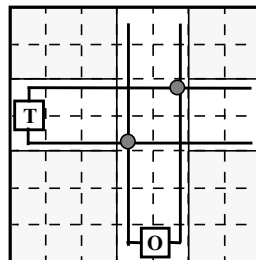


Figure 31. Testing global crosspoint CIPs

tal sweep of the V-STAR, the position of H-STAR is different. This progression of testing and roving of the STARs is illustrated in Figure 32 for an FPGA consisting of an 8 by 8 array of PLBs.

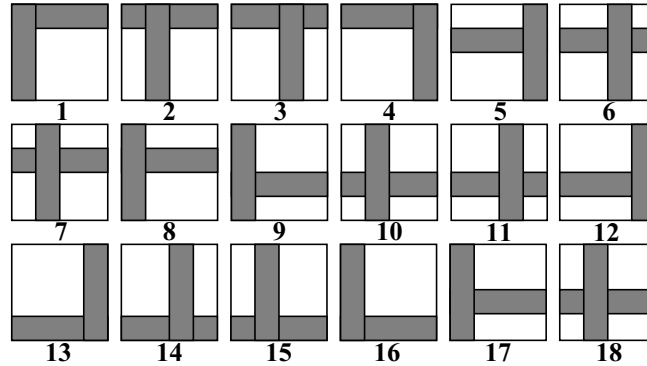


Figure 32. Roving Order for Testing Interconnect

The V-STAR and H-STAR are used, respectively, to test vertical and horizontal routing resources. In addition, several configurations are needed to test different bunches of WUTs within a STAR. Once a BISTER detects a failure in interconnect, diagnostic configurations are downloaded to repeatedly subdivide the suspected WUTs into two subsets that will be included in new BISTERS to be compared with known fault-free WUTs. This adaptive diagnosis process is guaranteed to locate any fault detected.

Next, we describe the basic implementation of the various BISTER test sessions for on-line testing the different types of routing resources and CIPs in the interconnect network. The entire set of test configurations is organized as a sequence of five test sessions, where we define a test session as a group of test phases targeting the same types of faults. Table 4 summarizes the five test sessions for testing interconnect faults in FPGA. Some of these test sessions may not be required for some FPGAs (depending on the architecture of the interconnect network). Similarly, the number of test phases associated with each test session will vary due to the specific differences in arrangements of CIPs as well as in the bus structures, such as the number of wires in each bus.

Table 4: Summary of interconnect BIST test sessions

Session	Target Faults
1	shorts, opens, and some CIP faults in global busses
2	CIP faults in x1 busses, shorts & opens in local routing
3	faults in crosspoint CIPs between global & local busses
4	multiplexer CIP faults
5	faults in crosspoint CIPs between global busses

The first test session tests all the global busses (such as those busses illustrated in Figure 27a) for shorts and opens. In addition, these configurations test all breakpoint CIPs along the global busses for stuck-off faults. The BISTER architecture for testing vertical busses is illustrated in Figure 33, where the bold lines represent busses under test. Two PLBs (denoted by T) form a $2n$ -bit counter-based TPG. The lower n -bits are used to drive an exhaustive set of test patterns on the A and B WUT busses. The upper n -bits are used to drive exhaustive test patterns on the two sets of adjacent busses (denoted by the dashed lines). The complete set of $2n$ -bit exhaustive patterns can then detect shorts between wire segments within the WUTs as well as between the adjacent busses

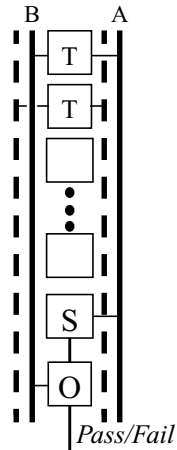


Figure 33. BISTER for Vertical Busses

regardless of any bus rotations within any bus, since the $2n$ -bit count sequence ensures opposite logic values will be on adjacent wire segments. Two BISTERS are placed within the STAR such that the A WUTs are sandwiched between two dashed busses. Multiple test phases are needed to test all the vertical busses within the V-STAR. For example, in the first test phase the BISTER is configured as shown in Figure 33, while during the second test phase, the busses are reversed so that the busses denoted by dashed lines are under test. Subsequent test phases have the WUTs moving across the global busses associated with a given column of PLBs until all vertical busses have been tested. A similar set of test phases are used to test the horizontal busses in the H-STAR. The ORA (denoted by O) compares the A WUTs and B WUTs driven by the same TPG cell. The PLBs denoted by S are swapper cells which may or may not be needed depending on the particular bus and/or PLB architecture.

In test session 2, we test most of the global-to-local $x1$ crosspoint CIPs for stuck-off faults and $x1$ breakpoint CIPs on the direct busses for stuck-on faults. Note that $x1$ routing resources are common to all FPGA interconnect architectures and provide the primary routing resources for local, low-fanout nets. The test phases in test session 2 use the basic testing arrangement shown in Figure 34 where the busses are interconnected via crosspoint CIPs such that they crisscross through the local routing resources of the PLB, instead of passing straight down on a single global bus. As illustrated in Figure 34, one set of WUTs (grey line) goes through the LUTs of the PLB, while the other set of WUTs (dashed line) goes through the bidirectional buffers such that the bidirectional buffers are also completely tested in both directions during this test session. Breakpoint CIPs are tested for stuck-on faults by the crisscross routing since opposite logic values will be applied to both sides of the breakpoint CIPs as illustrated in Figure 34. Note that no global horizontal routing resources are used in this V-STAR BISTER to facilitate interconnection of the partitioned system function that is still in operations outside of the STARS. As in the case of test session 1, there are two BISTERS contained in the STAR. Since routing through the local interconnect resources is required during this test session, local routing resources can also be tested during this test session, including the bidirectional buffers associated with each PLB which must be tested in both directions. Multiple test phases are typically required for this test session for both the V-STAR and the H-STAR as a result of the different types and numbers of crosspoint CIPs on these busses.

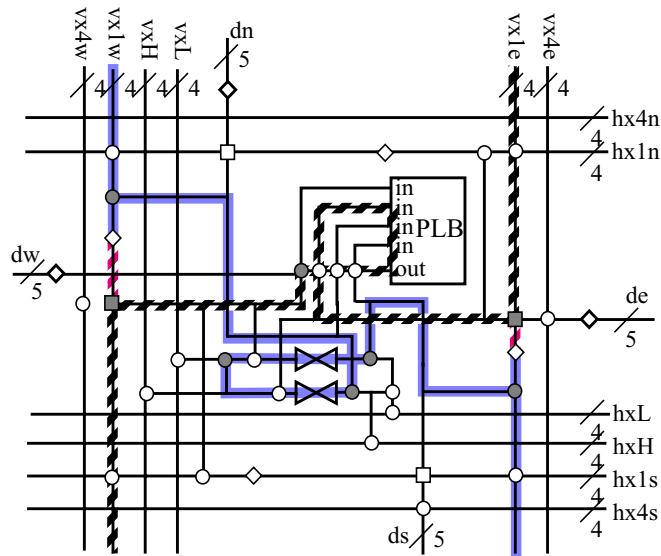


Figure 34. Example of Test Phases for Local Routing

In test session 3, we test the crosspoint CIPs between direct busses and global busses. The BISTER structure used for these test phases is illustrated in Figure 35. Because of the long spans of these global busses across multiple PLBs, we must use multiple ORAs. Test patterns are supplied on the busses and arrive at ORA cells through the crosspoint CIPs being tested in these configurations. Note that we have already tested these busses for shorts (in test session 1) and now we are only testing for crosspoint CIPs stuck-off.

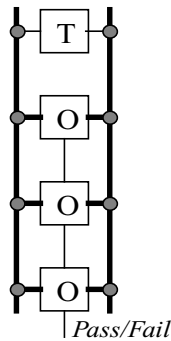


Figure 35. Testing Crosspoint CIPs between Direct and Global Busses

In test session 4, we test the multiplexer CIPs at the input to the PLB for stuck-on and stuck-off faults. Signals passing through the PLB will detect multiplexer CIP stuck-off faults for those CIPs selected to drive the PLB. Stuck-on faults in the unselected inputs of the multiplexer CIPs are detected by supplying the opposite logic values of those being passed to the PLB on the selected inputs. An example of these test phases is shown in Figure 36, where the signals passing through the PLB are reordered on the bus to the MUX CIP in order to supply the opposite logic value to the unselected inputs to the MUX CIP. In addition to testing the local routing wire segments, crosspoint CIPs, and bidirectional buffers during session 2, it is possible to also test some of the MUX CIPs on those WUTs that pass through the LUTs of the PLB. However, test session 2 is perhaps the most complicated and difficult to implement of all five test sessions and, as a result, maintaining the MUX CIP test phases as a separate test session has advantages. For those FPGAs where

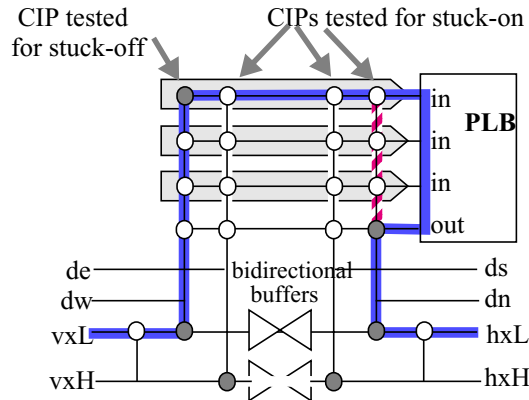


Figure 36. Example of MUX CIP Test Phases

the MUX CIPs are independent of vertical and horizontal global routing resources, all test phases of test session 4 can be performed with either the V-STAR or the H-STAR alone.

Both the V-STAR and H-STAR are needed to test the crosspoint CIPs between the global busses in the FPGA. This is because vertical global routing resources passing through the H-STAR and horizontal routing resources passing through the V-STAR are reserved for interconnecting the partitioned working areas that are separated by the STARs. Since testing these crosspoint CIPs requires both horizontal and vertical global routing resources, the testing can take place only at the intersection of the H-STAR and V-STAR. The four test phases for testing the crosspoint CIPs on a given set of global-to-global busses are illustrated in Figure 37. Two PLBs residing in the H-STAR are used as a $2n$ -bit counter-based TPG with one n -bit field supplying test patterns to the activated crosspoint CIPs, and with the other n -bit field providing test patterns to the global busses bordering the nonactive crosspoint CIPs. In this way the activated CIPs are tested for stuck-off faults, while the nonactivated CIPs are tested for stuck-on faults. In Figure 37a and b, the upper half of crosspoint CIPs at the STAR intersection are tested for stuck-off faults while the lower half of the crosspoint CIPs are tested for stuck-on faults. The testing roles are reversed in Figure 37c and d. Two PLBs in the V-STAR are used as ORAs with routing resources associated with the 2-by-2 array of PLBs at the intersection of the two STARs forming the set of CIPs under test during this test session. This basic arrangement of PLBs is flipped and/or rotated in order to test the four corners of the FPGA.

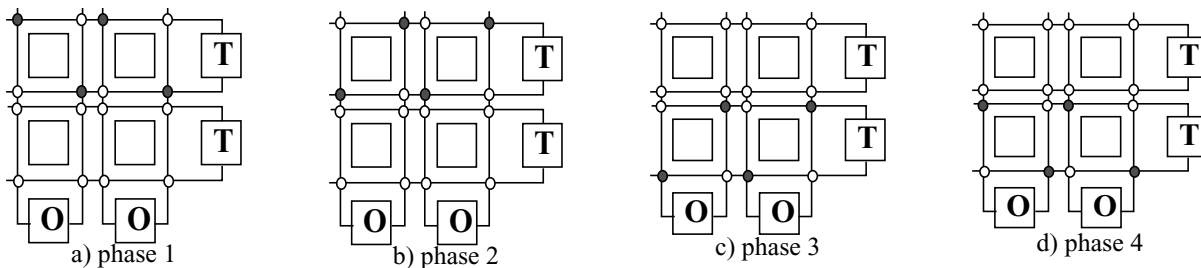


Figure 37. Testing global crosspoint CIPs

The on-line interconnect BIST approach was implemented for the ORCA 2C and 2CA series FPGAs. The number of test phases required for each test session is summarized in Table 5. These five test sessions perform a complete test of all routing segments and their interconnecting CIPs

within a V-STAR in a total of 25 configurations and within an H-STAR in a total of 18 configurations. This number is less than the number of test phases required for a complete on-line PLB BIST [3] (104 for the ORCA 2C series and 136 for the ORCA 2CA series FPGAs). However, the 39 total test phases for our on-line interconnect BIST do not provide the same diagnostic resolution as the PLB testing within a STAR. Therefore, we have developed additional diagnostic test phases for interconnect fault location and identification described in the next subsection.

Table 5: Summary of interconnect BIST test sessions and phases

Test Session	Target Faults	No. of VSTAR Phases	No. of HSTAR Phases
1	shorts, opens, and some CIP faults in global busses	7	7
2	CIP faults in x1 busses, shorts & opens in local routing	4	4
3	faults in crosspoint CIPs between global & local busses	3	3
4	multiplexer CIP faults	7	0
5	faults in crosspoint CIPs between global busses	4	

The on-line interconnect test sessions and their associated test phases were implemented and applied to three known-faulty ORCA 2C15A FPGAs from Lucent Technologies. Two of the FPGAs (Chip 1 and Chip 2) exhibited routing faults consistent with the results in [57], while the third faulty FPGA (Chip 3) was found to contain only the previously described logic fault (one faulty PLB with faulty FFs). The interconnect testing results are summarized in Table 6.

Table 6: Summary of Interconnect BIST Test Results on Faulty Chips

Chip	STAR	STAR Position	Test Session	Test Phases
1	V-STAR	4	2	1
			3	3
			4	1, 2
	H-STAR	5	1	5, 6
2			1, 3	
2	V-STAR	6	1	7
			2	1, 2
	H-STAR	1	2	1, 2, 3, 4
			3	2
	H-STAR	3	1	7
			3	1
3	V-STAR	9	3	2, 3
			4	4, 5, 6, 7
	H-STAR	2	3	1, 2, 3

While these particular test sessions were not specifically designed for diagnosis, the on-line BIST configurations do provide a significant level of diagnostic resolution that is much higher than that obtained in our original off-line routing BIST [57]. The diagnostic resolution that can be obtained directly from the testing results is illustrated in Figure 38 for Chips 1 and 2. While the exact location of the faulty interconnect cannot be determined by the testing results, it can be seen from the

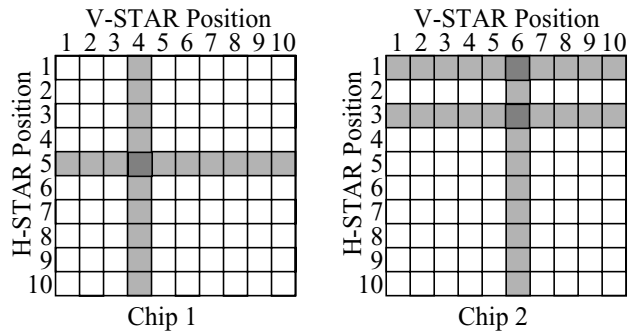


Figure 38. Testing Results from Faulty ORCA 2C15s

failing areas of the faulty FPGAs in Figure 38 that the suspect regions can be quickly narrowed down to 19 percent of the total chip area in Chip 1 and 28 percent in Chip 2.

4.7 Interconnect Diagnosis

Once a fault is detected, we must identify the fault and its location to a sufficient level of accuracy to facilitate efficient reconfiguration around the fault for fault tolerant applications. To achieve the necessary diagnostic resolution, we use a number of techniques described in this section. We begin by analyzing the results of the on-line BIST to assist in narrowing the search space of the fault(s). For example, when a STAR contains multiple BISTERS, a failing result from one of those BISTERS provides an initial target for the application of additional diagnostic configurations. Since all WUTs are tested twice and compared to two other sets of WUTs, we can determine in which WUT (A or B) was faulty through analysis of the ORA failure indications. This is illustrated in Figure 39a where during the second test phase the middle WUTs are swapped between TPGs. For faults in only one set of WUTs, we can identify which WUT contains faults by looking at whether the failures moved to the other ORA when the set of WUTs were swapped which will uniquely identify the set of WUTs with faults. When we have multiple faulty WUTs that prevent a unique identification of the correct sets of faulty WUTs, we apply additional diagnostic phases that compare the suspected sets of WUTs with other sets of WUTs (different from the ones initially used).

To identify the faulty region in a set of WUTs, we add multiple ORAs along the run and reverse the direction of the test pattern flow through the WUTs during a second test phase. This is illustrated in Figure 39b. Another technique is to subdivide the BISTER into smaller BISTERS and retest to identify the faulty region of the WUTs as shown in Figure 39c. This divide-and-conquer method can be repeated to obtain high diagnostic resolution.

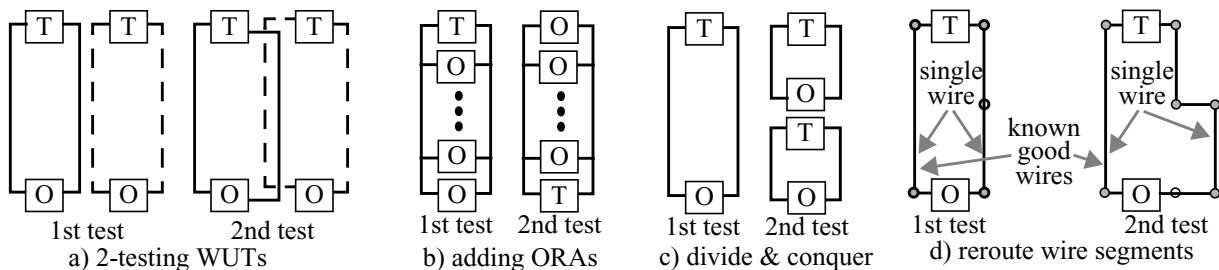


Figure 39. Diagnostic Configurations

Finally, to identify the faulty wire in a set of WUTs, we reduce the set of WUTs to a single wire under test, as illustrated in Figure 39d. After cycling through all of the wires that made up the original set of WUTs, we determine the faulty segment or CIP by rerouting to eliminate a segment at a time during the subsequent tests. An alternative to identifying the faulty wire from the set of WUTs is to rotate the test pattern bits (from the TPG) on the wires in the suspect WUT while keeping the test pattern assignments constant on the known good set of WUTs. This requires independent ORAs for each pair of wires being compared but this approach aids in obtaining good diagnostic resolution without increasing the number of diagnostic test phases.

The definition of a segment depends on the type of its fault. For opens, a segment is a wire bounded by two CIPs with no other CIPs in between. For example, in Figure 40, B_1 and B_2 are breakpoints, and X_1 and X_2 are crosspoints. The heavy line denotes a signal net routed through X_1 and X_2 . The dashed line denote segments that are connected to the same signal net, although they don't serve to connect it to other segments (note that a crosspoint CIP does not break continuity along the horizontal or vertical segments). Each one of the three segments B_1X_1 , X_1X_2 , and X_2B_2 may have an open fault. For shorts, however, a segment is a wire bounded by breakpoint CIPs. For example, a short between B_1X_1 and another segment Z will affect the entire segment B_1B_2 , which will be treated as a single faulty segment.

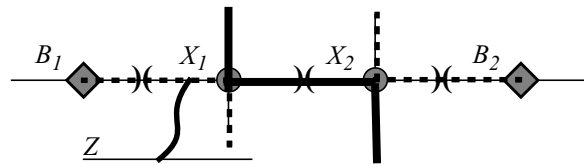


Figure 40. Wire Segments and Faults

Let P and Q be two faulty segments. To distinguish between P and Q being both open and P and Q being shorted to each other, we configure a TPG to generate the four possible patterns (00, 01, 10, 11) for the two suspects, and record the cases that fail. Any mismatch for the 00 or 11 patterns would indicate an open, while any mismatch for the 01 or 10 patterns would indicate a short. In case of an open, we may initially only know the open segment between two breakpoints CIPs. However, we can diagnose further to determine which internal segment is open by using the strategy illustrated in Figure 39d. Here we reroute portions of the segment based on the crosspoint CIP locations along that segment in order to determine the relative location(s) of the open(s). For example, in Figure 40 we can diagnose each one of the three segments B_1X_1 , X_1X_2 , and X_2B_2 independently by the TPG to ORA connections. In this example, the dotted lines would extend beyond the breakpoint CIPs and represent two different configurations of the BIST structure (one for diagnosing B_1X_1 and the other for X_2B_2), while the third configuration would be represented by the bold line.

Application of the additional diagnostic test configurations were applied to both FPGAs. From these diagnostic configurations, we identified a single routing fault in Chip 1 at the intersection of the two STARS. Furthermore, we were able to determine that the fault is a short in a local routing segment in row 10 column 8. In addition, we were able to determine that the faulty segment is not shorted to another interconnect segment but is possibly shorted to either the logic or the configu-

ration memory. This is an important finding since previous work in FPGA logic and interconnect testing and diagnosis has assumed that faults are either in the logic or in the interconnect resources such that the two domains can be considered independently. However, this actual manufacturing defect indicates that the FPGA diagnosis problem is much more complex than previously thought.

It is obvious from the BIST results for Chip 2 that there are multiple routing faults (at least two faults). Application of the diagnostic test configurations confirmed that there are indeed two faults. The first fault is a short in a local routing segment in row 1 column 12 (again at the intersection of the two failing STARs). Like the short in Chip 1, the diagnostic tests indicate that this short in Chip 2 is also shorted to either the logic or the configuration memory. Therefore, we conducted further testing to identify what these segments were actually shorted to, in other words, the other half of the short. We were unable to conclusively determine a short to either logic or to the configuration memory, but we found another possible scenario to be a resistive short to the power bus. The second fault in Chip 2 is in the horizontal global routing resources in row 5 (specifically the hx4s bus in Figure 27) and does not effect V-STAR testing. This fault is a short between three of the wires in the four-wire bus and the short is located between columns 6 and 8.

4.8 Fault Injection Emulation

Since it is difficult to obtain actual faulty FPGA (particularly with logic faults as opposed to routing faults) we have developed the capability to insert faults into specific PLBs and specific CIPs via reconfiguration of the PLBs and CIPs. This provides a capability similar to physical fault insertion, and gives us the ability to emulate both random and clustered faults within the FPGA. We have used this method for verifying both our fault detection and diagnosis approach, and the fault tolerance mechanism.

The basic idea is similar to the fault insertion based on fault masks used in a parallel fault simulation, but in this case we operate on the configuration bit files just prior to each download of the system and BISTER tile configurations. In parallel fault simulation, each bit of a computer word is used to simulate a faulty circuit during fault simulation. As a result, a 16-bit or 32-bit word can be used to simulate 16 or 32 faulty circuits in parallel. This requires the following set of logical operations be performed on each logic value at each node in the circuit netlist during parallel fault simulation [1]:

$$\mathbf{Z} = \mathbf{V} \cdot \bar{\mathbf{F}} + \mathbf{F} \cdot \mathbf{S} \quad (6)$$

where ‘+’ is the logical OR operator and ‘.’ is the logical AND operation. In this expression, \mathbf{V} represents the logic value at a given node and would be contained in an N -bit computer word to represent the values for N faulty circuits in parallel. The fault mask, \mathbf{F} , contains a ‘1’ in each bit position for which a faulty circuit being simulated contains a fault at the given node, and \mathbf{S} gives the stuck-at values (either stuck-at-0 or stuck-at-1) that are to be injected for those faults active at that node in the parallel fault simulation. Therefore, the fault mask \mathbf{F} acts as multiplexing control values which let the normal circuit logic value of \mathbf{V} pass to the node value \mathbf{Z} in the case of those parallel faulty circuits where no fault exists at the given node; otherwise the stuck-at logic value replaces the normal circuit logic value [1].

Using this basic idea of the fault mask, our fault injection emulator performs a similar operation on the configuration file to be downloaded into the FPGA. We start with the original configuration file, C , to be downloaded into the FPGA to perform the desired system function. This file is the automatically generated by the CAD tools that support the particular SRAM-based FPGA(s) being used in the system. We then create a fault mask file, F , and a stuck-at value file, S , to complete the fault injection operation as follows:

$$D = C \cdot \bar{F} + F \cdot S \quad (7)$$

where D is the resultant configuration file with injected faults to be downloaded into the FPGA. The size of the fault mask, F , and the stuck-at-values file, S , is the same as that of the original configuration file, C , and the resultant download configuration file, D . A logic 1 in any given bit position in the fault mask will replace the original configuration data bit with the corresponding value specified in the stuck-at value file in the resultant download configuration file.

In our method, the generation of a new download file takes place using the operation described above prior to each download to the FPGA. As a result, faults can be changed in the fault mask and stuck-at value files at any time. This facilitates the emulation of permanent faults (by maintaining those faults in the fault mask and stuck-at value files) or transient faults (by removing those faults from the fault mask after some period of time). In addition, there can be any number of faults specified in the fault mask with the location of the faults being randomly distributed or clustered. Since the fault injection operation takes place prior to each download, the fault mask can be modified prior to each download to add or subtract faults temporally.

The faults that can be emulated by this method include stuck-at faults in the logic resources (LUT and FFs) or functional faults (such as a FF that acts as a latch or it is activated on the opposite edge of the clock). The faults that can be emulated in the programmable interconnect network include wires shorted or open in combinations that can be made by the appropriate CIPs turned on or off in the faulty mode of operation. It is important to note that no FPGA logic or routing resources are needed to emulate these faults since only manipulation of the configuration memory bits (via manipulation of the configuration file) is required to emulate the faults. However, we have encountered a few limitations with this method. For example, the LUTs in most FPGAs can also be configured to operate as small RAMs (larger RAMs can be constructed from multiple LUTs). While permanent stuck-at bits in the LUT can be emulated with our technique, we cannot emulate permanent stuck-at bits when the LUT is in a RAM mode of operation because the system function can overwrite our stuck-at value downloaded into the FPGA once system operation takes over.

The difficulty of using this fault injection emulation method is determining the configuration memory bit mapping for fault emulation. In other words, what does a given bit in the configuration file control within the FPGA architecture? FPGA manufacturers have been very reluctant to release their configuration memory bit maps on the grounds that it may reveal proprietary architectural information to competitors. This argument turns out to be rather weak since the CAD tools provided by the FPGA manufacturers typically contain all the features needed to reverse engineer the configuration memory bit mapping. For example, using the graphical architectural editor (such as EPIC in the ORCA Foundry tool suite), one can activate any given CIP in the FPGA and subsequently generate the configuration file to determine the location of the bit con-

trolling that particular CIP in the FPGA. While this is a somewhat tedious process, the regularity, symmetry, and repetitive structure of the FPGA simplifies the process by allowing the determination of a given configuration memory bit for a given PLB to be translated to any PLB via the appropriate offset in the configuration memory address.

Most SRAM-based FPGAs have their configuration memory partitioned into vertical slices (also referred to as frames), each with a unique address. Within the frame, configuration bits repeat down the column to control the PLBs and routing CIPs within that frame. Once a set of frames have been mapped to establish the FPGA logic and routing resources that the bits in those frames control, the subsequent set of frames for the next column of PLBs will have identical bit maps with the only change being the corresponding frame address offset. The exception to this is the inter-quadrant routing that is found in some FPGAs (this contains specialized routing resources for global functions such as clocks); this results in an additional offset, depending on which of the four quadrants the fault is to be injected.

Once the bit and frame locations of all desired faults to be emulated have been determined, faults can be injected in these locations by activating the correct fault mask bits in F and specifying the associated stuck-at value in S . Since these are text files, they can be easily modified manually or via a dedicated program that allows faults to be added or removed from the text file. In our fault injection system, we create a text file that lists faults to be emulated using a meaningful naming convention. As many faults as desired can be listed in this text file. A specialized program called Faulting then takes this information and converts the fault location data (the row, column, type, and particular element data) to the appropriate bit location within the frames of the fault mask file by inserting a 1 in the appropriate location for that fault. Similarly, the appropriate bit in the stuck-at value file is manipulated by FaultGen for each fault in the fault list. This process continues until all faults from the fault list have been entered into the fault mask and stuck-at value files. Once the fault and stuck-at value files have been constructed, the faults can be injected into the FPGA on the next download.

The fault list can be changed as a function of time and the FPGA can be reconfigured (by re-downloading) to inject the new set of faults. This allows for the addition of faults to the system or the removal of faults for transient fault emulation. Programs can be used to create the fault list for random or clustered spatial or temporal fault distributions. This facilitates efficient and effective evaluation of testing and diagnosis techniques as well as fault tolerant approaches. For demonstrations of our BIST and BIST-based diagnosis, as well as our logic and interconnect FT techniques, we have included the ability to interactively specify the faults to be emulated using a graphical user interface (GUI), shown in Figure 41, where the background shows the 20 by 20 array of PLBs for an ORCA 2C15 FGPA. This GUI allows injection or removal of faults at any time while the FPGA is operating. A row and column location is selected by clicking on an area of the PLB array.

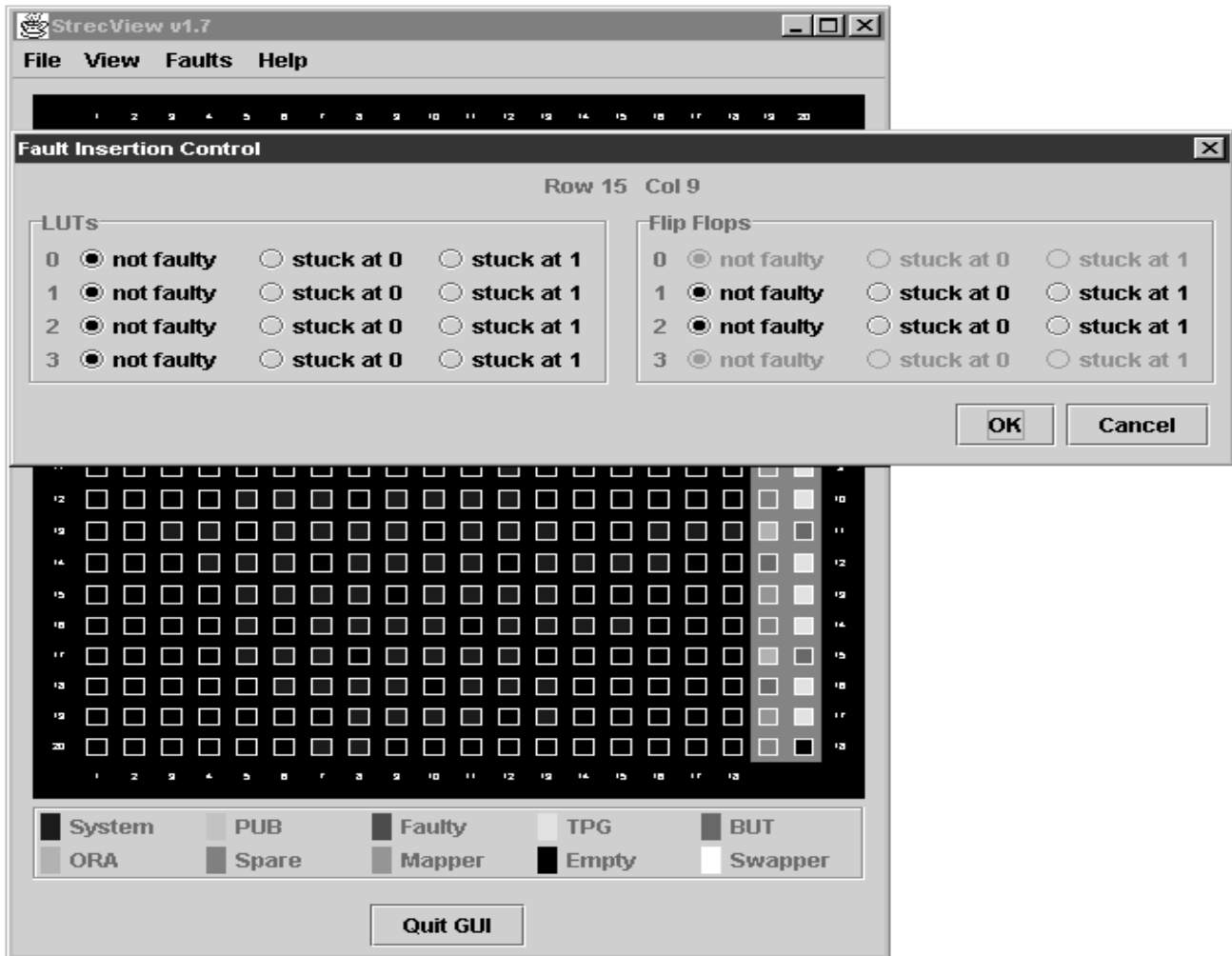


Figure 41. TREC and Fault Injection Emulation GUI

5. Fault-Tolerant Techniques

5.1 Basic Features

In most conventional FT methods, faults are detected within the working part of the system, and they must be located and bypassed as fast as possible so that normal system operation can resume as soon as possible. For our roving STARs approach, we divide the FPGA into two distinct parts: the STAR areas where all testing takes place, and the working area where the system function operates. A consequence of our roving STARs strategy is that the faults we detect are always in a STAR area of the FPGA, and thus they do not affect the working area of the FPGA. So, we don't need to interrupt the normal operation to replace the faulty resource by a fault-free one, since the former is not performing any system function when the fault is detected. Once a fault is detected, we first set a flag to notify the operating system that a rollback may be required if it is determined that mission-critical data may have been corrupted since the last time the faulty element was tested. Another important difference from conventional techniques is that fault diagnosis and FT reconfiguration do not have severe real-time constraints. Since normal system operation continues in the working area, we can allow much more time for accurate diagnosis and for computing any required fault-bypassing configurations, compared with an approach in which normal operation is interrupted for diagnosis and reconfiguration.

Fault tolerance is performed after faults are detected and diagnosed (located). Reconfiguration for logic fault tolerance and reconfiguration for interconnect fault tolerance are distinct tasks. First we determine if the system function can continue to work correctly in the presence of the located fault(s). In many situations this is possible and then no reconfiguration is needed. If a fault does affect the system function, we determine alternate configurations that avoid the faulty resource.

Besides being applied to avoid faults, reconfiguration can be used to reduce the performance degradation caused by fault avoidance. In other FT approaches, the system clock speed must be set to accommodate the worst-case FT configuration that is possible. In the event that the worst case never occurs, the system is penalized with a slower than necessary clock speed. We have introduced the concept of an adaptive system clock implemented by a clock generator whose programmable period can be set by the TREC (via the control interface between the TREC and the FPGA). The initial clock frequency is set to the maximum value allowed in the fault-free circuit. If the circuit critical path changes as a result of FT reconfiguration, TREC adjusts the clock rate based on post-routing timing analysis. The timing analysis is run incrementally only for the signal nets affected by reconfiguration. Without such an adjustment the clock must be slow enough to work for any possible FT reconfiguration. In contrast, adjusting a programmable clock period will introduce timing penalties only when required, as a result of new faults. This contributes to a more graceful system degradation as faults occur.

In the following sections, we first review related research, and then describe the details of our approach to logic and interconnect fault tolerance. We introduce our three-level approach that we apply during both logic and interconnect fault tolerance. Then in later subsections, we describe specific details and unique features of both our logic and interconnect FT approaches.

5.2 Related Research

5.2.1 Logic Fault Tolerance

In their FT technique, Kumar et al. used adaptive customization to avoid defective sections and artificially enhance a programmable gate array's yield [30]. Their technique was strictly off-line and fault tolerance was achieved by bypassing faulty programmable resources at the time the circuit was initially mapped to the FPGA. They described a two-step process to achieve defect tolerance: test of an unprogrammed device to locate defective components and program the device to avoid the defective portions. They presented heuristics for adaptively programming an FPGA in the presence of faults.

Hatori et al. introduced redundancy as a FT method for FPGAs [23]. Their method used specialized selector circuitry to reconfigure FPGA circuits in the presence of faults. Similar to methods used for fault tolerance in SRAMs, an additional column of programmable logic blocks is added to the FPGA. In the event that a logic block was faulty, all functions mapped to the column where the faulty block was located were shifted toward the spare column. All subsequent columns of functions between the column with the faulty block and the spare column were also shifted a distance of one column toward the spare column. Their selector circuits required approximately 2 percent additional area overhead, and they required spare columns. Since they eliminated an entire column for one fault, their fault tolerance was low, and they estimated a performance degradation of 5 percent when compared to similar non-FT FPGAs.

Durand and Piquet proposed an FT FPGA architecture with the ability to repair itself in the presence of faults [8]. They used a special multiplexer with self-repair and self-diagnosis capabilities. Their special circuitry was capable of detecting logic failures during run-time, and the architecture had a limited capability to reconfigure itself dynamically. Their FT technique allocated two extra columns of cells, and in the event that a fault was detected, they bypassed the faulty column by shifting toward the spare column. Since the authors reported that more than complete logic redundancy was required for their method, this approach results in a large area overhead.

Narasimhan et al. developed a FT technique for FPGAs or wafer-scale integrated arrays [42, 43]. They use a pebble shift algorithm to reconfigure around faulty blocks. Their method is flexible. It is not limited to one fault per row, column, or tile. However, it must be performed off-line, and it requires the application cease execution for reconfiguration.

Howard and Tyrrell described concepts for increasing yield on FPGAs [24]. Their FT idea was to insert extra bypassing columns or rows into the FPGA. They looked at both removing an entire row (column) if it contained a fault and removing only the faulty block from the row. Their methods were strictly for yield enhancement and the faults were bypassed at the time the configuration for the FPGA was downloaded. They also introduced global long lines to reduce the performance hit associated with reconfiguration.

Kelly and Ivey used redundancy to detect faults in applications mapped to FPGAs [29]. Their FT technique used a shift method to reconfigure in the presence of faults. They incorporated a reconfiguration switch to reconfigure and used normal place and route tools for mapping circuits to

FPGAs. A step-based switch configuration algorithm was used to configure the special switches around faulty elements.

Cuddapah and Corba used Xilinx SRAM-based FPGAs to demonstrate the FT capabilities of FPGAs [7]. In their study, they randomly picked logic blocks to be faulty. Then they reconfigured the circuit around these faults using commercially available tools. The main contributions of their work were an algorithm to determine fault coverage of a design and a definition of the fault recovery rate for any given design implemented in an SRAM-based FPGAs. Additionally, they demonstrated that fault recovery was feasible on FPGAs by other than modular redundant methods.

Dutt and Hanchek et al. developed a method to increase FPGA yield [9, 19]. Their method uses node covering and reserved routing resources to replace the functionality of faulty cells. One row (column) of cells is reserved for spares. If a cell in any given column (row) goes bad, the functionality of all cells in the column (row) from the faulty cell to the spare cell is shifted toward the spare cell. Spare routing resources are used to eliminate overhead of rerouting the updated circuit placement. The main advantage of this method is that it is very fast. Since the spare resources have already been allocated to cover a limited number of faults, the reconfiguration time is linear with respect to the number of faults. There are several issues with this methodology. The most significant problem is the limitation of the fault coverage. The number of faulty cells that can be covered is limited to one in each column (row). If two faults were found in the same row, this method breaks down. There is also an associated resource overhead. For an N by N array of cells, at least N cells must be left unused. There is also a routing overhead caused by the reservation of routing resources. Additionally, when cells are moved there is an associated performance degradation associated with increasing the length and number of programmable interconnect resources.

Emmert and Bhatia developed a FT technique for incrementally reconfiguring FPGA mapped circuits around faulty programmable resources [11, 13]. They used mini-max grid matching, to match faulty PLB locations to unused spare PLB resources. Then they incorporated a shift methodology to shift PLB logic cell functions between the faulty PLB and its matched spare location toward the spare location. The shift method reduced the performance degradation caused by matching logic cell functions to spares that were spatially separated by large distances. They applied their method to the Xilinx FPGA. Later, Lakamraju and Tessier improved on the idea of shifting logic by introducing the idea of shifting logic within a PLB [33], and thus they reduced the amount of reconfiguration required for logic fault tolerance.

Lach et al. developed low-overhead FT systems [31, 32]. Their basic approach was to partition a design into a number of tiles. Each tile was allocated a number of spare PLBs. In the event a fault in a tile was found, a spare PLB was used to replace the faulty PLB. Their reconfiguration approach provided multiple configurations for each tile. In the event that a fault was detected, the configuration associated with the fault was downloaded. The main drawback of this method is low fault tolerance. It is limited to one fault per tile. If multiple faults occur in close proximity (within a tile), this method breaks down. There is no ability to draw spares from other tiles to cover several faults in a single tile.

With the exception of Lach et al., all previous methods rely on reconfiguration around faults to be calculated on-line. This requires fast computations in order to maintain high availability for the

FT system. The precompiled configurations of Lach et al., on the other hand, significantly reduce system down-time.

5.2.2 Interconnect Fault Tolerance

Many papers have focused on fault tolerance for FPGAs; however, most previous work is targeted primarily toward logic fault tolerance. In this section we discuss previous research in the area of fault tolerance for FPGAs that include interconnect fault tolerance.

Dutt and Hanchek developed an off-line method to increase FPGA yield [9, 10]. Their method uses reserved, spare routing resources. Spare routing resources are used to cover faulty interconnect resources. Spare interconnect resources are allocated as a grid, and when a fault occurs, it is replaced by an extra segment or grid of segments. This method is very fast, but is limited in the fault patterns that it can tolerate; for example, it cannot tolerate a cluster of faults in the same local area. There is also an overhead caused by the reservation of routing resources.

Mahapatra and Dutt proposed a dynamic method for allocating interconnect resources to bypass faults only after faults have been located [36]. If the new required segments conflict with the current usage of routing tracks, the layout is incrementally modified to make room for the new segments.

Incremental routing for logic and interconnect FT was first introduced by Emmert and Bhatia [12, 13]. Their incremental router used the read/write capability of the FPGA configuration memory to ripup and reroute without a netlist. Later Dutt et al. presented similar ideas on incremental routing [10].

Roy and Nag developed a FT method for interconnect faults [47]. They introduced the concept of compatibility for interconnect faults, and demonstrated certain cases where single interconnect faults could be handled independently. However, in [47] they do not handle multiple interconnect faults.

In [32], Lach et al. extend their logic tiling method to consider interconnect faults. Similar to their method for handling logic faults, several replacement configurations are precomputed for programmable signals that do not extend beyond the tile, so that each configuration leaves some routing resources unused. Then when a fault occurs, they try to find a configuration that does not use the faulty resource; this method, however, is not able to tolerate all possible faults or a group of faults in the same tile. For signals that do extend between tiles, they propose two alternatives. In some cases the tile size can be enlarged to encompass the signals. This works well for hierarchical architectures like CPLDs and for dedicated, linked interconnect paths like fast carries. For signals that are routed through interconnects that extend across several tiles, they leave a long segment vacant, to be used to replace any parallel segment in the same area. But after one faulty segment is replaced by the long spare, no additional faults can be tolerated in that region.

Our approach overcomes the main limitations of previous work – the need to stop the system function to allow for diagnosis and computing fault-bypassing reconfigurations, and the inability to tolerate several interconnect faults in the same region. Our approach is the only one integrated

within a system that provides a complete test and diagnosis strategy. Additional features will be presented in the next sections.

5.3 Multilevel Fault Tolerance Approach

A major advantage of our reconfiguration-based approach is that we can tolerate a large number of both logic and interconnect faults. In this subsection we describe our three-level approach.

Level 1 consists of leaving a STAR (or possibly both STARS) parked where faults have been detected. While the STAR is parked over the faults, system operation goes on without interruption because the faults are not located in the working area. STAR parking -- done for both PLB faults and interconnect faults -- continues while the faults are diagnosed by the TREC, possibly using additional configurations for diagnosis in the parked STAR.

The detected faults may affect the next slice of the working logic to be relocated in the parked STAR position. The currently unused logic and routing resources in the working area are considered spares to be used to bypass faults. The first question is whether the application logic is mapped such that after relocation it would fall on the newly diagnosed faulty resources. If the faults affect only spare resources, then no action is required. Otherwise, the next question is whether the faulty resources are usable in the context of the functionality required by the system function. The next subsections will discuss usability in more detail. If the fault is not compatible with the current configuration, then our FT methods incrementally reconfigure the working area to avoid the faults before the working area is relocated in the next step of the STAR roving process. Some fault-bypassing roving configurations (FABRICs) can be precompiled, but most faults will require new FABRICs to be computed on-line; this computation also takes place while the STAR is parked over the faults.

A major advantage of STAR parking is that testing can continue while a STAR is parked. Let us assume that H-STAR is the parked STAR. After the faults in H-STAR have been located, and while TREC is computing FABRICs, V-STAR can continue roving and testing. This process will still test all PLBs and all vertical interconnect, but the horizontal interconnect resources usually tested by the now-parked H-STAR will not be tested during this period. Thus, a partial testing process is active even when one STAR is parked.

Level 2 occurs when we move the parked STAR off of the faults and it restarts roving. During Level 2 we apply the precompiled or newly computed FABRICs. By using the partial reconfiguration capability of FPGAs, the size of these FABRICs is kept small relative to the memory required for storing complete FPGA configurations. A FABRIC replaces a faulty resource with a spare one. Ideally, we would like to always have a spare resource in the neighborhood of the fault. The next subsections will discuss spare allocation strategies for PLBs and interconnect that attempt to achieve this goal.

Level 3 is entered as the last recourse when all the spare PLBs or spare interconnect resources in the working area have been used for fault tolerance. Then we use some of the roving spares provided by the two STARS to bypass faults. This level is referred to as STAR stealing. By making use of reconfiguration and alternately stealing from one STAR and then from the other, it may be

possible to continue roving with both STARs, one at a time. In this case we lose the ability to test global crosspoint CIPs. We will try to maintain both STARs roving one at a time for as long as possible. In the event that one STAR is completely used up (for spares) and we start stealing from the other STAR, we can rove with only a partial STAR. Note that the resources taken from the STAR will no longer be spare. When this occurs, the extent of testing will be reduced with every new fault since some tiles will no longer contain BISTERS, not enough fault-free PLBs may be available to allow accurate diagnosis, and more interconnect resources will no longer be testable.

5.4 Fault Tolerance for PLBs

Because of the roving process, the system logic does not have a fixed placement. Figure 42 illustrates how the same physical PLB is time shared among four different logic cell functions, depending on the positions of the STARs. (A logic cell function is the function mapped to one PLB.) Our FT techniques correctly deal with this dynamic behavior. Whenever possible we reuse faulty PLBs; this is a drastic change from conventional FT techniques that throw out faulty resources. This technique allows us to handle more logic faults than there are available spares. If reuse is not possible, we compute FABRICs. When reconfiguration is required, we have developed optional spare logic preallocation strategies to decrease the time required for reconfiguration and decrease the impact on the size of the incremental configuration files. Another major advantage of our technique over conventional techniques is that we can handle groups of faults in a tight

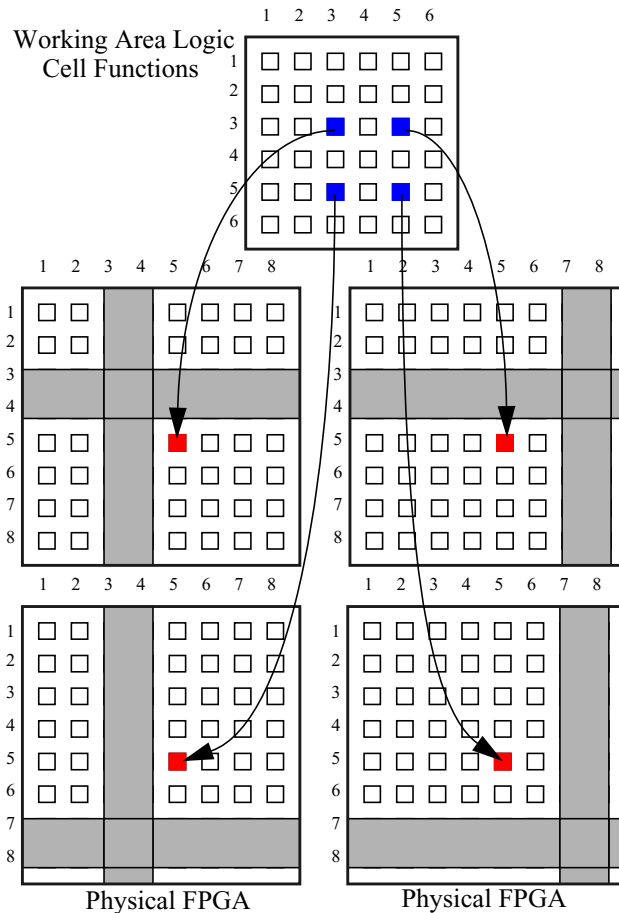


Figure 42. Example of Four Logic Cells Time Sharing One PLB

area. Other techniques limit the number of faults they can handle by providing a limited number of spare logic resources in a subarea or tile of the FPGA. When these local spares are exhausted, their FT techniques fail. In addition to our local spare allocation strategies, we also provide a global reconfiguration approach that makes use of mini-max matching to allow a large number of logic faults in the same area to be tolerated.

5.4.1 Reusing Faulty PLBs

PLBs are usually composed of a set of LUTs and FFs. For example, if one FF is faulty, that FF can't be used by the system function, but the LUTs and the other FFs may be used. As another example, consider a LUT with a memory cell stuck at 0. If the faulty LUT implements a logic function where the faulty memory cell is set to 0, then the fault is tolerated without requiring any reconfiguration and any loss of system resources. If the needed value and the stuck value are complementary, or if the faulty LUT is used as a RAM, then the fault is not compatible. In this case there are two choices. We can reconfigure the system function to completely avoid the faulty LUT, or we can attempt to find a system logic function that can make use of the faulty memory cell in its stuck-at condition. This still requires reconfiguration, but we have eliminated the requirement of a spare resource for the fault.

If the faulty LUT/RAM has an unused input/address bit, another way to tolerate the stuck cell is to simply use the half of the LUT that does not contain the faulty memory cell. For example, if we have a 4-input LUT/RAM with 16 memory cells, and the PLB is using only 3 inputs, we can still use half of the memory cells to implement a 3-input function or a RAM with 3 address bits.

With both of the methods described above we increase the fault tolerance and the lifetime of the system. The usability of a defective LUT has to be analyzed separately for each one of the four logic cell functions of the system function that may be relocated over it.

As stated in the previous subsection, when a faulty programmable logic resource (a LUT or a FF) is not usable, there is still a potential for a PUB, by using the nonfaulty part of the PLB. If the PUB contains a logic cell function that does not use the faulty resource, it can be used as is without any reconfiguration. For example, the fault may be in a section of the PLB that is not used in the relocating logic cell function, such as in an unused LUT or FF, or it may affect a LUT operation not used in the relocating logic cell function, such as multiplication. Sometimes a PUB requires local reconfiguration of a logic cell function [33]. For example, the fault may affect one of the LUTs, but the relocating logic cell function has another LUT currently unused that can be considered as a spare and connected to replace the faulty LUT. The usability of a defective PLB as a PUB also must be analyzed separately for each one of the four logic cell functions of the system logic that may be relocated over it.

We regard usability as a compatibility relation between a fault in a PLB and a desired function for that PLB: if the faulty PLB can correctly perform the function, we say that the fault is compatible with the function. Any fault is compatible with the "empty function" of a spare PLB. A fault that affects only sections of a PLB that are not used in the desired function is also compatible with that function. A defective PLB with unused parts can be made compatible with a desired function by

remapping. For uniformity, we consider a fault-free PLB as having an “empty fault,” which is then compatible with any function.

To show the effect of PUBs when reconfiguring for logic fault tolerance, we used several benchmark circuits. We randomly picked a number of PLBs (and a LUT or FF within each PLB) to be faulty. Then we reconfigured by moving any logic cell function that was not compatible with a faulty PLB to a new location. We increased the number of faults until we could no longer reconfigure the circuit. In Figure 43 we see representative test results for reconfiguring with and without using PUBs. The data for the Digital Single Sideband Modulator (DSSM) circuit show that

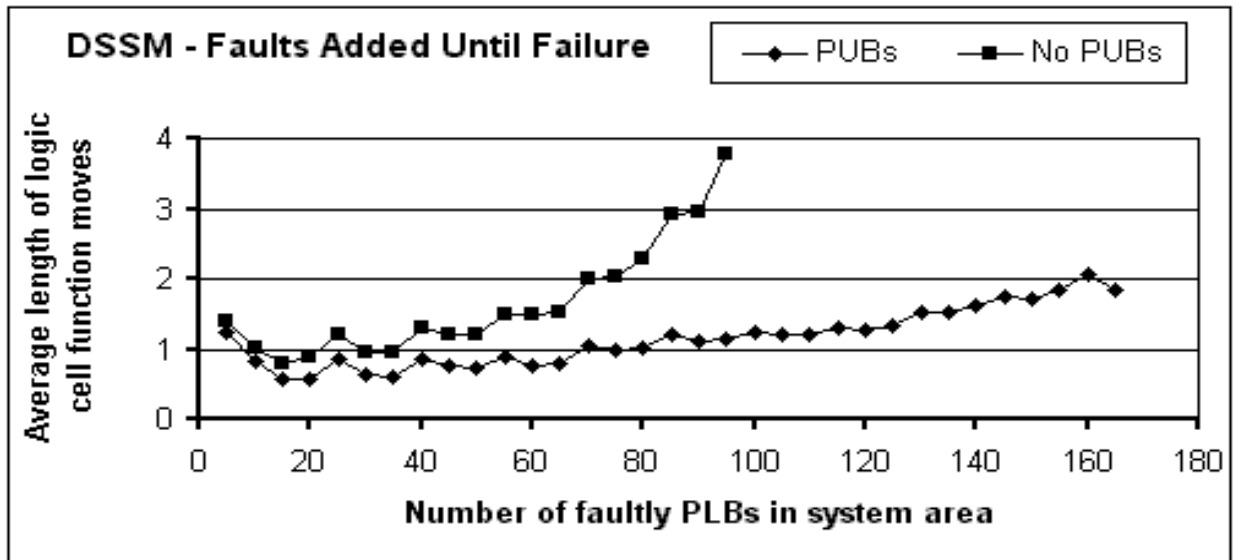


Figure 43. Number of Faults Tolerated with and without PUBs

the number of logic faults that can be tolerated increases with the use of PUBs. The y-axis shows the average distance that an incompatible logic cell function was moved to be placed on a compatible spare PLB or PUB. The x-axis shows the number of faulty PLBs injected into the circuit. Without the use of PUBs, reconfiguration halts when the number of faulty PLBs exceeds the number of spares in the system working area. With PUBs, the number of faulty PLBs actually exceeds the number of spare PLBs in the working area.

5.4.2 Bypassing Incompatible Faults

If a defect is not compatible with the system function, the system function must be reconfigured to avoid the incompatible fault. One approach is to move the entire logic cell function to a compatible spare PLB location. Note that a spare PLB may not be fault free, since we allow a spare PLB to be relocated over a defective one. The condition for replacement is to find a spare PLB compatible with the desired function (this also includes the fault-free spare case). The actual replacement takes place in the next roving step, where the reconfiguration makes the relocating logic cell function fall on a compatible spare instead of the incompatible defective PLB. Having the compatible spare in the neighborhood of the defective PLB helps minimize the extent of the changes in the layout of the system function, including changes in the delays of the signals that have to be rerouted.

Moving a logic cell function to a compatible spare location will work as long as 1) a compatible spare is available and 2) no two logic cell functions need the same spare. For the second case, we use a grid matching technique [11] to perform reconfiguration and use other spares available in the working area of the FPGA. When no more spares are available in the working area, we move to STAR stealing.

FABRICs for single faults can be precompiled and stored to quickly map around defective PLBs. But in a system where some faults have been bypassed, the initially computed FABRICs may no longer be valid; so after one FABRIC is applied, TREC updates the other FABRICs that may have been affected by the incremental reconfiguration. The initial FABRICs are precomputed under the assumption of a single fault, so when multiple faults occur in the same region, TREC has to compute a new FABRIC that concurrently bypasses all the faults. This takes longer than using the precompiled FABRICs, but it does not interfere with the system function execution since a STAR is currently parked over the faults. After several faults have occurred in the same area, it is possible that all the spare PLBs have been used to replace defective PLBs, so it may become necessary to remap the entire system function and reallocate the remaining spares to achieve a more uniform spare distribution. Processing for this is done off-line by the TREC, and it requires the entire system configuration file be reloaded during the next rove.

5.4.3 Preallocating Spare Resources

We describe two optional spare PLB allocation strategies for the working area of the FPGA, but we are not limited to just using these. In the first strategy, we guarantee that for working area PLB utilization less than 80 percent (which is typical for most applications), every system logic cell function will be adjacent to at least one spare PLB. For the second strategy, for a working area PLB utilization less than 92 percent, every system logic cell function is no further than one PLB from a spare.

Initial placement and routing of the system logic is done in the $(N-2)$ by $(N-2)$ working area of the FPGA. For our first strategy, we will constrain the design so that at least 20 percent of the working area PLBs are left spares. This is not excessive, since typical PLB utilization is less than 80 percent for most applications. Higher utilizations make the design difficult to route. We can show that if this constraint is satisfied, we can always find a placement of the system logic so that every system logic cell function is adjacent to at least one spare PLB in the working area of the FPGA. Figure 44a illustrates such a placement for a 12 by 12 FPGA. To force standard design tools to generate evenly distributed spares, we reserve the location of spare PLBs by preplacing dummy logic cell functions in spare PLBs before executing the standard placement algorithm. In our implementation, for each system logic cell function, we select one of its adjacent spares as its preferred replacement. Note that the same spare may be designated as the preferred replacement for several working logic cell functions. In Figure 44, the following equations were used for determining if a position with working area coordinates (r,c) should be reserved as a spare. If we let $\delta=r(\bmod 5)$ and if $c(\bmod 5) = (2\delta + 1)(\bmod 5)$, then working area location r,c is a spare location. Additional spares are needed on the edge of the working area (labeled with e in Figure 44) to ensure that fringe PLBs are sufficiently close to a matching spare PLB in order to meet the defined distance criteria. Figure 44b demonstrates the same concepts for our second strategy. Note that we

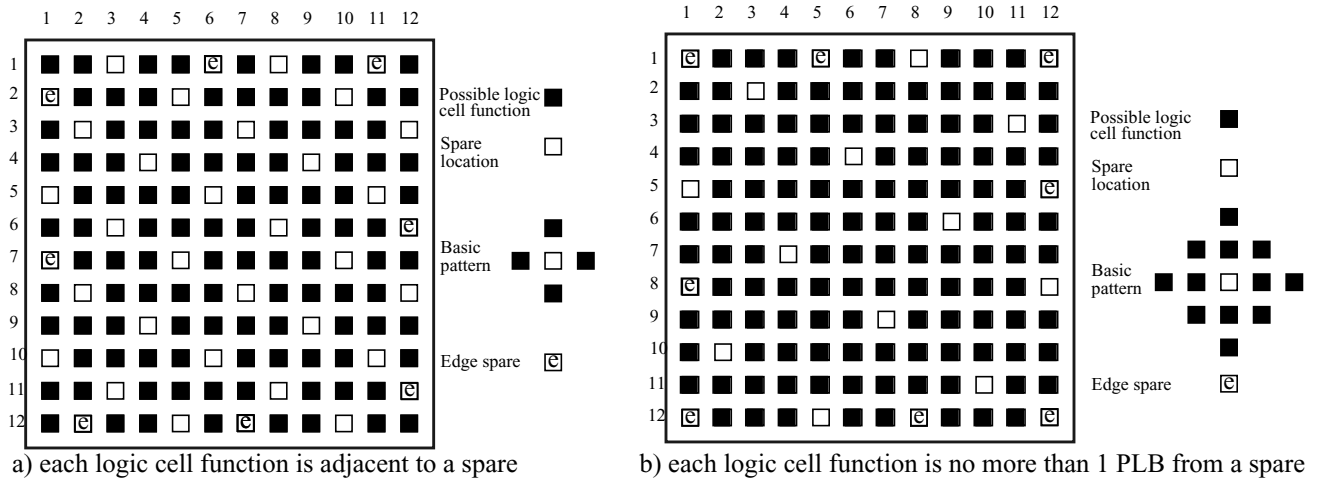


Figure 44. Spare Cell Allocation Patterns

don't expect the PLB utilization to be so high in the initial design, but only after a number of defective PLBs have been replaced by spares.

Table 7 shows the performance penalty for the benchmark circuits described earlier when implemented in an ORCA 2C series FPGA. The column labeled "System w/STARs" gives the worst-case performance for the benchmark circuit with STARs inserted and no preallocated spare PLBs. The column labeled "System w/STARs and FT spares" shows the worst-case operating frequency for the system function with preallocated spares for each of the 10 STAR positions. The last column shows the percent difference in operating frequency between the circuits without and with preallocated spares. For the circuits tested, the percent difference ranges between 2.5 and 15.1 percent. It should be noted that preallocated spares are not a requirement of our FT method; however, they reduce the time for incremental reconfiguration for low numbers of faults.

Table 7. Performance Penalty with Spare PLBs

Circuit	System w/ STARs	System w/STARs and FT spares	% diff
Huffman	98.9	116.6	15.1
Fibonacci	130.8	136.0	3.8
Wallace Tree Multiplier	149.7	166.8	10.3
Digital Single Sideband Modulator	76.6	83.3	7.2
Hilbert	87.5	90.6	3.4
Random Num Generator	56.4	60.1	6.2
Mono-FFT	120.8	123.9	2.5

5.4.4 Multiple Logic Faults

When two logic cell functions are assigned the same spare PLB location, there is a conflict if they both need to be remapped to a spare location. When this occurs, we use a matching strategy to match logic cell functions to compatible spare locations. Minimax matching matches multiple logic cell functions (logic cell functions currently mapped to incompatible PLB locations) to compatible spare locations such that the maximum distance between any logic cell function and its corresponding spare location is minimized. We take this approach to reduce the amount of recon-

figuration required when moving logic cell functions to spare locations and to reduce any adverse effect on system operating speed.

5.5 Fault Tolerance for Interconnect

Because of the roving process, the system interconnect is not fixed. Similar to system logic, programmable routing resources used to connect signal nets may vary depending on the position of the STARS. We have found that for a large number of interconnect faults, the system function does not need to be reconfigured. A large part of our interconnect fault tolerance approach deals with determining whether an interconnect fault is compatible with the layout of the system function. We only reconfigure the system function to tolerate interconnect faults when the layout is incompatible with the faults. Below we describe the process of determining and reusing faulty interconnect resources. For the case when we have an incompatible layout, we describe our multistep approach for incremental system configuration to avoid the incompatible faults. We also describe an optional strategy for reserving spare programmable interconnect resources.

5.5.1 Reusing Defective Interconnect

Reusing defective interconnect is a significant departure from the way most conventional FT techniques work, which is to bypass the faulty interconnect resources. In our approach, we allow a signal net to be routed through faulty interconnect resources (segments and CIPs) whenever possible. This increases the effective circuit routability and leads to more graceful degradation and longer mission life.

We define a layout of a programmable interconnect network as the set of positions (open or closed) for all its CIPs, needed for a specific routing of the netlist. Similar to [47], we say that an interconnect fault f is compatible with a layout L ($f \sim L$) if and only if f does not change the connections implemented by L . If $f \sim L$, then L can tolerate f and we don't have to reconfigure to bypass f .

For example, in Figure 45, assume a layout L where breakpoint CIPs B_1 and B_2 are open, cross-point CIPs X_1 and X_2 are closed, and segment Z is not used. Then an open on B_1X_1 is compatible with L , while an open on X_1X_2 is not. More interestingly, the short between B_1B_2 and Z is also compatible with L , since Z does not carry any signal. Thus the routed signal net can use the shorted segment B_1B_2 . Table 8 gives detailed conditions that need to be met for single-fault compatibility.

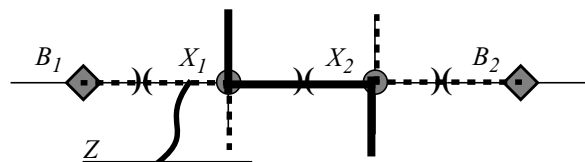


Figure 45. Illustration for Fault Compatibility

Note that the compatibility relation between a fault f and a layout L may be affected by the presence of another fault g . This happens because g changes the structure of the programmable net-

Table 8. Single-Fault Compatibility Conditions

f	Condition for $f \sim L$
segment open <i>or</i> CIP stuck-open <i>or</i> segment stuck-at	L does not transmit values through the faulty resource
CIP stuck-closed	L uses the CIP <i>or</i> L drives values only on at most one of the shorted segments
short between segments	L drives values only on at most one of the shorted segments <i>or</i> L uses the shorted segments for the same signal

work on which L is implemented. This interaction between faults is not considered in [47]. For example, in Figure 46, assume a layout L where crosspoint CIPs X_1 and X_2 are closed, and breakpoint CIPs B_1 and B_2 are open. Segment A is used by signal S_1 via crosspoint X_1 , segment C is used by signal S_2 via crosspoint X_2 , and segment B is not used. Let f be the short between seg-

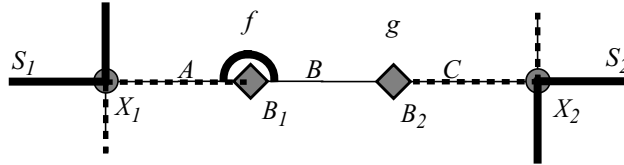


Figure 46. Multiple Fault Interaction

ments A and B , and let g be the breakpoint B_2 stuck-closed. The single fault f is compatible with L , as is the single fault g . However, the multiple fault $\{f, g\}$ shorts the signals S_1 and S_2 , so it is not compatible with L . Figure 47 shows the opposite situation, where an incompatible fault (open on segment X) becomes compatible with the layout in the presence of a short between A and B , because the short restores the connection cut by the open.

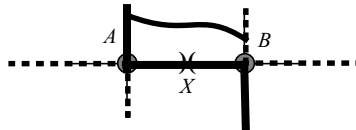


Figure 47. Compatible Multiple Fault

Our procedures rely on a network model coupled with a layout database. The network model is a graph that represents the topological relations between the components of the programmable interconnect network – wire segments, PLB pins, FPGA pins, and CIPs. The layout database defines the CIP settings required to route all the signal nets in the circuit. From the layout database we can retrieve all the segments and CIPs used by a given signal, and determine which signal is routed through a given segment or CIP. In a fault-free network, a segment may be driven by only one signal.

To record faults in the network model, every CIP and segment has an additional field that indicates its fault status: a CIP can be fault free, stuck-on, or stuck-off, and a segment can be fault free, open, stuck, or shorted. For a shorted segment, we also record the identity of its partner – the other segment involved in the short. As illustrated in Figure 48, the presence of crosspoint CIPs on a shorted segment complicates the reuse problem, because the actual short may involve any one of the nine pairs of portions of the two wire segments (A,X) , (A,Y) , ..., (C,Y) , (C,Z) . While this

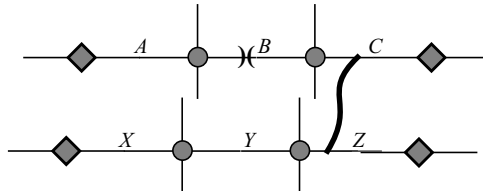


Figure 48. Where Is the Short?

resolution is not needed to model the short, it may lead to a problem if an open also occurs on one of the original segments: for example, if B becomes open, the topology of the network is different depending on which side of the open the short resides. In such a case we use a separate diagnosis procedure [54] to determine the exact topology caused by the multiple fault.

We determine the faulty resources that may be reused (and the ones that need to be bypassed) indirectly, by finding the signal nets that have to be rerouted. This is more efficient, since our incremental router works based on signal nets, and the same net may be affected by more than one fault. We say that a signal net S (part of layout L) is compatible with a group of faults if and only if S can connect its source to all of its sinks in the presence of the faults, and no other signal drives the segments of S . A signal has to be rerouted only if it is incompatible with the current group of faults.

The procedure to find the incompatible nets works as follows. We first insert all the located faults in the network model. Next, we process the faults to find all the signal nets affected by faults. Then we trace every affected signal S from its source to all its sinks, keeping track of the segments used. Tracing stops at open segments, at stuck-open CIPs, and at stuck segments, but it does go through stuck-on CIPs, and it jumps from a shorted segment to its partner. If faults prevent S to reach at least one of its destinations, then S is incompatible. We record all the segments reached by every traced net. After all signals are traced, we can identify the segments driven by more than one signal; the signals that drive these segments are shorted, and only one of them may be allowed to continue to drive the shorted segments, while the other ones will be marked as incompatible. To determine which signal to leave in place, first we exclude the ones already found to be incompatible because the trace has not reached some of their destinations. To select among the remaining signals, we analyze the timing margins (slacks) of their paths, and chose the one with the least slack. In this way, we minimize the impact of rerouting on the system timing, since the signals to be rerouted have more slack. We break ties by selecting the net most difficult to reroute as the one left in place. Rerouting difficulty for a net can be approximated by its length, since shorter nets are likely to be easier to reroute. Other criterion could be based on layout density.

For example, in Figure 46, tracing S_1 will go through A , B , C , and continue along S_2 , while tracing S_2 will go through C , B , A , and continue along S_1 . Since the traced segments are recorded as being driven by both S_1 and S_2 , these signals are shorted. Assuming that neither S_1 nor S_2 have other faults, we select one of them to be left unchanged (based on the analysis described above), while the other will be allowed to continue to drive the shorted segments. This is possible since after rerouting, the multiple fault will become compatible with the new layout.

Tracing the signal in Figure 47 will not go through the open segment X , but it will reach all its destinations following the connection between A and B created by the short. Hence the signal is compatible with the multiple fault, and it does not have to be rerouted.

In Figure 49, assume that the source of S_1 is on the left. Then tracing S_1 stops at the open segment X , and tracing S_2 reaches B via the short between B and C . Because of the open, B is driven by only one signal (S_2), so that there is no other signal shorted to S_2 , which is hence compatible with the multiple fault, and does not have to be rerouted.

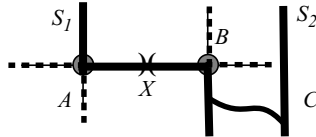


Figure 49. Signal Compatibility

The fault status of CIPs and segments is recorded in the network model. An ideal router would correctly use this information, so that it will never route a net through a stuck-open CIP, an open segment, or a stuck-at segment. However, our current implementation relies on a commercial router that does not provide such features. Our workaround is to mark the incompatible faults by creating dummy nets that use the corresponding faulty resources. These dummy nets have no source or sink and are created only to tie up unusable CIPs and segments. They are never ripped up, and thus they will not be used to route any real nets. However, this solution is less flexible than the ideal one.

Note that routing done in a faulty interconnect network can take advantage of faults to route signal nets in ways that would have been impossible in the fault-free network. For example, Figure 50 shows an open that allows the use of crosspoints X_1 and X_2 by two different signals, S_1 and S_2 ; in the fault-free network, they may be used only by the same signal.

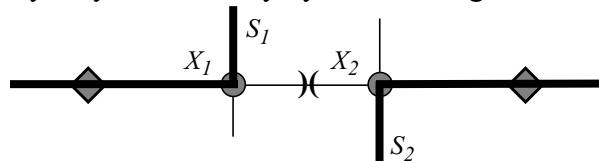


Figure 50. Taking Advantage of an Open Segment

The described solution is an ideal one since it assumes that the router can read the layout database in which we maintain the fault status of CIPs and segments. In our implementation, we use a commercial router that requires us to reserve a CIP and two wire segments to prevent the incorrect use of faulty resources. However, we can reuse shorts. When an incompatible short occurs (two or more signals using the same net segment group), we will compare the signals against the critical path. If one of the signals is in the critical path, we will keep it intact and ripup the other signals. If neither signal is in the critical path, we will pick the one to ripup based on net length, since shorter signals should be easier to reroute. We cannot make use of opens with the commercial router; we can only use one portion of the open segments. The ability to use multiple open segment in what was once a continuous wire segment would require a specialized router. Our implemented solution relies on marking the incompatible faults by creating dummy nets that use the corresponding faulty resources. These dummy nets have no source or sink and are just used to tie up unusable CIPs and segments. They are never ripped up, and thus they will not be used to route any real nets. However, this solution is less flexible than the ideal one.

As a side note, we discovered a shortcoming (relative to fault tolerance) of the critical path analysis tool (*trce*) used in our implementation. This is worth mentioning because other such static tim-

ing analysis tools also suffer from the same limitations. The specific problem relates to shorts (either shorted segments or stuck-closed CIPs). When a short occurs, there may be additional capacitance added to segments used to route signal nets. In this case, *trce* will return a maximum clock speed or a minimum propagation delay that is inaccurate and may cause the clock to be set at an inappropriate speed. This is a problem only if the additional delay, not visible to *trce*, creates a path delay longer than the clock period. To accurately set the clock speed, the static timing analysis tool must work with the updated model illustrated in Figure 45.

5.5.2 Bypassing Incompatible Interconnect Faults

Routing and incremental routing are NP-hard problems. For dense circuits, finding one solution where the circuit is completely routed may be impossible. If a solution is found, there may not be any alternate solutions. Additionally, routing run-times may be exceedingly long. Thus in our approach to interconnect fault tolerance, we try to maintain the original circuit routing as much as possible. This improves the possibility of obtaining a 100 percent routed circuit, and reduces the amount of time required to incrementally route. Given an interconnect fault f that is not compatible with the layout, we use a multistage fault-bypassing technique; if we have a set of faults, we handle them the same way we handle a single fault. We must perform our routing fault tolerance for each of the STAR positions. Each step in our process is described below.

Stage 1: In this stage we perform local logic reconfiguration to bypass interconnect faults that will not allow some signal nets to connect to specific inputs or outputs on a PLB. If an input to a PLB is blocked, we try to move the function of the blocked resource to an unused resource in the same PLB. For example, Figure 51 shows an example of a stuck-open CIP denying access to an input pin on LUT B. If LUT A is not being used, there is a possibility we can move the function from LUT B to LUT A; keeping the LUT function within the same PLB is less likely to introduce delays. For this, we reconfigure the PLB logic and we partially (or fully) ripup the signal nets going to the inputs and outputs of LUT A [33]. We add the signals to the set S of unrouted nets for later incremental rerouting. If such a transformation is not possible within the blocked PLB, we partially (or fully) ripup all the signal nets attached to the PLB, move the logic cell function of the PLB to a new compatible spare location (using mini-max grid matching), and add the ripped up signal nets to S .

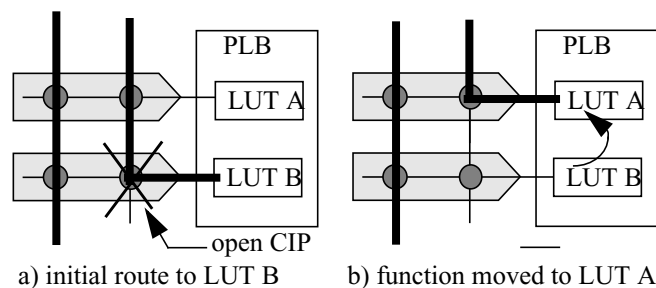


Figure 51. Example Stuck-Open CIP

Stage 2: In this stage we perform partial signal net ripup. We partially ripup any signal net s that is incompatible with f and add it to our set S of unrouted signal nets. If f is in an uncongested global area or a bypassable local area, it may be possible to ripup the signal net without disturbing other routed signal nets. For example, Figure 52 shows a signal net that was originally routed through a segment between crosspoint CIPs X_1 and X_2 that is now open. If X_3 and X_4 aren't being

used in other signal nets, the part of the signal route between X_1 and X_2 can be ripped up and rerouted through X_1 , X_3 , X_4 , and X_2 .

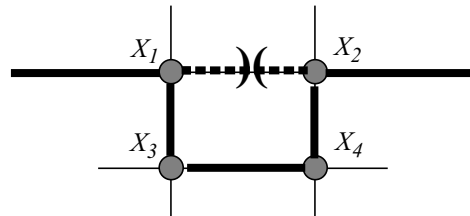


Figure 52. Bypassing an Open Segment

Stage 3: In this stage, we attempt to incrementally reroute each $s \in S$ without changing the route of other nets not affected by faults. Initially S contains all nets affected by faults. Note that we mark any f so the router won't use f in any subsequent route. If we are successful, we can stop our interconnect FT reconfiguration at this stage. Otherwise, we move on to subsequent stages.

Stage 4: After Stages 1 through 3, if we have unrouted nets (S is not empty), we enter Stage 4. This is likely to occur in areas of congested routing, where most routing resources have been already used or/and are faulty. Here we use a more global transformation, ripping up all the signal nets within by a variable-sized window in the area around the unrouted signal nets. If we are not successful, we increment the window size and repeat the process until we reach the boundaries of the FPGA. If we are successful, we stop. If at the end we still have unroutes, we proceed to Stage 5.

Stage 5: At this stage we have been unable to achieve a 100 percent routed circuit. Thus to achieve a routed circuit, we require more drastic measures. In this stage, we verify that all faulty logic and interconnect resources are properly marked as either partially or completely unusable. Then within areas reserved for STARS, we perform system function place and route with no preallocated spare resources in place. In the event that this is successful, we can incrementally reinsert spare resources for future use. Note that the commercial place and route tools we use for implementation do not have the ability to make use of partially usable logic or interconnect resources. For this we require special tools with modifications to check for compatibility.

If Stage 5 is not successful, we can steal resources from a STAR to increase available resources for placing and routing the system function. With one STAR still available for testing, we retain the ability to test all logic and some of the interconnect resources.

5.5.3 Preallocating Spare Resources

Our approach does not require that we preallocate spare interconnect resources, since typically an FPGA will have many unused interconnect resources. Thus we have implemented an optional spare interconnect preallocation strategy to evenly distribute the spare interconnects across the FPGA. This will increase the likelihood of having a spare in the vicinity of the fault, which in turn will reduce fault avoidance reconfiguration time, will increase the probability of a successful incremental routing, and minimize the impact of reconfiguration on nonfaulty signal nets. Preallocating spare interconnect resources also has disadvantages. There is an inherent increase in the initial circuit place and route time, and for dense circuits performance can also be degraded.

Our spare interconnect preallocation strategy is based on routing a set of dummy signals that are evenly distributed across the FPGA. To accomplish this, we reserve programmable routing resources by using them to route dummy signal nets that have no sources or sinks. Below we list the sequence for inserting spare interconnect resources, and in the rest of this section we describe the steps.

- 1) Define a set of dummy signals.
- 2) Route the set of dummy signals.
- 3) Place and route the system application function.
- 4) Remove the dummy signals.

In step 1, we create a parameterized set of dummy signal nets. The parameters controlling the type, number, and direction of dummy signal nets form a triple (t, w, d) . The variable t controls the type of interconnect used for the dummy signal net. Types range from segments that are one PLB in length to segments that span the entire FPGA. The variable w defines the number of spares of type t to be reserved in each channel of the FPGA. The variable d defines which channels are used for the dummy signal nets. Valid values of d include H (horizontal), V (vertical), B (both horizontal and vertical).

In step 2, we route the dummy signals defined in step 1. We accomplish this step using a special tool that reserves specific, user-defined interconnect resources (so they will not be used by the general router to route the circuit) for spares.

If the system function will not route in step 3, we go back and reduce the number of interconnect resources reserved in step 1 by reducing w or/and d . Then in step 4 we remove the dummy signals (and any dummy logic reserved for logic fault tolerance), leaving the interconnect (and logic) spares available for fault bypassing.

5.6 Fault Tolerance Summary

In this section we have described our FT approach for both logic and interconnect. In our approach, we have introduced several new concepts for both logic and interconnect fault tolerance. Since determination of all FT configurations (if they are required) takes place with one or both STARS parked over the faults, the system function is free to continue executing, unaffected by any faults. Thus, when used with the roving STARS approach for test and diagnosis of faults on FPGAs, the approach is on-line.

Whenever possible we make use of faulty programmable resources. For logic we use the fault if it is compatible with the function programmed in the faulty resource. If the fault is not compatible with the current logic, we attempt to find logic that is compatible with the fault. If that fails, we use the faulty PLB as a PUB, and make use of the nonfaulty portions of the PLB. This technique is a big improvement over previous techniques that bypassed the faulty PLB. We have shown that we can actually continue to execute even when the number of faulty PLBs exceeds the number of spare PLB resources.

When we do reconfigure for logic fault tolerance, we make use of optional, preallocated local spare resources whenever possible. Our spare resource allocation strategies guarantee local (adjacent) spares if the logic resource usage is less than 80 percent. If local spares are not available, or if a group of PLBs become faulty, we use a mini-max grid matching strategy to match faulty logic cell locations to compatible spare resources such that the amount of reconfiguration is minimized. We have introduced the idea of a programmable clock for fault tolerance. This allows the clock to initially be set to its fastest possible rate, and the clock is only reduced as necessary when faults affect the circuit's critical paths.

When using faulty interconnect we consider not only single faults but also multiple faults. For sparse circuits, most of the interconnect faults that can occur are compatible with the circuit layout. If the interconnect faults are compatible with the current configuration of the circuit layout we do not reconfigure. This is one way to increase the fault tolerance of the circuit and extend the lifetime of the circuit. When reconfiguring to avoid interconnect faults, we use a five-step process to determine FABRICs. We also presented an optional interconnect spare allocation strategy to evenly distribute spare interconnect resources across the FPGA.

6. Summary and Conclusions

We have developed an integrated approach for on-line FPGA testing, diagnosis, and fault tolerance, applicable to any FPGA supporting incremental RTR. Its central concept of roving STARs sets aside two sections of the FPGA in which self-testing goes on without disturbing the normal system activity in the rest of the chip. The roving of the STARs periodically brings every section of the device under test. Our approach guarantees complete testing of both PLBs and interconnect, and does not require any part of the chip (including the configuration memory) to be fault free. A STAR consists of several independent BISTERS, which concurrently test disjoint tiles of PLBs and interconnect resources within the STAR. The logic BISTER is repeatedly configured using a rotating strategy so that every PLB in its tile is completely tested in every mode of operation, and practically any combination of faulty PLBs is guaranteed to be detected. The interconnect BISTER is repeatedly configured to completely test all interconnect resources (both global and local) within the STAR for shorts and opens as well as CIP faults and faults affecting the configuration memory bits that control the CIPs. All test-related activities, including STAR reconfigurations, are done without disturbing the normal system operation.

Our adaptive diagnosis technique guarantees maximal diagnostic resolution in locating defective PLBs and interconnect faults. In addition to reporting a faulty PLB, we can also identify its failing internal module or its failing mode of operation. This level of diagnostic accuracy provides the basis of a new form of fault tolerance, where a defective PLB is allowed to be used in the system logic, provided that its intended operation is not affected by the identified faults. Similarly we can diagnosis interconnect faults to a level of resolution that facilitates the reuse of faulty interconnect resources. This reuse of fault logic and routing resources results in a more graceful degradation and longer mission life.

We have developed a dynamic FT method, where both spare interconnect and spare PLB resources needed to bypass a fault are allocated only after the fault has been detected and diagnosed, and the spare resources are always present in the neighborhood of the located fault. Thus fault reconfiguration paths may be much shorter than in other methods. Since the detected faults are always in a spare area, the normal system operation need not be interrupted to diagnose and bypass the fault.

We have implemented this approach using commercially available FPGAs – the ORCA series 2C and 2CA. We have successfully used our integrated approach for both fault-free and faulty FPGAs that included logic and interconnect faults. To further evaluate our testing, diagnosis, and FT techniques, we have developed a fault injection emulation capability that allows us to inject any number of logic and routing faults into any region of the FPGA before and/or during operation of the FPGA with the system function and roving STARs. Our roving STARs approach has been applied to seven different benchmark system functions, including two from the set of ACS benchmark circuits. These results have demonstrated the feasibility of our approach.

7. Participants, Patents, and Publications

During the course of this project, the participants included one Distinguished Member of the Technical Staff from Agere systems, two Department of Electrical and Computer Engineering faculty, nine MSEE graduate students and nine BSEE undergraduate students. All participants were U.S. citizens with the exception of two graduate students (one from Sri Lanka and one from India). These participants are summarized below in terms of their role in the project as well as the time period during which they made their contribution:

1. **Miron Abramovici** (Distinguished Member of the Technical Staff, Agere Systems) was responsible for the overall project and its coordination.
2. **John M. Emmert** (Assistant Professor, UNC-Charlotte) was responsible for the fault tolerance for both logic and routing resources as well as the TREC.
3. **Charles E. Stroud** (Professor, UNC-Charlotte) was responsible for the on-line testing and diagnosis of both logic and routing resources.
4. **Sajitha Wijesuriya** was responsible for the implementation of the initial BIST architecture for the PLBs as well as the RTR process. He joined the project in May 1998 and received an MSEE degree in May 1999.
5. **Carter Hamilton** was responsible for the roving process and initial implementation of the TREC. He joined the project in May 1998 as an undergraduate and received BSEE and MSEE degrees in December 1998 and May 2000, respectively.
6. **Brandon Skaggs** was responsible for the development of the logic BIST and PUB diagnosis. He joined the project in March 1999 and received BSEE and MSEE degrees in May 1999 and August 2000, respectively.
7. **Matthew Carter** was responsible for the initial development of interconnect BIST. He joined the project in May 1999 and received a BSEE degree in May 2000.
8. **Matthew Lashinsky** was responsible for the development of the interconnect BIST. He joined the project in January 2000 and received BSEE and MSEE degrees in May 2000 and December 2001, respectively. He was a finalist for the UNC-Charlotte Cameron Applied Research Award in 2001 for his work on this project.
9. **Jeremy Nall** was responsible for the development of interconnect diagnostics. He joined the project in February 2000 and received a BSEE degree in December or 2001. He will graduate with an MSEE in May 2002. He was a finalist for the UNC-Charlotte Cameron Applied Research Award in 2002 for his work on this project.
10. **James Roller** was responsible for initial development of interconnect BIST. He joined the project in January 2001 and received a BSEE degree in May 2001.
11. **Andrew Taylor** was responsible for interconnect fault tolerance. He joined the project in February 2000 and received a BSEE degree in December or 2001. He expects to graduate with an MSEE in August 2002.
12. **Stanley Baumgart** was responsible for logic fault tolerance. He joined the project in April 2000 and received an MSEE degree in August 2001.

13. **Pankaj Kataria** was responsible for interconnect fault tolerance. He joined the project in September 1999 and received an MSEE degree in May 2001.
14. **Jason Cheatham** was responsible for the TREC. He joined the project in September 1999 and received BSEE and BSCS degrees in December 2001 and May 2002, respectively.
15. **Michelle Cheatham** was responsible for the GUI. She joined the project in May 2001 and received BSCS and MBA degrees in December 2001 and May 2002, respectively.
16. **Thomas Slaughter** was responsible for the fault injection emulator. He joined the project in May 2001 and expects to graduate with a BSEE degree in May 2003.
17. **Vinay Verma** worked on the initial design of the BISTERS during the summer of 1998. He is a Ph.D. student at the University of Illinois, Chicago.
18. **Mark Boyd** worked on the initial implementation of the roving process during the summer 1999. He is a Ph.D. student at the University of California, Santa Cruz
19. **Xiaoming Yu** worked on the problem of mixed-mode diagnosis during the summer 2001. He is a Ph.D. student at the University of Illinois, Champaign-Urbana.
20. **Shantanu Dutt** (Assistant Professor, University of Illinois, Chicago) participated in the initial discussions on the fault-tolerance features of the project.

The project has resulted in a number of patents as well as papers and presentations at conferences and workshops, along with invited presentations as summarized below:

Patents:

1. M. Abramovici and C. Stroud, "Fault Tolerant Operation of FPGAs," U.S. Patent # 6,256,758 issued 7/3/2001.
2. M. Abramovici and C. Stroud, "On-line Testing of the Programmable Logic Blocks in FPGAs," pending, application no. 09/405,958, filed 9/27/1999, patent has been allowed.
3. M. Abramovici and C. Stroud, "On-line Testing of the Programmable Interconnect Network in FPGAs," pending, application no. 09/406,219, filed 9/27/1999, patent has been allowed.
4. M. Abramovici and C. Stroud, "On-line Diagnosis of the Programmable Logic Blocks in FPGAs," pending, application no. 09/671,853, filed 9/27/1999.
5. M. Abramovici, J. Emmert, and C. Stroud, "On-line Fault Tolerant Operation via Incremental Reconfiguration of FPGAs," pending, application no. 09/611,449, filed 7/6/2000.
6. M. Abramovici and C. Stroud, "Diagnosis of the Programmable Interconnect Network in FPGAs," pending, application no. 09/994,299, filed 11/26/2001.
7. M. Abramovici, J. Emmert, and C. Stroud, "Interconnect Fault Tolerance in FPGAs," pending, application no. 60/305,534, filed 7/6/2001.
8. M. Abramovici and C. Stroud, "BIST-Based Delay Fault Testing in FPGAs", patent application in preparation, to be filed 7/8/2002.

Journal Papers:

1. M. Abramovici and C. Stroud, "BIST-Based Test and Diagnosis of FPGA Logic Blocks", *IEEE Trans. on VLSI Systems*, Vol. 9, No. 1, pp. 159-172, 2001.
2. J. Emmert, J. Cheatham, P. Kataria, C. Stroud, and M. Abramovici, "Predicting Performance Penalty for Fault Tolerance in Roving Self-Testing AREAs (STARs)," *Lecture Notes in Computer Science*, Vol. 1896, pp. 545-554, 2000.

Conference Papers & Presentations:

1. M. Abramovici, C. Stroud, S. Wijesuriya, C. Hamilton, and V. Verma, "Using Roving STARs for On-Line Testing and Diagnosis of FPGAs in Fault-Tolerant Applications," *Proc. International Test Conf.*, pp. 973-982, 1999.
2. M. Abramovici and C. Stroud, "BIST-Based Delay Fault Testing in FPGAs," to be published *Proc. IEEE International On-Line Testing Workshop*, 2002.
3. C. Stroud, J. Nall, M. Lashinsky, and M. Abramovici, "BIST-Based Diagnosis of FPGA Interconnect," to be published *Proc. IEEE International Test Conf.*, 2002.
4. M. Abramovici, J. Emmert, and C. Stroud, "Roving STARs: An Integrated Approach to On-Line Testing, Diagnosis, and Fault Tolerance for FPGAs in Adaptive Computing Systems," *Proc. NASA/DoD Workshop on Evolvable Hardware*, pp. 73-92, 2001.
5. M. Abramovici, C. Stroud, and J. Emmert, "Using Embedded FPGAs for SoC Yield Improvement," *Proc. AMC/IEEE Design Automation Conf.*, pp. 713-724, 2002.
6. M. Abramovici and C. Stroud, "BIST-Based Detection and Diagnosis of Multiple Faults in FPGAs," *Proc. IEEE International Test Conf.*, pp. 785-794, 2000.
7. M. Abramovici, C. Stroud, B. Skaggs, and J. Emmert, "Improving BIST-Based Diagnosis for Roving STARs," *Proc. IEEE International On-Line Testing Workshop*, pp. 31-39, 2000.
8. C. Hamilton, G. Gibson, S. Wijesuriya, and C. Stroud, "Enhanced BIST-Based Diagnosis of FPGAs via Boundary Scan Access," *Proc. IEEE VLSI Test Symp.*, pp. 413-418, 1999.
9. C. Stroud, M. Lashinsky, J. Nall, J. Emmert, and M. Abramovici, "On-Line BIST and Diagnosis of FPGA Interconnect Using Roving STARs," *Proc. IEEE International On-Line Testing Workshop*, pp. 31-39, 2001.
10. J. Emmert, S. Baumgart, P. Kataria, A. Taylor, C. Stroud, and M. Abramovici, "On-Line Fault Tolerance for FPGA Interconnect with Roving STARs," *Proc. Defect and Fault Tolerance in VLSI Systems Conf.*, pp. 445-454, 2001.
11. T. Slaughter, C. Stroud, J. Emmert, and B. Skaggs, "Fault Injection Emulation for Field Programmable Gate Arrays," *Proc. International Society for Optical Engineering ITCOM*, pp. 1-9, 2001.
12. J. Emmert, C. Stroud, B. Skaggs, and M. Abramovici, "Dynamic Fault Tolerance in FPGAs via Partial Reconfiguration," *Proc. Field-Programmable Custom Computing Machines Conf.*, pp. 165-174, 2000.

13. J. Emmert and J. Cheatham, "On-Line Incremental Routing for Interconnect Fault Tolerance in FPGAs Minus the Router," *Proc. Defect and Fault Tolerance in VLSI Systems Conf.*, pp. 149-157, 2001.

MSEE Theses:

1. Sajitha Wijesuriya, "Built-In Self-Test for Programmable Interconnect in Field Programmable Gate Arrays," MSEE thesis, Dept. of Electrical Engineering, University of Kentucky, May 1999.
2. Carter Hamilton, "On-Line Testing and Reconfiguration of Field Programmable Gate Arrays," MSEE thesis, Dept. of Electrical Engineering, University of Kentucky, Aug. 2000.
3. Brandon Skaggs, "On-Line Built-In Self-Test and Diagnosis of Field Programmable Gate Array Logic Blocks," MSEE thesis, Dept. of Electrical Engineering, University of Kentucky, Aug. 2000.
4. Pankaj Kataria, "Incremental Routing Strategy Based on Window Rip-Up," MSEE thesis, Dept. of Electrical & Computer Engineering, University of North Carolina at Charlotte, May 2001.
5. Stanley Baumgart, "On-Line Reconfiguration of FPGAs for Fault Tolerant Applications in Adaptive Computing Systems," MSEE thesis, Dept. of Electrical & Computer Engineering, University of North Carolina at Charlotte, Aug. 2001.
6. Matthew Lashinsky, "On-Line Built-In Self-Test of Field Programmable Gate Array Interconnect," MSEE thesis, Dept. of Electrical & Computer Engineering, University of North Carolina at Charlotte, December 2001.
7. Jeremy Nall, "On-Line Diagnosis of Field Programmable Gate Array Interconnect," MSEE thesis in preparation, Dept. of Electrical & Computer Engineering, University of North Carolina at Charlotte, expected graduation May 2002.
8. Andrew Taylor, "Faulty Interconnect Reuse and Incremental Routing for Enhanced FPGA Interconnect Fault Tolerance," MSEE thesis in preparation, Dept. of Electrical & Computer Engineering, University of North Carolina at Charlotte, expected graduation August 2002.

Invited Presentations:

1. Design Automation Conference (invited tutorial), New Orleans, LA, June 2002.
2. University of Iowa, Iowa City, IA, April 2002.
3. Carnegie-Mellon University, Pittsburgh, PA, May, 2001.
4. Institute for System Engineering Research (INESC), Lisbon, Portugal, July, 2001.
5. University of Porto, Porto, Portugal, November, 2001.
6. Spotlight on Research TV Series, Charlotte, NC, September, 2001.
7. University of Michigan, Ann Arbor, MI, March 2002.
8. Purdue University, Lafayette, IN, March 2002.
9. University of Kentucky, Lexington, KY, March, 2002.

- 10.VLSI Test Symposium, Monterey, CA, April, 2002.
- 11.Texas A&M University, College Station, TX, April, 2002.
- 12.Duke University, Raleigh, NC, to be presented August 2002.

8. References

- [1] M. Abramovici, M. Breuer, and A. Friedman, *Digital Systems Testing and Testable Design*, IEEE Press, 1994.
- [2] M. Abramovici and C. Stroud, "BIST-Based Test and Diagnosis of FPGA Logic Blocks," *IEEE Trans. on VLSI Systems*, Vol. 9, No. 1, pp. 159-172, 2001.
- [3] M. Abramovici, C. Stroud, B. Skaggs, and J. Emmert, "Improving On-Line BIST-Based Diagnosis for Roving STARS," *Proc. IEEE Intn'l On-Line Test Workshop*, pp. 31-39, 2000.
- [4] M. Abramovici, C. Stroud, S. Wijesuriya, C. Hamilton, and V. Verma, "Using Roving STARS for On-Line Testing and Diagnosis of FPGAs in Fault-Tolerant Applications," *Proc. Intn'l. Test Conf.*, pp. 973-982, 1999.
- [5] Altera Corp., <http://www.altera.com>.
- [6] A. Burress and P. Lala, "On-Line Testable Logic Design for FPGA Implementation," *Proc. IEEE Intn'l. Test Conf.*, pp. 471-478, 1997.
- [7] R. Cuddapah and M. Corba, *Reconfigurable Logic for Fault Tolerance*, Springer-Verlag, 1995.
- [8] S. Durand and C. Piguet, "FPGAs with Self-Repair Capabilities," *Proc. ACM Int'l Workshop on FPGAs*, pp. 1-6, 1994.
- [9] S. Dutt and F. Hanchek, "REMOD: A New Methodology for Designing Fault-Tolerant Arithmetic Circuits," *IEEE Trans. on VLSI Systems*, Vol. 5, No. 1, pp. 34-56, 1997.
- [10] S. Dutt, V. Shanmugavel, and S. Trimberger, "Efficient Incremental Rerouting for Fault Reconfiguration in Field Programmable Gate Arrays," *ACM/IEEE Intn'l Conf. on Computer-Aided Design*, pp. 173-177, 1999.
- [11] J. Emmert and D. Bhatia, "Partial Reconfiguration of FPGA Mapped Designs with Applications to Fault Tolerance," *Proc. Intn'l Conf. on Field-Programmable Logic*, pp. 141-150, 1997.
- [12] J. Emmert and D. Bhatia, "Incremental Routing in FPGAs," *Proc. IEEE Intn'l Application Specific Integrated Circuits Conf.*, pp. 302-305, 1998.
- [13] J. Emmert and D. Bhatia, "A Fault Tolerant Technique for FPGAs," *Journal of Electronic Testing*, Vol. 16, pp. 591-606, 2000.
- [14] J. Emmert, C. Stroud, J. Cheatham, A. M. Taylor, P. Kataria, and M. Abramovici, "Performance Penalty for Fault Tolerance in Roving STARS," *Proc. Intn'l Conf. on Field Programmable Logic*, pp. 545-554, 2000.
- [15] J. Emmert, C. Stroud, B. Skaggs, and M. Abramovici, "Dynamic Fault Tolerance in FPGAs via Partial Reconfiguration," *Proc. IEEE Symp. on Field-programmable Custom Computing Machines Conf.*, pp. 165-174, 2000.
- [16] *Field Programmable Gate Arrays Data Book*, Lucent Technologies, 1998.
- [17] C. Hamilton, "On-Line Testing and Reconfiguration of Field Programmable Gate Arrays," MSEE Thesis, Dept. of Electrical Engineering, Univ. of Kentucky, 2000.
- [18] C. Hamilton, G. Gibson, S. Wijesuriya, and C. Stroud, "Enhanced BIST-Based Diagnosis of FPGAs via Boundary Scan Access," *Proc. IEEE VLSI Test Symp.*, pp. 413-418, 1999.
- [19] F. Hanchek and S. Dutt, "Methodologies for Tolerating Logic and Interconnect Faults in FPGAs," *IEEE Trans. on Computers*, Vol. 47, No. 1, pp. 15-33, 1998.

- [20] I. Harris and R. Tessier, "Interconnect Testing in Cluster-Based FPGA Architectures," *Proc. ACM/IEEE Design Automation Conf.*, pp. 49-54, 2000.
- [21] I. Harris and R. Tessier, "Diagnosis of Interconnect Faults in Cluster-Based FPGA Architectures," *Proc. IEEE Intn'l Conf. on Computer Aided Design*, pp. 472-476, 2000.
- [22] N. Hastie and R. Cliff, "The Implementation of Hardware Subroutines on Field Programmable Gate Arrays," *Proc. IEEE Custom Integrated Circuits Conf.*, pp. 31.4.1-31.4.4, 1990.
- [23] F. Hatori, et al., "Introducing Redundancy in Field Programmable Gate Arrays," *Proc. IEEE Custom Integrated Circuits Conf.*, pp. 7.1.1-7.1.4, 1993.
- [24] N. Howard, A. Tyrrell, and N. Allinson, "The Yield Enhancement of Field Programmable Gate Arrays," *IEEE Trans. on VLSI Systems*, Vol. 2, pp. 115-123, 1994.
- [25] W. Huang and F. Lombardi, "An Approach to Testing Programmable/Configurable Field Programmable Gate Arrays," *Proc. IEEE VLSI Test Symp.*, pp. 450-455, 1996.
- [26] W. Huang, F. Meyer, X. Chen, and F. Lombardi, "Testing Configurable LUT-Based FPGAs," *IEEE Trans. on VLSI Systems*, Vol. 6, No. 2, pp. 276-283, 1998.
- [27] T. Inoue and H. Fujiwara, "Universal Fault Diagnosis for Lookup Table FPGAs," *IEEE Design & Test of Computers*, Vol. 15, No. 1, 1998.
- [28] C. Jordan and W. Marnane, "Incoming Inspection of FPGAs," *Proc. European Test Conf.*, pp. 371-377, 1993.
- [29] J. Kelly and P. Ivey, "Defect Tolerant SRAM Based FPGAs," *Proc. Intn'l. Conf. on Computer Design*, pp. 479-482, 1994.
- [30] V. Kumar, A. Dahbura, F. Fischer, and P. Juola, "An Approach for the Yield Enhancement of Programmable Gate Arrays," *Proc. IEEE Int'l Conf. on Computer Aided Design*, pp. 226-229, 1989.
- [31] J. Lach, W. Mangione-Smith, and M. Potkonjak, "Low Overhead Fault-Tolerant FPGA Systems," *IEEE Trans. on VLSI Systems*, Vol. 6, No. 2, pp. 212-221, 1998.
- [32] J. Lach, W. Mangione-Smith, and M. Potkonjak, "Algorithms for Efficient Runtime Fault Recovery on Diverse FPGA Architectures," *Proc. Intn'l. Symp. on Defect and Fault Tolerance In VLSI Systems*, 1999.
- [33] V. Lakamraju and R. Tessier, "Tolerating Operational Faults in Cluster Based FPGAs," *Proc. ACM Intn'l. Symp. on FPGAs*, pp. 197-194, 2000.
- [34] F. Lombardi, D. Ashen, X. Chen, and W. Huang, "Diagnosing Programmable Interconnect Systems for FPGAs," *Proc. ACM/SIGDA Intn'l Symp. on FPGAs*, pp. 100-106, 1996.
- [35] Lucent Technologies, <http://www.micro.lucnet.com/micro/fpga>.
- [36] N. Mahapatra and S. Dutt, "Efficient Network Flow Based Technique for Dynamic Fault Reconfiguration in FPGAs," *Proc. Fault Tolerant Computing Symp.*, pp. 122-129, 1999.
- [37] D. Mange, M. Sipper, A. Stauffer, and G. Tempesti, "Toward Robust Integrated Circuits: The Embryonics Approach," *Proc. of the IEEE*, Vol. 88, No 4, pp. 516-541, April 2000.
- [38] D. Mange, M. Sipper, A. Stauffer, and G. Tempesti, "Toward Self-Repairing and Self-Replicating Hardware: The Embryonics Approach," *Proc. 2nd NASA/DoD Workshop on Evolvable Hardware*, pp. 205-214, July 2000.

- [39] E. McCluskey, "Verification Testing - A Pseudoexhaustive Test Technique," *IEEE Trans. on Computers*, Vol. 33, No. 6, pp. 541-546, 1984.
- [40] McDonald, B. Philhower, and H. Greub, "A Fine Grained, Highly Fault Tolerant System Based on WSI and FPGA Technology," *More FPGAs*, W. Moore and W. Luk (eds.), Abingdon EE & CS Books, pp. 114-126, 1991.
- [41] H. Michinishi, et al., "A Test Methodology for Interconnect Structures of LUT-Based FPGAs," *Proc. Asian Test Symp.*, pp. 68-74, 1996.
- [42] J. Narasimhan, K. Nakajima, C. Rim, and A. Dahbura, "Yield Enhancement of Programmable ASIC Arrays by Reconfiguration of Circuit Placements," *IEEE Trans. on CAD*, Vol. 13, No. 8, pp. 976-986, 1994.
- [43] J. Narasimhan, C. Rim, K. Nakajima, and A. Daubura, "Yield Enhancement of Wafer Scale Integrated Arrays," *Proc. IEEE Conf. on Wafer Scale Integration*, pp. 178-184, 1991.
- [44] M. Renovell, J. Portal, J. Figueras, and Y. Zorian, "SRAM-based FPGA: Testing the LUT/RAM Modules," *Proc. IEEE Intn'l. Test Conf.*, pp. 1102-1111, 1998.
- [45] M. Renovell, J. Portal, J. Figueras, and Y. Zorian, "Testing the Interconnect of RAM-Based FPGAs," *IEEE Design & Test of Computers*, Vol. 15, No. 1, 1998.
- [46] M. Renovell, J. Portal, J. Figueras, and Y. Zorian, "SRAM-based FPGA: Testing the Embedded RAM Modules," *Journal of Electronic Testing: Theory and Application (JETTA)*, Vol. 14, No. 1, 1999.
- [47] K. Roy and S. Nag, "On Routability for FPGAs Under Faulty Conditions," *IEEE Trans. on Computers*, Vol. 44, pp. 1296-1305, 1995.
- [48] N. Shnidman, W. Mangione-Smith, and M. Potkonjak, "On-line Fault Detection for Bus-Based Field Programmable Gate Arrays," *IEEE Trans. on VLSI Systems*, Vol. 6, No. 4, pp. 656-666, 1998.
- [49] L. Shombert and D. Siewiorek, "Using Redundancy for Concurrent Testing and Repairing of Systolic Arrays," *Proc. Fault-Tolerant Computing Symp.*, pp. 244-249, 1987.
- [50] B. Skaggs, "On-Line Built-In Self-Test and Diagnosis of Field Programmable Gate Array Logic Blocks," MSEE Thesis, Dept. of Electrical Engineering, Univ. of Kentucky, 2000.
- [51] "Standard Test Access Port and Boundary-Scan Architecture," *IEEE Standard P1149.1*, 1990.
- [52] A. Steininger and P. Scherrer, "On the Necessity of On-Line BIST in Safety Critical Applications," *Proc. Fault-Tolerant Computing Symp.*, pp. 208-215, 1999.
- [53] C. Stroud, P. Chen, S. Konala, and M. Abramovici, "Evaluation of FPGA Resources for Built-In Self-Test of Programmable Logic Blocks," *Proc. ACM/SIGDA Intn'l. Symp. on FPGAs*, pp. 107-113, 1996.
- [54] C. Stroud, M. Lashinsky, J. Nall, J. Emmert, and M. Abramovici, "On-Line BIST and Diagnosis of FPGA Interconnect Using Roving STARs," *Proc. IEEE Intn'l On-Line Test Workshop*, 2001.
- [55] C. Stroud, E. Lee, and M. Abramovici, "BIST-Based Diagnostics of FPGA Logic Blocks," *Proc. IEEE International Test Conf.*, pp. 539-547, 1997.

- [56] C. Stroud, S. Konala, P. Chen, and M. Abramovici, "Built-In Self-Test for Programmable Logic Blocks in FPGAs (Finally, A Free Lunch: BIST Without Overhead!)," *Proc. IEEE VLSI Test Symp.*, pp. 387-392, 1996.
- [57] C. Stroud, S. Wijesuriya, C. Hamilton, and M. Abramovici, "Built-In Self-Test of FPGA Interconnect," *Proc. IEEE Intn'l. Test Conf.*, pp. 404-411, 1998.
- [58] W. Tsu, K. Macy, A. Joshi, R. Huang, J. Wawrzynek, and A. DeHon, "High-Speed, Hierarchical Synchronous Reconfigurable Array," *Proc. ACM Intn'l. Symp. on FPGAs*, pp. 125-134, 1999.
- [59] A. van de Goor, *Testing Semiconductor Memories: Theory and Practice*, John Wiley & Sons, 1991.
- [60] S.-J. Wang and T.-M. Tsai, "Test and Diagnosis of Faulty Logic Blocks in FPGAs," *Proc. IEEE Intn'l. Conf. on Computer Aided Design*, 1997.
- [61] Xilinx, Inc., <http://www.xilinx.com>.
- [62] C. Zeng, N. Saxena and E. McCluskey, "Finite State Machine Synthesis with Concurrent Error Detection," *Proc. IEEE Intn'l. Test Conf.*, pp. 672-679, 1999.

List of Acronyms

ACS	Adaptive computing system
BIST	Built-in self-test
BUT	Block under test
CAD	Computer-aided design
CED	Concurrent error detection
CIP	Configuration interconnect point
CPLD	Complex programmable logic devices
FABRIC	Fault-bypassing roving configuration
FF	Flip-flop
FPGA	Field-programmable gate array
FT	Fault-tolerant
GIU	Graphical user interface
H-STAR	Horizontal self-testing area
I/O	Input/Output
LUT	Look-up table
MUX	Multiplexer
ORA	Output response analyzer
ORCA	Optimized reconfigurable cell array
PLB	Programmable logic block
PUB	Partially usable block
RAM	Random access memory
RTR	Run-time reconfiguration
SRAM	Static random access memory
STAR	Self-testing area
TAP	Test access port
TPG	Test-pattern generator
TREC	Test and reconfiguration controller
V-STAR	Vertical self-testing area
WUT	Wire under test