Using Embedded FPGAs for SoC Yield Improvement

Miron Abramovici
Agere Systems
Murray Hill, NJ 07974
miron@agere.com

Charles Stroud
University of North Carolina
Charlotte, NC 28223
cestroud@uncc.edu

Marty Emmert
Wright State University
Dayton, OH 45435
marty.emmert@wright.edu

Abstract: In this paper we show that an embedded FPGA core is an ideal host to implement infrastructure IP for yield improvement in a bus-based SoC. We present methods for testing, diagnosing, and repairing embedded FPGAs, for which complete testability is achieved without any area overhead or performance degradation. We show how an FPGA core can provide embedded testers for other cores in the SoC, so that cores designed to be tested with external vectors can be tested with BIST, and the entire SoC can be tested with a low-cost tester.

Categories and Subject Descriptors: B.8.1 [Performance and Reliability]: Reliability, Testing, and Fault-Tolerance

General Terms: Algorithms, Design, Reliability

1. Introduction

Reusing embedded IP cores in System-on-Chip (SoC) design has become the primary means of improving the productivity of designers faced with very large and complex circuits. One problem in manufacturing a SoC with millions of transistors using deep-submicron technologies (0.13μm and below), is an increase in the probability of defects in silicon, which results in decreasing manufacturing yield. The economic consequences of low yield for expensive SoC devices can be disastrous. To effectively deal with this increased defect density, we need efficient methods for fault detection, location, and fault tolerance implemented on-chip. Typically, such methods are implemented by embedded IP blocks referred to as *infrastructure IP*. In this paper we will show that an embedded FPGA core is an ideal vehicle for infrastructure IP.

But an embedded FPGA is also a very useful functional SoC component. A SoC design where at least part of the user-defined logic is implemented in an embedded FPGA provides the following benefits:

1) Usually fixing logic design errors in an ASIC requires a "respin" to change a set of manufacturing masks. The cost of a respin is around $1M, and it also causes a 2-to-6 months hit in the time-to-market. In contrast, a fix that can be done in the FPGA can be completed in one day.
2) The embedded FPGA may be reused for different functions at different times. For example, a cell phone may work with different protocols (CDMA, GSM, etc.) in different geographical areas, and the logic implementing each protocol can be downloaded in the FPGA on a demand basis.
3) Although FPGA clock speeds are slower than ASIC clock speeds, the massive parallelism of an FPGA can be exploited to implement many algorithms (such as image processing, signal processing, cryptography, etc.) at higher performance levels than on μPs or DSPs. Usually such FPGA implementations consume less power than μP or DSP implementations, because they can be run at lower clock frequency.
4) The same SoC can be used as a platform for different products, or for different versions of the same product, which are created by different FPGA configurations.
5) A good strategy is to use the embedded FPGA to implement protocols and algorithms likely to change in the future (for example, hardware support for a standard not yet officially adopted). In this way, future changes will be easy to implement without changing the SoC.
6) The embedded FPGA can support remote and Internet-based field-upgrades, which may be very important features for system manufacturers.

2. Using an FPGA Core for Infrastructure IP

In addition to the functional features described above, an embedded FPGA can also provide great benefits in implementing infrastructure IP. SoC yield-improvement requires detecting faults, locating them, and repairing them so that the SoC will work correctly in their presence. Methods for detecting, locating, and repairing faults in RAMs are described in [9]**.** The problems we address in this paper are detecting, locating, and repairing faults in an embedded FPGA; some of our previous work dealt with the same problems in the context of on-line testing [4], and some of the solutions developed for on-line testing can be directly applied to, or adapted for, manufacturing testing of an embedded FPGA. Another contribution of this paper is novel ways to use an embedded FPGA to test other cores in the SoC.

We assume a bus-based SoC architecture, where the embedded cores are connected by a common system bus. The FPGA core can become the bus master. The SoC is tested by automatic test equipment (ATE) for manufacturing testing, or by a maintenance processor for in-system testing. In either case, the tester also controls the configuration process for the embedded FPGA core via the boundary-scan access mechanism of the SoC [22]; the same mechanism can be used both for manufacturing test and for in-system test. Various configurations for the FPGA are stored on disk. We will refer to the external tester as *test and reconfiguration controller* (TREC). If BIST is used within the SoC, TREC initiates the internal

This material is based upon work supported in part by the DARPA ACS program under contract F33615-98-C-1318, and by Lucent Technologies.

DAC 2002, June 10-14, 2002, New Orleans, Louisiana, USA.

BIST controller and processes the results. TREC also performs diagnosis and fault tolerance functions that will be described in the following sections. Later we will show that the embedded FPGA allows TREC to be a low-cost tester (possibly PC-based).

Our basic principle for using reconfigurable logic for infrastructure IP is "***create the infrastructure only when needed***." Hence instead of having the infrastructure for test always present in the circuit, we download it in the embedded FPGA only when we want to test the SoC. In most cases, this infrastructure will be used only once for manufacturing testing, so having it permanently in the system is clearly wasteful (we can recreate it for maintenance testing, if and when needed). Even if the FPGA implements user-defined system functions during normal operation, this logic is not needed when the SoC is under test and may be temporarily replaced by test logic. This is a generalization of our "free lunch" FPGA BIST [24]. Typically, logic BIST for ASICs introduces about 20% area overhead and some performance degradation. In contrast, BIST for FPGAs is a "free lunch" with *no area overhead or delay penalties*, since the entire FPGA is configured only for BIST, and all the BIST logic disappears when the FPGA is no longer under test.

Our basic principle can have surprising consequences. For example, the proposed IEEE P1500 Standard for Embedded Core Test [16] specifies that embedded cores should provide a wrapper for test purposes. An embedded FPGA core, however, can safely ignore this requirement, since the wrapper can be downloaded in the FPGA only during the SoC test.

Taking advantage of FPGA features, such as reconfigurability and regular structure, allows FPGA testing to achieve features that are impossible for ASICs: for example, FPGAs can be tested pseudo-exhaustively [20] and faults can be very precisely located. Pseudo-exhaustive testing results in *practically complete fault coverage* without the use of computationally expensive ASIC test tools, such as fault simulation and automatic test-pattern generation (ATPG).

Two types of fault tolerance exist for FPGAs. The first, *manufacturer-level fault-tolerance*, relies on providing spare resources that can be used to replace the faulty ones. For example, some FPGAs have a spare column and a hardware mechanism that allows any column to be replaced by an adjacent one [8]; the replacement for each column is controlled by a fuse. After the faulty column is diagnosed, all columns between the faulty one and the spare are "shifted" by one position, so that the faulty one is completely avoided, and the change is invisible to the outside world. The costs of this approach - the additional area needed for the spare resources and the performance penalty introduced by the column selection hardware - are paid by all chips, including the non-defective ones.

The second approach, *user-level fault-tolerance*, avoids these costs since it relies on the spares naturally available in an FPGA, where any application uses only a subset of the existing resources. Knowing the user circuit to be implemented in the FPGA, and the exact location of the faulty resources, one can modify the existing implementation (mapping, placement, routing) to replace the faulty resources with fault-free spares. For example, this approach allowed the Teramac custom computer [10] to be build from 864 FPGAs, out of which 75% were defective. Note that configuration data for each faulty FPGA should be separately maintained. Unlike manufacturing-level fault tolerance, which is independent of the target circuit, user-level fault tolerance can ignore any fault that does not affect the circuit. User-level fault tolerance requires testing and diagnosis of FPGAs to be done by users, and this may be unacceptable for off-the-shelf FPGAs typically tested by their manufacturers.

In contrast, an embedded FPGA core is tested only as part of an newly fabricated SoC, and the SoC designer is also responsible for its testing. In this environment, user-level fault-tolerance is clearly the right solution for yield improvement.

The remainder of the paper is organized as follows. Section 3 explains why a commonly used approach - testing the user's circuit - is not good for FPGA test. Section 4 presents techniques for testing and diagnosing logic cell faults in an embedded FPGA. Section 5 describes techniques for detecting and locating faults in the programmable interconnect network. Section 6 shows how to use an embedded FPGA to create the infrastructure for testing other cores. Section 7 presents fault tolerance methods that allow the embedded FPGA to work correctly in the presence of the located faults. Section 8 concludes the paper.

3. A Wrong Approach: Testing User's Circuit

Many FPGA design and test flows generate tests for the circuit implemented in the FPGA using ATPG tools for ASICs. However, such an approach will not completely test the FPGA hardware. The first problem is testing of a configuration multiplexer (MUX), which is a commonly used hardware mechanism to select 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 1a, assume that we set the configuration bit *S* to 0 to connect subcircuit *C0* to *X*. Then subcircuit *C1* disappears from the circuit model seen by the user. This is correct from a design viewpoint, because the value *V1* produced by *C1* can no longer affect the MUX output 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 $\overline{V}$.

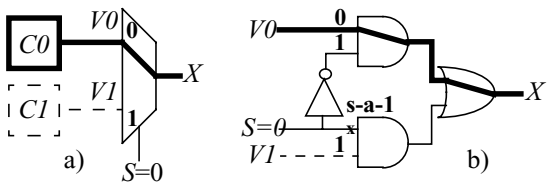

Figure 1. Configuration MUX

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 [1]). Thus in Figure 1a, 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. Hence the testing of the MUX may not be complete. For example, the s-a-1 fault in the gate-level MUX model in Figure 1b 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.

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 incorrectly ignored. Since an FPGA has tens of thousands of configuration MUX structures, such a test is likely to have a poor quality.

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.

A second problem with testing only the user's circuit is that spare resources are never tested, and when the need arises to replace a faulty resource with a spare one, we cannot be sure that the spare is fault-free. Our approach is to test the entire FPGA and to construct tests that are independent of the applications to be implemented in the device.

4. Testing the Logic Cells

4.1 BIST for Logic Cells

The embedded FPGA is composed of an array of programmable logic cells connected by programmable interconnect resources. Usually the logic and the interconnect of an FPGA are separately tested. The goal of logic tests is to detect any faults (single or multiple) affecting the operation of a cell, and also any combination of multiple faulty cells.

Figure 2 illustrates our logic BIST architecture [1]. Some of the cells are configured as *test-pattern generators* (TPGs) and *output-response analyzers* (ORAs), while another group of cells are configured as *blocks under test* (BUTs). The two TPGs provide identical test patterns to alternating rows of identically configured BUTs. The outputs of the BUTs are compared by the ORAs which latch any mismatches. The *Pass/Fail* result flip-flops of all ORAs are connected in a scan chain. The BUTs are then repeatedly reconfigured so that they are tested in all of their modes of operation. Each reconfiguration of the embedded FPGA to test a different cell mode of operation is referred to as a *test phase*. A *test session* is a collection of test phases that completely test the BUTs in all of their modes of operation. The number of test phases is a function of the cell architecture. Once the BUTs have been tested, the roles of the cells are reversed, so that in the next test session the previous BUTs become TPGs or ORAs, and vice versa. Figure 3 shows the cell functions in the two test sessions for an 8×8 FPGA. Since half of the cells are BUTs during each test session, only two test sessions are needed to test all cells in the embedded FPGA.

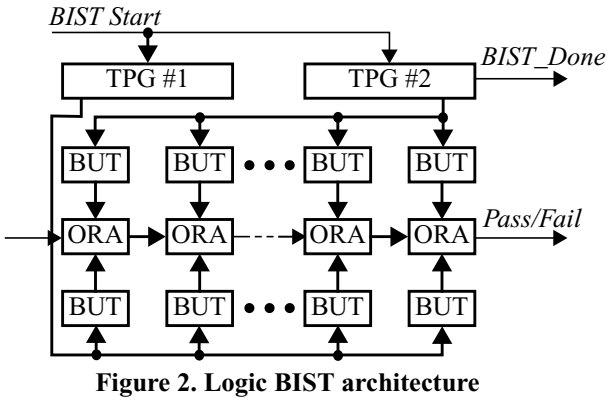

Figure 2. Logic BIST architecture

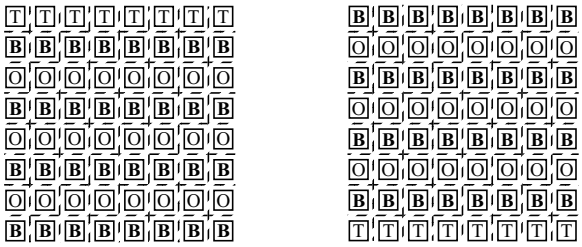

Figure 3. Cell functions for the two test sessions

In every test phase, TREC performs the following steps: 1) reconfigure the embedded FPGA with a BIST configuration retrieved from the disk, 2) initiate the BIST sequence, and 3) read the *Pass/Fail* results from the ORAs. The downloading of the BIST configuration and the control of the BIST process are done via the boundary-scan access mechanism of the SoC [22]. Since the test application time for any phase is dominated by the embedded FPGA reconfiguration time, an important goal of the BIST approach is to minimize the total number of test configurations.

Figure 4 illustrates the typical structure of a cell, consisting of a memory block that can function as a look-up table (LUT) or RAM, several flip-flops, and multiplexing output logic. The LUT/RAM block may also contain special-purpose logic for arithmetic functions (counters, adders, multipliers, etc.) The RAM may be configured in various modes of operation (synchronous, asynchronous, single-port, dual-port, etc.). The flip-flops can also be configured as latches, and may have programmable clock-enable, preset/clear, and data selector functions. Our TPG applies exhaustive patterns to every subcircuit of a cell for each one of its modes of operation, except for the RAM modes, where the memory block is checked with RAM March tests which cover most RAM-specific faults. Exhaustive testing of every subcircuit is feasible since their number of inputs is reasonably small. This results in ***practically complete cell fault coverage*** without explicit fault model assumptions and without fault simulation.

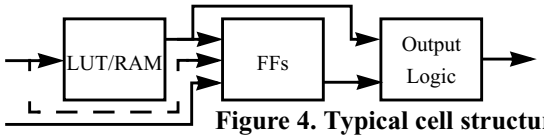

Figure 4. Typical cell structure

Although there are situations that make detection of multiple faulty cells more complicated (a pair of compared BUTs have equivalent faults and do not cause a mismatch; a faulty ORA does not detect a mismatch; a faulty TPG skips patterns that detect a faulty BUT), *almost any combination of multiple faulty cells is guaranteed to be detected* by our two test sessions. We have shown that the conditions that would allow a group of faulty cells to escape detection are extremely restrictive, so that they are very unlikely to occur in practice [1].

4.2 Logic Cell Diagnosis

Figure 5 illustrates a test session where two faulty BUTs (denoted by black squares) are detected. Errors are observed at the ORAs bordering the faulty cells (shaded squares). Clearly, the set of failing ORAs is unique for every faulty BUT, and therefore it can be used to locate single faulty cells. (As a consistency check, the two failing ORAs observing the same BUT must fail in the same phases [1].)

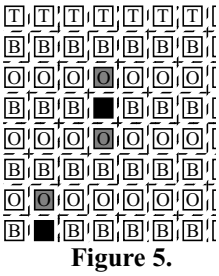

Figure 5.

Actually in Figure 5 we are dealing with a group of two faulty cells, but because they are observed at disjoint ORAs, they do not interact. Locating BUTs that are observed at common ORAs is more difficult, but we have developed diagnostic procedures [1] that can locate almost any combination of multiple faulty cells likely to occur in practice. These procedures rely on analyzing the sets of failing test phases obtained at different ORAs, and by applying additional BIST configurations when needed.

In addition to diagnosing the faulty cells, we can also identify the faulty subcircuits (LUTs, flip-flops) or the defective modes of operation (e.g., count-up, multiply) within a faulty cell [2].

Although our tests are application-independent, we can take advantage of knowing the circuits to be implemented in the embedded FPGA. Specifically, TREC may avoid applying additional diagnostic configurations when the suspected cells are not used in any of the target circuits. However, accurate diagnosis of unused cells may be required later to allow these cells to replace other defective cells for fault tolerance.

If a configuration memory bit that controls a resource in a logic cell is stuck, this fault is detected by the tests for the corresponding resource. However, the faulty resource and the stuck fault in its controlling configuration bit cannot be distinguished, unless the configuration memory has a readback feature that can detect the stuck bit.

5. Testing the Interconnect Network

5.1 BIST for Interconnect

The programmable interconnect network consists of wire segments of different lengths that can be connected via programmable switches referred to as *configurable interconnect points* (CIPs). Wire segments connecting non-adjacent cells form global routing resources, while local routing resources connect a cell to global routing resources or to adjacent cells. The basic CIP structure consists of a transmission gate controlled by a configuration memory bit (Figure 6a). There are three types of CIPs which we refer to as the *cross-point CIP* (Figure 6b), the *break-point CIP* (Figure 6c), and the *multiplexer* (*MUX) CIP* (Figure 6d) [18]. While a cross-point CIP connects wire segments located in disjoint planes (a horizontal segment with a vertical one), a break-point CIP connects two wire segments in the same plane. The MUX CIP comes in two varieties: decoded and non-decoded. A decoded MUX CIP is a group of 2^k cross-point CIPs sharing a common output wire and controlled by k configuration bits, such that the input wire being addressed by the configuration bits is connected to the output wire. A non-decoded MUX CIP contains a configuration bit controlling each transmission gate; here only one of the configuration bits is active for any configuration. There is also a *compound CIP* (Figure 6e), which is a combination of four cross-point and two break-point CIPs, each separately controlled by a configuration bit [31].

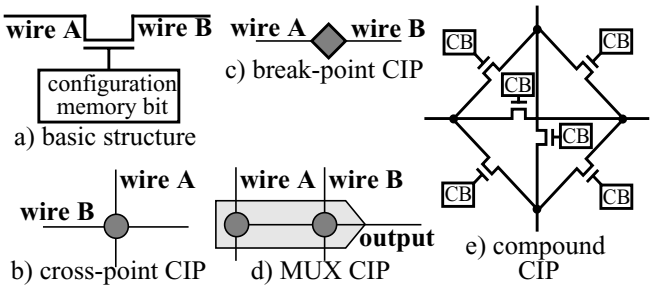

Figure 6. Configurable interconnect points

The fault model we consider is typical for interconnect: CIPs stuck-closed (stuck-on) and stuck-open (stuck-off), wires stuck at 0 or 1, open wires, and shorted wires. 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 data books) to determine adjacent "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. This treatment makes our method layout-independent.

Detecting a CIP stuck-open (closed) fault also detects the stuck-at-0 (1) fault in the configuration memory bit that controls that CIP. However, most previous work in FPGA interconnect testing has ignored the problem of shorts involving configuration memory bits. This is understandable, because the configuration memory does not appear in any model available to users. Nevertheless, this is an important class of faults, since the configuration memory is physically distributed across the entire chip, so that each bit is close to the resource it controls. Hence shorts between configuration bits (or the wires that transmit them) and the wire segments in the interconnect network are quite likely and cannot be ignored. We will analyze the effect of such faults at the end of Section 5.

Figure 7 illustrates the concepts of the interconnect BIST. We configure subsets of routing resources (wire segments

and CIPs) to form two groups of *wires under test* (WUTs) that receive exhaustive test patterns from a TPG, while the values at the other end of the WUTs are compared by an ORA [26]. A WUT consists of several wire segments connected by closed CIPs. To check local routing resources, WUTs may also go through cells configured to pass their inputs directly to outputs. In Figure 7, the WUTs are shown by bold lines, and the activated (closed) CIPs are shown in gray, while the open CIPs are white. To test the open CIPs for stuck-on faults, when the TPG drives a value v on the WUTs (0 in Figure 7), $\bar{v}$ must be applied to the wire segments on the opposite side of the open CIPs.

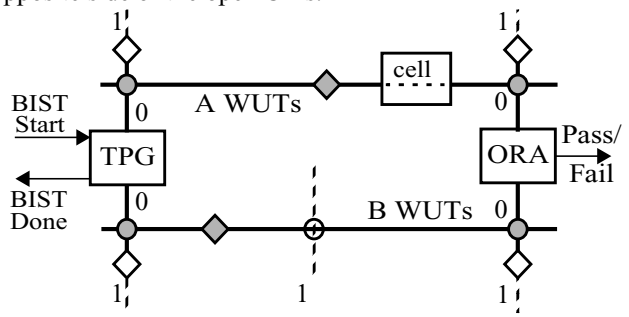

Figure 7. Interconnect BIST architecture

The TPG applies exhaustive patterns to the WUTs, all possible 2^n test vectors to every group of n WUTs. For reasonably small n, the application time for 2^n test vectors is still negligible compared to the reconfiguration time of the FPGA. The TPG also provides test patterns to the opposite sides of open CIPs that isolate the WUTs from the rest of the interconnect.

If the two compared sets of WUTs have identical (or equivalent) faults, then no mismatch will be detected at the ORA [15]. For example, each set of WUTs could have a different CIP stuck-open in its i^{th} wire. To overcome this problem, we test every set of WUTs (at least) twice, each time being compared with a different set of WUTs [25].

The basic routing BIST architecture shown in Figure 7 is implemented within *self-testing areas* (STARs), where each STAR spans two columns or two rows of the embedded FPGA to test vertical or horizontal interconnect resources, respectively [4][5]. Figure 8 illustrates the structure for testing the horizontal interconnect. Testing within STARs is done concurrently.

While most of the interconnect resources are tested either in an horizontal or in a vertical STAR, testing the cross-point CIPs connecting global horizontal and vertical busses requires both a vertical and an horizontal STAR, where the cross-point CIPs at the intersection of the STARs are under test (Figure 9).

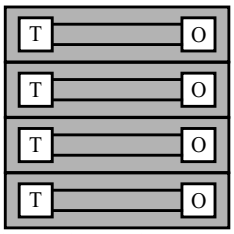

Figure 8. Horizontal interconnect testing

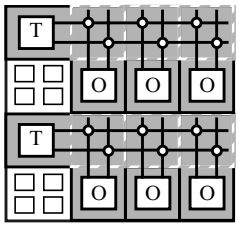

Figure 9. Global cross-point CIP test

5.2 Interconnect Fault Diagnosis

Our diagnosis procedure begins by analyzing the BIST results to narrow the search area, and proceeds only in the failing STARs. We use an adaptive strategy, where the next diagnostic phase is determined based on the results obtained so far. Figure 10 illustrates several diagnosis techniques (failing ORAs are darkened).

In Figure 10a, after the first phase we do not know which group of WUTs is faulty. During the second test phase, the WUTs are swapped between TPGs, and the faulty group can be identified by checking whether the failure moves to the other ORA or stays at the first ORA.

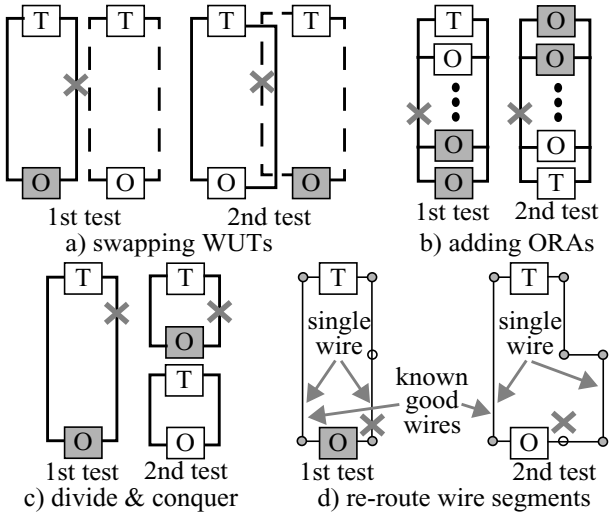

Figure 10. Diagnostic configurations

Figure 10b depicts one technique used to identify the faulty region in a set of WUTs. We add several ORAs to observe the WUTs at different points and use two phases with opposite directions of the test pattern flow through WUTs. Faults are located between the highest failing ORA in the first phase and the lowest failing ORA in the second phase. An alternative technique is to divide the WUTs into pairs of shorter WUTs and retest (Figure 10c). This "divide and conquer" method can be recursively repeated to obtain a very high diagnostic resolution.

Identifying the faulty WUT in a group can be accomplished by either testing one wire at a time (and comparing with a known-good wire), or by making different ORA connections with the faulty set of WUTs and observing the movement of ORA failures (a similar idea to that of Figure 10a). The faulty wire segment or CIP is then identified by re-routing to eliminate a segment (or CIP) at a time during the subsequent tests (Figure 10d).

Diagnostic phases can be either precomputed and stored on disk, or generated "on the fly" by TREC. In general, the diagnostic phases illustrated in Figure 10a through 10c can be precomputed, while the detailed diagnostic phases implied

by Figure 10d would be generated "on the fly" for most FPGA interconnect architectures.

A tool like JBITS [29] that can efficiently generate configuration streams without relying on CAD tools, would be ideal to use during testing. Since no such tools are commercially available, we rely on commercially available place-and-route tools used incrementally, followed by configuration-generation software; these may run for several minutes to generate multiple diagnostic phases, but this time could be acceptable if TREC is a low-cost tester. A limit on the time allowed for computing diagnosis configurations can be established based on factors such as the cost of a SoC, the production volume, the current yield, the probability of repairing the defective SoC given the number of faults found so far, the target resolution, etc. The use of precomputed diagnostic phases in conjunction with an adaptive diagnostic procedure minimizes the total time spent on TREC, while maintaining defect-tolerant capabilities.

To illustrate the diagnostic resolution that can be obtained with these techniques, we will assume a single fault in a typical configuration of interconnect resources illustrated in Figure 11a. Here segment *X* is bounded by break-point CIPs that control its connections to *Y* and *Z*, while segment *W* shares no CIPs with *X*. For example, if a cross-point CIP is stuck-open (Figure 11b), this is the only condition that would allow the vertical and the horizontal segments connected by the that CIP to be driven to complementary values. A similar reasoning identifies an open wire segment (Figure 11c) In these cases, diagnostic resolution is to a CIP or a segment. However, we cannot distinguish between an open in a segment and a stuck-open fault in its adjacent break-point CIP (Figure 11d). Figure 11d also shows another pair of equivalent faults - a short between two segments separated by a break-point CIP and that CIP being stuck-on. On the other hand, shorts between two segments that do not share a CIP, such as *W* and *X* in Figure 11e, can be diagnosed by checking that each segment is fully functional while the other segment it is shorted to is unused, but when both segments are driven to opposite values, then we detect a failure on at least one of them.

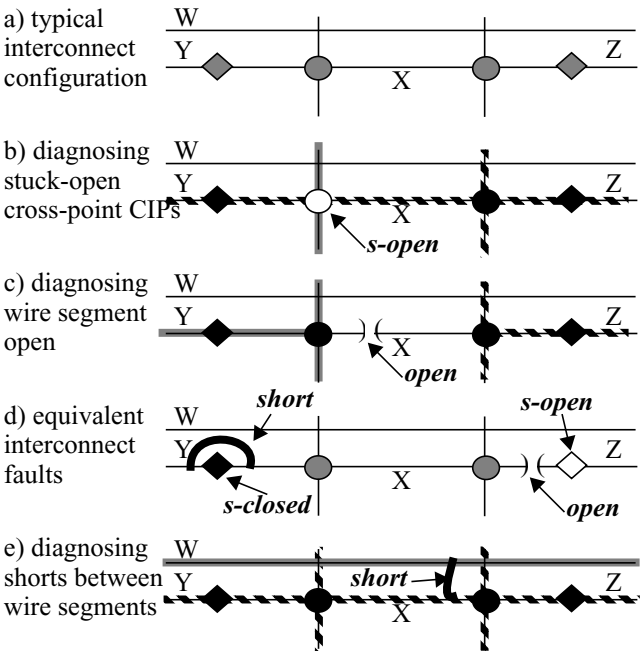

Figure 11. Diagnostic resolution examples

For multiple faults in the same area, there are cases where we will not always be able to identify the faulty resources among a set of suspects. These cases arise when some faults interfere with the routing configurations needed to diagnose other faults. In such cases, we will "play it safe" by labeling the entire set of suspects as faulty, so that all of them will be bypassed by fault tolerant techniques.

To illustrate the problem of detecting and diagnosing shorts with configuration memory bits, consider a short between a configuration bit *B* and an interconnect segment *S,* and let us assume that *B* controls a CIP *C*. Note that the value *b* of *B* is constant during a phase. This short may be detected in any configuration where *S* is used and at least once is set to value $\overline{b}$. The detection will occur in one of the two cases: 1) the short creates a wrong value on *B* and *C* is under test; 2) the short creates the wrong value on *S* and *S* is under test.

The problem is that *C* may be under test in different configurations, and the value of *B* may be affected in only some of them. Similarly, *S* may be tested in different configurations, and the value of *B* may affect only some of them. This behavior would seem to be that of an intermittent fault, as the same resource looks like it fails only some of the configurations where it is tested. So an interconnect diagnosis procedure that looks for consistency in all cases is doomed to fail. The practical solution in such situations is to label all the suspected resources as faulty. Avoiding all the potential defects is a reasonable strategy for yield enhancement.

6. Testing Other Cores with the FPGA Core

The embedded FPGA can also be used to test any other embedded core connected to the same system bus. We will separately analyze *BIST-cores*, whose testing is based on in-core BIST, and externally-tested-(*ET)-cores*, which rely on vectors external to the core.

6.1 BIST-Cores

For a BIST-core, its ***BIST logic may be moved from the core to the embedded FPGA***, as illustrated in Figure 12. (Similar techniques have been described for using an FPGA installed on a board to test other devices on the board

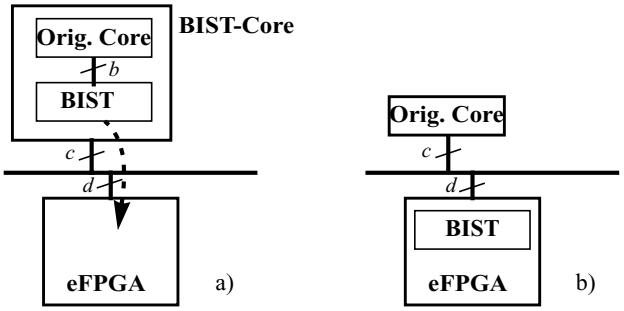

Figure 12. Relocating the BIST logic

[22][27].) In practice, this means that the core providers do not have to implement BIST in embedded cores; instead, they provide the BIST logic as an additional IP to be implemented in the FPGA core. Several conditions must be satisfied to make this transformation possible. First, the BIST logic should not be intrusive, that is, it should be easy to separate from the rest of the logic. Let b, c, and d denote, respectively, the number of connections between the BIST logic and the original core (without BIST), between the core (with BIST) and the system bus, and between the embedded FPGA and the system bus. The relocation of the BIST logic is possible if $b \le c$ and $b \le d$. Another condition is for the FPGA to supply test patterns at the speed required to test the core. Of course, the BIST logic implemented in the core would be faster than its FPGA implementation, but this speed may be higher than the one used in normal operation, when the core is accessed over the system bus. So if the FPGA can apply the test at the same speed the core is accessed over the system bus, and if the bus width conditions are also satisfied, the BIST logic may be relocated in the FPGA.

We will illustrate this scheme for testing an embedded RAM core. Figure 13 illustrates a typical RAM BIST architecture. *Data In*, *Address*, *Control*, and *Data Out* are included in the system bus. During BIST, the inputs *Data In*, *Address*, and *Control* are replaced with the signals produced by an integrated BIST controller and TPG, which generate a March test for the RAM block. The controller/TPG also produces the expected output responses to be compared with the RAM output data by the ORA, which latches any mismatches observed to produce a failure indication at the end of the BIST sequence. (Note that separate tests are needed to check the connections between the RAM block and the I/Os of the RAM core.)

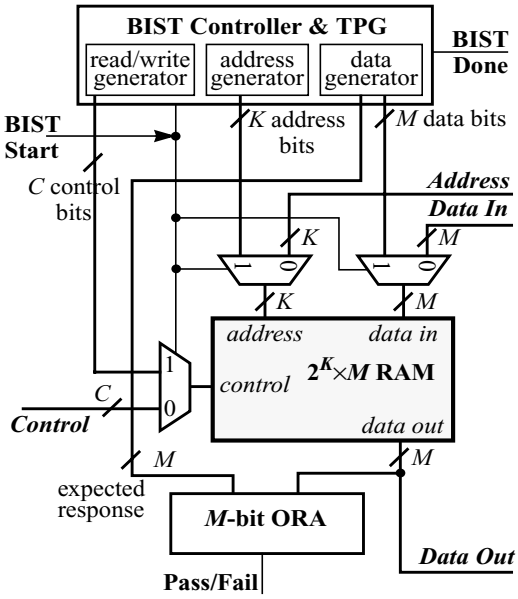

Figure 13. Typical RAM BIST architecture

Using the embedded FPGA, we can follow the scheme in Figure 12, remove all the BIST logic from the RAM core, and download the BIST controller/TPG and the ORA logic in the FPGA only when we need to test the RAM. This results in several advantages. First, the RAM core is smaller, being reduced to only the shaded RAM block, and faster, since we no longer need multiplexers to select between system and BIST data (this selection is now done by the system bus). Second, when the BIST logic is in the same core with the RAM (Figure 13), we also need to test the BIST logic to make sure that it will correctly test the RAM; this is no longer necessary when the RAM BIST logic is implemented in the FPGA core, because the self-test of the embedded FPGA (described in the previous sections) completely tests all its resources. Third, while defects in the BIST logic of a RAM core cannot be repaired and will likely cause the entire SoC to be discarded, defects in the embedded FPGA core can be easily tolerated (see Section 7).

If we have several RAM cores in the SoC, one drawback of this approach is that it can test only one RAM at a time, because the FPGA can observe the output of only one core at a time. In contrast, with the BIST logic in the same core with the RAM, all embedded RAMs may be tested in parallel, provided that the power dissipation constraints of the SoC are not violated.

A compromise solution, illustrated in Figure 14, moves only part of the BIST logic in the embedded FPGA core, while allowing several RAMs to be tested concurrently. We move only the BIST controller/TPG in the embedded FPGA, and we leave the ORA and a separate data generator block in the RAM core to produce only the expected data. The FPGA provides the same *Data In*, *Address*, and *Control* inputs to all RAMs under test. All embedded RAMs can be tested concurrently, since here results analysis is independently done within each RAM core. This scheme still avoids the BIST multiplexers which cause the performance degradation.

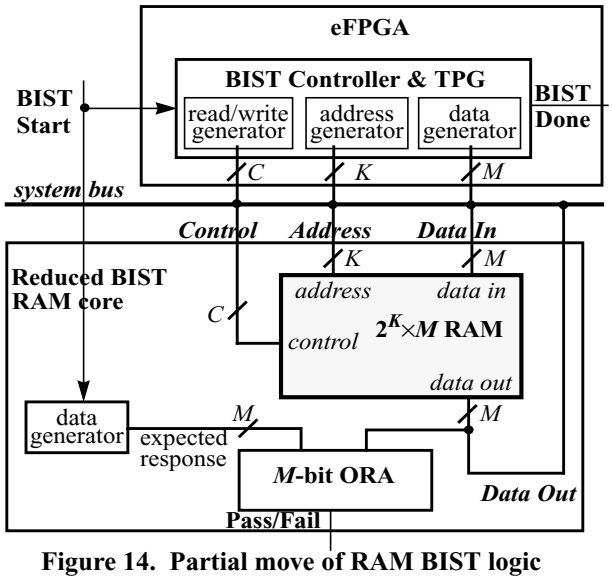

Figure 14. Partial move of RAM BIST logic

Similarly, the embedded FPGA core can be used in turn to implement (at least part of) the BIST logic for most embedded BIST-cores that are connected to the system bus. Moving the BIST logic in the embedded FPGA requires analyzing the trade-offs involving the area overhead and the performance

penalties introduced by in-core BIST, the total test time, and the SoC power constraints. This analysis is the responsibility of the SoC designer; to make the choice possible, the core provider should supply both BIST and non-BIST (and, where applicable, partial-BIST) versions of the same core.

6.2 ET-Cores

In this section we consider embedded ET-cores that are tested with externally applied vectors. Typically, these cores use scan DFT and the vectors are generated by an ATPG program. Applying an external test to an embedded core is a difficult problem (for example, the core may have more pins than the SoC); the solution usually consists of inserting DFT structures to allow the input stimuli and the core responses to be transported between core and SoC pins [16]. If the ET-core is connected to the system bus, our approach is to ***change the core testing mode from external to BIST and implement the BIST circuit in the embedded FPGA.*** Note that ***the ET-core does not have to be changed***, since the source for its input vectors and the sink for its output vectors are still external to the core under test. But from the SoC's perspective, this is a significant change.

Given a full-scan circuit *C*, many procedures exist to create a TPG that produces a pseudo-random test of reasonable length that always achieves 100% fault-coverage for *C*; [7][28][30] are more recent techniques of this type. The key concept is generating vectors whose signal probabilities are dynamically changed so that the random-pattern-resistant faults [6] of *C* not yet detected will be more likely (or even certain) to be detected by the subsequent vectors. Although some of these techniques require a large area overhead for circuits with multiple scan-chains, the *area overhead is no longer a primary concern when BIST is implemented in reconfigurable logic* (the only constraint is to fit within the embedded FPGA).

Figure 15 illustrates this scheme for an ET-core with a single scan-chain. The circuit under test (CUT) is the combinational part of the full-scan circuit. An input vector is applied at primary inputs (PIs) and via the scan chain, and an output vector is observed at primary outputs (POs) and via the scan chain. The scan chain in the ET-core is extended by a register RI that supplies the PI values, and by a register RO that captures the PO values; RI and RO are implemented in the embedded FPGA, along with a TPG and an ORA. (A separate TPG per chain is needed for multiple scan chains.) The TPG provides the input sequence to the extended scan-chain, and the ORA processes the CUT outputs captured in the extended scan-chain. The ORA is a signature analyzer [6]. Note that the FPGA must have access to the serial data input, serial data output, and the scan-enable pins of the scan chain.

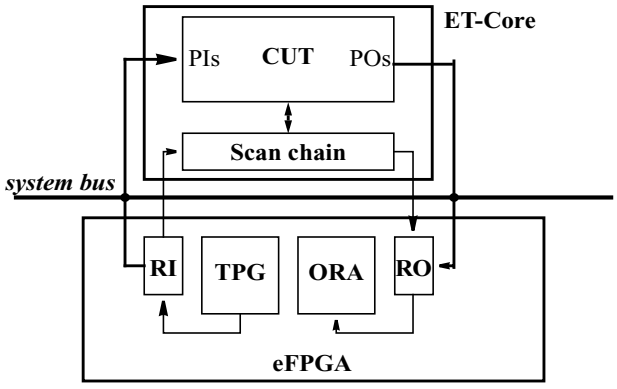

Figure 15. BIST for an ET-core

If the synthesis of the TPG is based on the structure of the CUT, which is not available to the SoC designer, then the tool that generates the TPG will be run by the core provider, and the TPG may be offered as an additional IP to be implemented in the FPGA core. Alternatively, if (non-compacted) ATPG vectors for the CUT are provided with the core, the TPG can be constructed to generate a pseudo-random test set that includes the given vectors. This solution allows the TPG to be created without access to the CUT implementation.

If this approach is feasible for every full-scan embedded core, then *the entire SoC can be tested without external vectors. The FPGA core takes turns to implement an embedded tester for every ET-core* connected to the system bus. If TREC does not need to store and apply vectors, then it can be a low-cost PC-based tester. The total test time will increase because most cores will not be concurrently tested, but this becomes less important when TREC is a low-cost tester.

7. Fault Tolerance Techniques

Our approach for user-level fault-tolerance for the embedded FPGA starts with the circuit implemented in the reconfigurable logic and with the location and type of every diagnosed fault. Note that we may have different circuits residing in the FPGA core at different times, either to perform different functions, or to implement various BIST circuits for other cores; the fault-tolerance analysis is separately done for each such circuit.

Given a circuit *C* implemented in the FPGA and a fault *f*, the first question is whether *f* affects the operation of *C*. If it does, then we have to reconfigure a region of *C* to bypass *f*. We will try to limit the extent of the reconfigured region as much as possible, to avoid changing the critical timing paths of *C*. The computation of most *fault-bypassing configurations* (FABCOs) is done by TREC. Although this computation may involve CAD tools, they are used only incrementally on a small region of *C*, and their run-times are reasonably small. A limit on the allowed computation time is established based on factors such as the cost of a SoC, the production volume, the current yield, etc. When computing a FABCO exceeds the time limit, we will let TREC test other chips, and either discard the defective SoC, or transfer the computation job to a different, and possibly more powerful, processor.

Sometimes the reconfiguration of *C* does affect its critical paths, and then the resulting SoC is functional, but it must operate at a lower clock frequency. If the SoC manufacturer provides several speed versions of the device, selling a repaired SoC as a slower device is much better than discarding it as defective.

In the process of bypassing faults in a defective FPGA core, we have created new FPGA configuration files for the

defective SoC; these files should be used only with the defective chip they were generated for. A good method to maintain the association between a faulty SoC and its configuration files is to create a unique *faulty chip serial number*, and to label both the defective SoC and its configuration files using this identifier. The serial number for all the fault-free chips will be zero. The serial number will be recorded in a small non-volatile memory within the SoC, and it will be readable using a *USERCODE* boundary-scan instruction [22]. The serial number should also appear in the header of every configuration file for this SoC, and the configuration process will be allowed to proceed only if there is a match between the serial number in the SoC and the one in the configuration file.

If fault f does not affect C, then no reconfiguration for tolerating f is required. This is clearly the case when *f* affects FPGA cells or interconnect resources not used by *C*. In addition, we have developed techniques for ***using defective resources whenever possible***. This increases the effective yield without having to change *C*, and in most cases without impact on performance. Since using defective resources avoids using spare resources for fault tolerance, we will be able to tolerate more faults in the same area. Using defective resources is a major departure from most previous work in fault tolerance, whose goal is to bypass every located fault.

7.1 Compatibility

If the function of a circuit *C* implemented in the FPGA is not changed by fault *f*, *C* and *f* are said to be ***compatible***. A compatible fault does not require *C* to be reconfigured. Note that this definition allows *f* to change the timing of *C*. We will define compatibility separately for logic faults and interconnect faults.

7.1.1 Compatibility for Logic Faults

We say that a logic fault *f* in a logic cell is *compatible with the function F* to be implemented in that cell, iff *F* does not change in the presence of *f*. For example, if *F* is a combinational function using only the LUTs, any fault in flip-flops is compatible with *F*. Similarly, a fault affecting only the multiplication operation is compatible with any *F* not using multiplication. In general, any fault that affects only modules or modes of operation of the cell that are not used by *F* is compatible with *F* [3][12]. Then any such *F* can be implemented in the faulty cell, and the configuration file of the fault-free FPGA core is also applicable to this faulty SoC.

Higher diagnostic resolution can allow additional compatibility relations. For example, suppose that testing the RAM module of the cell has identified *f* as a single bit in the RAM being stuck-at-0(1). Consider any function *F* using the RAM module only as a LUT. If the value needed for the faulty bit is 0(1), *f* is compatible with *F*.

7.1.2 Compatibility for Interconnect Faults

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 [21], we say that an interconnect fault *f* is *compatible with a layout L* iff *f* does not change the connections implemented by *L*. If *f* and *L* are compatible, then *L* can tolerate *f* and we don't have to reconfigure to bypass *f*.

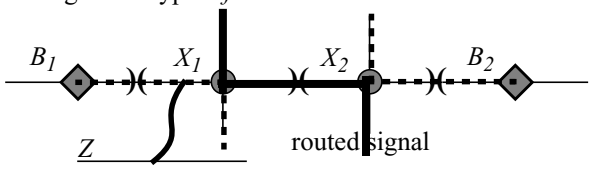

Figure 16. Illustration for fault compatibility

For example, in Figure 16, assume a layout *L* where breakpoint CIPs B_1 and B_2 are open, crosspoint 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 in 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 1 gives detailed conditions that need to be met for single-fault compatibility.

Table 1. Single-fault compatibility conditions

f	**Condition for *f*~*L***
segment open ***or*** CIP stuck-open ***or*** segment stuck-at	*L* does not transmit values through the faulty resource
CIP stuck-closed	*L* uses the CIP ***or*** *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 ***or*** *L* uses the shorted segments for the same signal

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 network on which *L* is implemented. This interaction between faults is not considered in [21]. For example, in Figure 17, assume a layout *L* where crosspoint CIPs X_1 and X_2 and 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 segments *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*.

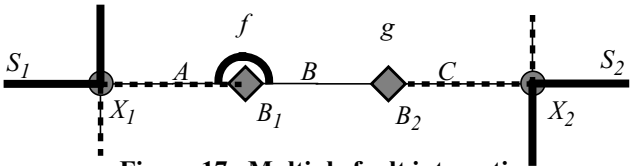

Figure 17. Multiple fault interaction

Our procedures rely on a network model coupled with a layout data base. The *network model* is a graph that represents the topological relations between the components of the programmable interconnect network - wire segments, cell pins, FPGA pins, and CIPs. The *layout data base* defines the CIP settings required to route all the signal nets in the circuit.

From the layout data base 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.

We determine the faulty resources that may be reused (and the ones that need to by bypassed) indirectly, by *finding the signal nets that have to be rerouted*. We say that a *signal net S is compatible with fault f* iff *S* can connect its source to all of its sinks in the presence of *f*, and no other signal drives any segment of *S*. (Note that *f* may be a multiple fault.)

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* from reaching at least one of its destinations, then *S* is incompatible with the faults. 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 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 longest net as the one left in place, since shorter nets are likely to be easier to reroute.

For example, in Figure 17, 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.

In Figure 18, 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. However, S_1 needs to be rerouted.

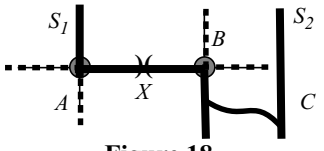

Figure 18.

7.2 Reconfiguration

A tool like JBITS [29] would be ideal to generate FABCOs by directly modifying configuration files without involving any CAD tools. Since no such tools are commercially available, we rely on commercially available place-and-route tools used incrementally. The concept of a fault, however, is alien to these tools, and we need modeling workarounds to make them avoid incompatible faulty resources. To avoid a defective logic cell, we program it to implement a dummy logic function whose inputs and outputs are unconnected. To avoid a defective CIP or wire segment, we program a dummy signal net with no source or sink to occupy the faulty resource and its immediate neighbors.

When we consider reusing a wire segment *S* which is shorted to an unused segment, the delay of *S* increases because of the additional capacitance provided by its partner. However, a conventional path tracing tool that reports the delay along the traced path, will not see the additional delay caused by the short. Our solution is to add the delay of the unused shorted segment to the delay of *S*, and if the new delay causes any path going through *S* to become too slow to operate at the specified clock speed, we give up on reusing *S* and mark the short as incompatible. Although this solution may be conservative in overestimating the delay of *S*, it guaranteed that the short will not affect the critical paths in the circuit.

7.2.1 Reconfiguration for Logic Cells

Many times the function of a logic cell does not utilize all its internal resources. In such cases, when a cell fault *f* is incompatible with the cell function *F*, it is often possible to avoid *f* by small changes in the way *F* is implemented. For example, if one of the LUTs in the cell is faulty, but *F* is not using all the LUTs, we remap *F* to use only fault-free LUTs, which makes *f* compatible and the cell reusable. Of course, we need to incrementally reroute the signals that connected to the inputs and output of the faulty LUT, to bring them to the replacement LUT.

Another example is a 3-input combinational logic function mapped to a 4-input LUT. This is implemented by setting identical values to every pair of RAM bits whose addresses differ only in the value of the “don’t care” address bit, so that the RAM has two identical halves. In this case we can tolerate any combination of faults that affect only one of the two halves, by simply setting the previously unused bit to the value that selects the fault-free half. This also requires an incremental reconfiguration.

If a defective cell has a logic fault incompatible with the function assigned to this cell, we relocate the function to a compatible unused cell, preferably one adjacent to the defective cell (see Figure 19). If we cannot move the function to a compatible cell adjacent to the defective one, we set up a

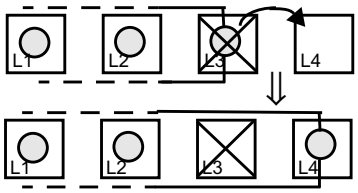

Figure 19. Cell replacement

chain of cell replacements [11], as illustrated in Figure 20. This approach averages the overall change in signal disturbance, but increases the number of signals that must be rerouted. Note that in our approach, no replacement cell (including the unused one) is required to be fault-free, but only compatible with the desired function. After cell functions are relocated, we use an incremental router to reroute the signal nets connected to the moved cells. Also we need to analyze the changes in the timing of the rerouted signals, to determine whether the chip should operate at the original clock frequency or a reduced one.

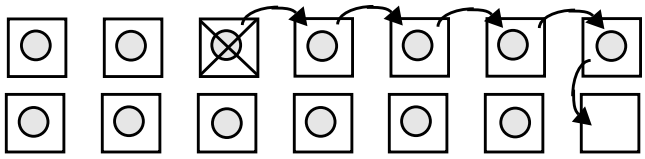

Figure 20. Chain of cell replacements

As a preprocessing step, we prepare a FABCO for each one of the used cells in the target circuit. This not a computationally expensive job, since all the circuit transformations are only incremental. These precomputed configurations are stored on the TREC disk, and are applied by TREC based on the diagnosis results. Note that the same unused cell can be designated as replacement for several working cells. After applying one FABCO, several other precomputed configurations may no longer be valid, since the circuit for which they were derived has been changed. It is another TREC task to keep track of the validity of precomputed FABCOs. If the FABCO for a defective cell has been invalidated, TREC computes a new one, by incrementally rerouting the signal nets connected to the moved function.

Another major advantage of our technique over conventional techniques is that we handle groups of faults in a tight area. Other techniques limit the number of faults they can handle by providing a limited number of spare logic resources in a region of the FPGA. When these local spares are exhausted, their fault tolerant techniques fail. We provide a global reconfiguration approach that uses *minimax matching* [19] to match defective cells to compatible cells, so that the maximum distance between any defective cell and its replacement cell is minimized.

7.2.2 Reconfiguration for Interconnect

Our experimental results show that in most practical circuits implemented in FPGAs, a large number of interconnect faults are compatible with the circuit [13]; this happens because many interconnect resources are unused, even in circuits where more than 70% of the cells are used. Every such fault can be tolerated without reconfiguration.

Next we describe our multi-step approach for avoiding incompatible interconnect faults. In the following procedure, *S* is the set of nets that have to be rerouted.

Stage 1: First we process incompatible faults that do not allow some signal nets to connect to inputs or outputs on a cell, because such faults may need reconfiguration for both logic and interconnect. If an input or an output of a cell is blocked, and the cell is not fully used, then we try to move the function of the blocked resource to an unused resource in the same cell. Figure 21 shows an example of a stuck-open CIP denying access to an input pin on LUT B. If LUT A is not used, we first try to move the function from LUT B to LUT A; keeping the LUT function within the same cell is less likely to introduce delays. For this, we reconfigure the cell logic and we rip-up the signal nets going to the inputs and outputs of LUT A [17]. If such a transformation is not possible within the blocked cell, we rip-up all the signal nets attached to the cell, and move the cell function a new compatible unused location. In either case we add the ripped-up signals to *S*

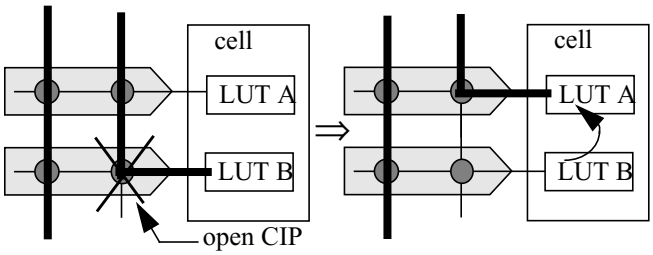

Figure 21. Reconfiguring a cell to bypass a CIP fault

Stage 2: We determine all the signal nets incompatible with the remaining faults using the procedure described in Section 7.1. For every such net we perform a partial rip-up, trying to maintain as much as possible from its original routing, and add the signal to *S*. For example, Figure 22a shows a signal net that was originally routed through segment X_1X_2 that is now open. Here we rip-up only the faulty segment.

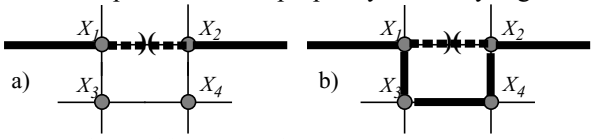

Figure 22. Bypassing an open segment

Stage 3: In this stage we attempt to incrementally reroute each signal net in *S*, without changing the routing of other non-affected signal nets. For example, in Figure 22a, if X_3 and X_4 are not used, the signal may be easily rerouted as shown in Figure 22b. Every successfully rerouted net is removed from *S*. If *S* becomes empty, we stop.

Stage 4: Here we use a more global transformation, completely ripping-up all the signal nets within a variable-sized window in the area around the signal nets in *S*, adding all the ripped-up signals to *S*, and rerouting all signals in *S*. If we are not successful, we increment the window size and repeat the process. If *S* becomes empty, we stop.

Stage 5: Here we do a new place-and-route for the entire circuit (with the faults marked by dummy signals).

Because many of interconnect faults are compatible with the layout, the last two stages are unlikely to be used in practice. Since each stage is computationally more expensive than its preceding one, this procedure will be subject to run-time bounds as described at the beginning of Section 7.

8. Conclusions

In this paper, we have shown that an embedded FPGA core is an ideal host to implement infrastructure IP for yield improvement in a bus-based SoC. The reconfigurable logic allows the infrastructure IP to be created only when needed, and enables BIST without any area overhead or performance degradation. Our BIST techniques achieve practically complete fault coverage for both logic cells and interconnect, and our adaptive diagnosis techniques can locate almost any combination of faulty cells or interconnect faults, and can also identify faults within defective cells. Such accurate diagnosis is a key factor in achieving effective repair for the embedded FPGA. Instead of the conventional approach of reconfiguring to bypass every located fault, our fault tolerance approach reuses defective resources whenever possible. This increases the effective yield without reconfiguration, and allows tolerating more faults in the same area. Reuse is based on new techniques to determine compatibility between faults in a cell and its function, or between a group of interconnect faults and the layout. We may reuse fault-free parts of defective cells and even shorted segments. For incompatible faults we determine fault-bypassing configurations by incrementally changing the implementation of the circuit, usually in a small area around the faults.

An embedded FPGA core makes significant changes to the SoCs testing paradigm. After the FPGA core is fully tested and repaired if needed, we can use it to provide embedded testers in turn for other cores in the SoC. This allows non-intrusive BIST logic to be removed from cores, reducing area overhead and delay penalties. It also enables cores designed to be tested with external vectors to be tested with BIST, and the entire SoC to be tested with a low-cost tester.

9. References

[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] 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

[3] 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

[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. Third NASA/DoD Workshop on Evolvable Hardware*, pp. 73-92, 2001

[5] M. Abramovici, C. Stroud, and J. Nall, "BIST-Based Diagnosis of FPGA Interconnect," submitted to *Intn'l. Test Conf*., 2002

[6] P. H. Bardell, W. H. McAnney, and J.Savir, *Built-In Testing for VLSI: Pseudorandom Techniques*, John Wiley and Sons, 1987

[7] N. Basturkmen, S. M. Reddy, and I. Pomeranz, "Pseudo Random Patterns Using Markov Sources for Scan BIST," submitted to *Intn'l. Test Conf*., 2002

[8] R. Cliff *et al*., "Programmable logic devices with spare circuits for replacement of defects," *U.S. Patent 5,485,102*, Jan. 1996

[9] K. Chakraborty and P. Mazumder, *Fault-Tolerance and Reliability Techniques for High-Density Random-Access Memories*, Prentice Hall, 2002

[10] B. Culbertson *et al*., "Defect Tolerance on the Teramac Custom Computer," *Proc. 5th IEEE Symp. on Field-Programmable Custom Computing Machines*, pp. 140-147, 1997

[11] J. Emmert and D. Bhatia, "Partial Reconfiguration of FPGA Mapped Designs with Applications to Fault Tolerance and Yield Enhancement," *Proc. Intn'l Conf. on Field-Programmable Logic*, pp. 141-150, 1997

[12] 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*, pp. 165-174, 2000

[13] J. Emmert, S. Baumgart, P. Kataria, A. Taylor, C. Stroud, M. Abramovici, "On-line Fault Tolerance for FPGA Interconnect with Roving STARs," *IEEE Intn'l. Symp. on Defect and Fault Tolerance in VLSI Systems*, pp. 445-454, 2001

[14] I. Harris and R. Tessier, "Interconnect Testing in Cluster-Based FPGA Architectures", *Proc. AMC/IEEE Design Automation Conf.*, pp. 49-54, 2000

[15] 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.

[16] IEEE P1500 Standard for Embedded Core Test, *http://grouper.ieee.org/groups/1500/*

[17] V. Lakamraju and R. Tessier, "Tolerating Operational Faults in Cluster Based FPGAs", *Proc. ACM Intn'l. Symp. on FPGAs*, pp. 197-194, Febr. 2000

[18] Lattice Semiconductor Co., *http://www.latticesemi.com/products/fpga*

[19] T. Leighton and P. Shor, "Tight Bounds for Minimax Grid Matching with Applications to Average Case Analysis of Algorithms," *Proc. Symp. on Theory of Computing*, pp. 91-103, 1986

[20] E. McCluskey, "Verification Testing - A Pseudoexhaustive Test Technique," *IEEE Trans. on Computers*, Vol. C-33, No. 6, pp. 541-546, June, 1984.

[21] K. Roy and S. Nag, "On Routability for FPGAs Under Faulty Conditions," *IEEE Trans. on Computers*, Vol. 44, pp. 1296-1305, 1995

[22] A. Russ and C. Stroud, "Non-Intrusive Built-In Self-Test for FPGA and MCM Applications," *Proc. IEEE Automatic Test Conf*., pp. 480-485, 1995

[23] "Standard Test Access Port and Boundary-Scan Architecture," *IEEE Standard P1149.1*, 1990

[24] 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

[25] 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*, pp. 31-39, 2001

[26] C. Stroud, S. Wijesuriya, C. Hamilton, and M. Abramovici, "Built-In Self-Test of FPGA Interconnect," *Proc. Intn'l. Test Conf*., pp. 404-411, 1998

[27] C. Stroud, *A Designer's Guide to Built-In Self-Test*, Kluwer Academic Publishers, 2002.

[28] N.A. Touba and E.J. McCluskey, "Bit-Fixing in Pseudo-Random Sequences for Scan BIST", *IEEE Trans. on CAD*, Vol. 20, No. 4, pp. 545-555, Apr. 2001.

[29] P. Sundararajanand and S. A. Guccione, "Run-Time Defect Tolerance using JBits," *Proc. ACM Intn'l. Symp. on FPGAs*, pp. 193-200, Febr. 2001.

[30] S. Wang, "Low Hardware Overhead Scan Based 3-Weight Weighted Random BIST," *Proc. Intn'l. Test Conf*., pp. 868-877, 2001

[31] Xilinx, Inc., *http://www.xilinx.com/products*