Fault Location in FPGA-Based Reconfigurable Systems

Subhasish Mitra, Philip P. Shirvani and Edward J. McCluskey

Center for Reliable Computing
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

Abstract

In this paper, we describe a new technique for locating faulty Lookup Tables (LUTs) in FPGA-based reconfigurable systems. The technique is in-place (does not alter the routing structure of the LUT network) and is based on pseudo-exhaustive Built-In Self-Test where each configured LUT is tested exhaustively. Our technique involves selective reprogramming of the LUTs and takes advantage of partial reconfiguration when it is available.

1. Introduction

Programmable logic devices (PLDs) have long been used as rapid and economical means to prototype digital logic designs. Field programmable devices (FPDs) are becoming more prevalent in end products because they can drastically reduce system cost and time-to-market by allowing for economical, last-minute system enhancements, performance improvements, and bug fixes. With the production of FPDs that can be reprogrammed in-system, multiple times, it is now feasible to implement *adaptive computing systems (ACS)* [Lach 98][Rupp 98][Saxena 98]. These systems can adapt to rapidly changing environmental and computational requirements by reconfiguring themselves on the fly, allowing for significant cost and performance advantages over a fixed implementation of the same system.

Due to the increased *dependability* (availability, reliability, maintainability, testability and fault tolerance) requirements in harsh and unusual operating environments, it was impossible to execute safety critical applications on low-cost commercial technology. However, executing these applications on adaptive hardware means that the system can now be both low-cost and highly dependable. Moreover, an adaptive system allows each application to adapt its level of fault tolerance to changing dependability requirements. Thus, these systems should be able to:

1. ***Detect*** run-time errors by using innovative on-line checking techniques, and ***detect*** manufacturing and configuration defects by using off-line Built-In Self-Test (BIST) techniques.

2. ***Precisely locate*** defects by reconfiguring themselves to include additional checkers and BIST logic.

3. ***Rapidly reconfigure*** themselves to avoid the located defects, and depending on the extent of the defects, operate with less on-line checking or in a degraded functional mode.

In this paper, we address the problem of locating the faulty unit once the on-line checkers detect errors. The platform under consideration is a field programmable gate array (FPGA) based system — specifically, we consider the SRAM-based FPGAs (e.g., the Xilinx FPGAs [Xilinx 96]). In the following paragraphs, we briefly describe the SRAM-based FPGA architecture and introduce the terminology — this will help in describing the problem under consideration.

An FPGA has a number of Configurable Logic Blocks (CLBs) interconnected by wires and switch boxes (shown by small black boxes in the diagram) with each other and to the input/output blocks. The architecture is shown in Fig. 1.1. The CLB contains logic function generators, flip-flops (latches), and a number of multiplexers and associated circuitry. Wires within the FPGA are connected through pass transistors. The pass transistor is turned on or off by loading a binary value into the configuration SRAM cell connected to its gate.

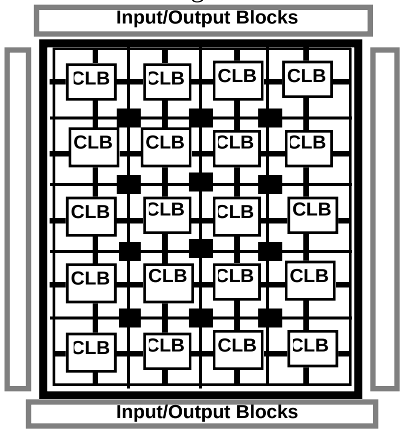

Figure 1.1. Architecture of a Field Programmable Gate Array (FPGA).

The CLBs contain function generators which are also made up of SRAMs. A function generator has a Lookup Table (LUT) made out of SRAM cells, together with some addressing circuitry. The LUT is mostly used for implementing combinational logic, although in the Xilinx XC4000 series it can also be used as a RAM. The function is implemented in the LUT by storing the corresponding truth table inside the LUT.

In this paper we address the fault location problem. For an FPGA-based reconfigurable system, faults can occur both in the logic blocks (LUTs, CLBs, etc.) and in the interconnects that provide connection between the different logic blocks. A typical fault in a logic block may be a stuck-at fault in any LUT location. A typical interconnect fault may be a stuck-at fault in the configuration SRAM cell whose content turns a pass transistor on or off, providing interconnection between two leads. Both of these types of faults are equally important. In this work, we will address the fault location problem for the FPGA logic blocks.

For the entire reconfigurable system to have high dependability, there are some stringent requirements for any fault location procedure. First of all, the diagnostic resolution should be fine-grained. For example, consider fault location technique A that identifies a faulty CLB in a design and technique B which locates a faulty LUT in the same design. So far as diagnostic resolution is concerned, technique B is generally more preferred than technique A because, typically a CLB contains three LUTs and when a faulty CLB is identified, all the three LUTs inside it will be discarded; however, if a faulty LUT is located, we can still use the other two LUTs in the CLB. Moreover, with a finer resolution, it may be easier for the synthesis tool to re-map the original design to the set of LUTs excluding the faulty one identified.

So far as highly dependable reconfigurable systems are concerned, it is not very desirable to shut the entire system down when fault location is being carried out in one part of the system. Thus, it is desirable that some part (possibly the critical part) of the system still functions even though fault location is being carried out in other parts of the system. We call these type of fault location techniques *partially on-line*. With the availability of partially reconfigurable FPGAs (e.g. the Xilinx XC6200 [Xilinx 96] and Atmel AT6000 [Atmel 97] families), it is becoming quite feasible to devise fault location techniques that are partially on-line. As an example, consider a case where a circuit has been mapped in such a way that it spans multiple FPGA modules. In that case, we can selectively reconfigure some CLBs (or LUTs) of the FPGA modules without bringing the entire module off-line. Finally, a fault location technique should be expected to be fast enough to cope with the stringent requirements of the system's responsiveness to a failure.

In this paper, we will focus on combinational logic circuits built out of the LUTs. In the future, we will extend this work for sequential circuits. Given a combinational logic network mapped onto a collection of LUTs with defined interconnection among the LUTs, our goal is to locate a faulty LUT when errors have been detected at the output of the given combinational logic circuit. The fault model under consideration is not restricted to a single stuck fault — later in this paper, we will discuss more about the fault model. Our technique utilizes the reconfigurability of the LUTs to perform the diagnosis. In addition, we rely on pseudo-exhaustive BIST [McCluskey 81] for pattern generation — hence, minimal external interaction is required. A novel aspect of our technique is the way the concept of pseudo-exhaustive BIST has been applied for fault-location purposes. Another novelty of our approach lies in the fact, that we do not alter the routing structure among the different LUTs (CLBs) of our design, i.e., it is an in-place fault location technique. This is an important feature, because typically designs mapped to FPGAs are routing-limited. Also, modifying the existing routing configuration during each fault location step is time consuming.

Another advantage of our technique is that, with partially reconfigurable FPGAs, while the circuit under consideration is being diagnosed, the remaining part of the system whose inputs are not connected to the outputs of this circuit, can still be operational (i.e., it is partially on-line) . Our technique can be extended to locate faulty CLBs rather than LUTs — i.e., the diagnostic resolution of our technique is flexible.

In Sec. 2 of this paper, we review recently published techniques related to diagnosis in FPGAs. In Sec. 3, we review the basic concepts related to pseudo-exhaustive BIST. Section 4 describes our basic scheme for fanout-free LUT networks and provides bounds on the reconfiguration complexity associated with our technique. In Sec. 4.A, we analyze the reconfiguration complexity of our algorithm. In Sec. 5, we extend the algorithm of Sec. 4 to handle general LUT networks. Section 6 presents experimental results. Finally, we provide a summary of this work in Sec. 7.

2. Previous work

Testing the logic blocks of reconfigurable FPGAs has been studied by many researchers [Jordan 93][Liu 95][Stroud 96]. Testing the interconnects was discussed in [Renovell 97][Renovell 98]. Techniques to locate faulty FPGA interconnects are described in [Huang 96][Lombardi 96]. Since our focus is on FPGA logic blocks, we will describe the techniques for logic block diagnosis only.

The testing technique described in [Stroud 96] requires a fixed number of reconfiguration sessions. It reconfigures some of the logic blocks as pattern generators or response analyzers, while testing the other blocks and vice-versa. The technique does not use any knowledge of the application that was implemented in the FPGA. Hence, it requires a set of configurations that cover all the faults under consideration for all possible configurations. This technique is extended in [Stroud 97] for diagnosis to locate the faulty logic blocks in an FPGA.

The method discussed in [Wang 97] can be used to locate multiple faults in an FPGA. The basic idea is similar to that of [Stroud 97]. Typically, a part of the FPGA is reconfigured to test another part and vice-versa. For example, the FPGA is divided into three sets of CLBs and each set tests another set according to a diagnostic graph. The test time in this method depends on the number of faults and is independent of the array size. The techniques in [Stroud 97] and [Wang 97] are based on BIST.

Another application independent diagnostic technique is presented in [Inoue 98]. The technique consists of two test steps: horizontal diagnostic and vertical diagnostic. These two steps identify the row and column respectively that contain a faulty CLB. The C-testability concept [Friedman 73] is used to improve this technique such that the test time is independent of the array size.

All the mentioned techniques are application independent and can be used for both production test and field test. However, when diagnostic is needed for an FPGA that implements a fixed application, the diagnostic procedure can be accelerated by using the design information rather than testing for all possible configurations (as will be seen in Section 4).

3. Pseudo-exhaustive BIST technique

A BIST technique based on Pseudo-Exhaustive (PE) testing was presented in [McCluskey 81]. An n-input combinational network can be tested thoroughly by applying all 2^n input combinations and verifying that the correct output is obtained for each combination. This technique is sometimes called *Exhaustive Testing*. However, most combinational networks have more than one output, and for many cases, each of these outputs depends on only a subset of the inputs. Thus, it may be possible to exhaustively test each output by applying all combinations of only those inputs on which the output depends — this is called *pseudo-exhaustive* testing. A simple example of a pseudo-exhaustive testing technique has been illustrated in Fig. 3.1.

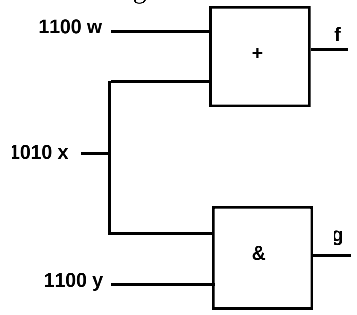

Figure 3.1 Pseudo-exhaustive BIST example.

The f output depends only on inputs w and x while the g output depends on x and y. The four input patterns shown in the figure are such that all possible combinations of values are applied to w and x, and also all combinations of x and y are present. Thus both f and g are tested exhaustively using only four input patterns. In fact, only four test patterns are used for pseudo-exhaustive testing of the example circuit instead of eight patterns required for full exhaustive testing of the circuit. Techniques for generating the minimal length pseudo-exhaustive test patterns have been described in [McCluskey 82]. The advantages of pseudo-exhaustive testing are:

(i) It is a BIST technique and does not require an expensive tester for storing the patterns to be applied. The memory requirement for storing the output responses can be eliminated by using a compaction technique such as signature analysis [McCluskey 85].

(ii) This technique does not rely on an explicit fault model and is thus not limited to any specific class of faults such as single stuck at faults.

(iii) This technique provides high fault coverage.

If any output of a given combinational logic circuit depends on all inputs, then the pseudo-exhaustive testing technique described above is not applicable. For this case, we have to use some *segmentation* (*partitioning*) techniques, described in [McCluskey 82]. If partitioning can be carried out such that the number of input lines to each subcircuit is significantly fewer than in the original circuit, it will be possible to test each subcircuit exhaustively. There are two ways of achieving partitioning: (i) *multiplexer partitioning* and (ii) *sensitized partitioning*.

In multiplexer partitioning, access to the embedded inputs and outputs of a subcircuit under test can be achieved by inserting multiplexers and connecting the embedded inputs and outputs of each subcircuit to those primary inputs and outputs that are not used by the subcircuit under test [McCluskey 81].

The multiplexers used for multiplexer partitioning may reduce the speed of operation and are costly to implement. However, it is possible to achieve partitioning without actually inserting any multiplexer at all. Circuit partitioning and subcircuit isolation can be achieved by applying the appropriate input pattern to some of the input lines. The effect achieved is similar to that of hardware partitioning: paths from the primary inputs to the subcircuit inputs and paths from the subcircuit output to the primary output can be sensitized. Using these paths, each subcircuit can be tested exhaustively. Extensive research has been conducted to propose various segmentation techniques for pseudo-exhaustive testing [Udell 87].

Research related to pseudo-exhaustive testing has typically focused on fault detection. However, in an FPGA based reconfigurable system, pseudo-exhaustive BIST (PE-BIST) can be advantageous for locating faulty modules (FPGA modules or CLBs or LUTs). In fact, the reconfigurability of the FPGAs provides an added advantage — we do not need to have any multiplexer or sensitized partitioning. We can just reprogram (reconfigure) a few logic blocks to achieve the effects of partitioning. We describe our technique in the next section.

4. Fault location using PE-BIST: fanout-free LUT networks

In this section, we will describe our basic fault location technique using pseudo-exhaustive BIST. The diagnostic resolution will be the LUTs, i.e., we try to locate the faulty LUTs. Also, we will assume that only a single LUT among a set of LUTs to which a given combinational circuit is mapped, can fail. However, we do not have any assumption about the number of faults inside a faulty LUT. For the ease of explanation, we will first focus on single-output LUT networks with no fanout. Later on, we will show how to handle general LUT networks with fanouts.

For example, let us consider the combinational circuit in Fig. 4.1. The circuit implements an output function $z = (ac + bc + def)g$. Let us suppose, that the circuit has been mapped to three lookup tables (LUTs) as shown in

Fig. 4.1. LUT1 implements the function $w = ac+bc$, LUT2 implements $x = def$, and LUT3 implements $z = wg+xg$. As mentioned earlier, the LUTs are SRAMs which store the truth tables of the functions they implement. For example, for the given implementation, LUT1 stores the truth table corresponding to $ac + bc$. Note that, typically LUTs have four to six inputs. For the purpose of illustration, we have used LUTs with three inputs in Fig. 4.1. We assume that the system is equipped with some on-line checkers, e.g., parity checkers, which perform concurrent error detection. Suppose, from the response of the on-line checkers, we found that the response z of the given network is erroneous. Now, it is our aim to locate the faulty LUT in the given network.

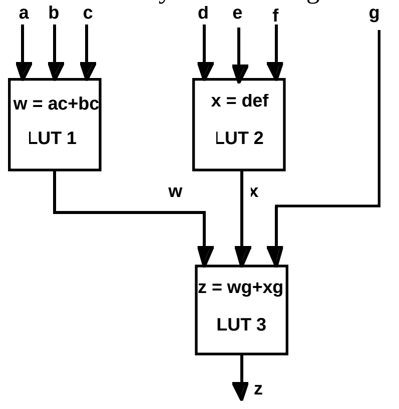

Figure 4.1 An example combinational circuit mapped to LUTs.

To achieve our goal, we use a pseudo-exhaustive BIST technique. We do not alter the interconnection structure among the different LUTs. However, we assume that a few CLBs can be configured as Linear Feedback Shift Registers (LFSRs) to generate all possible patterns of length 3. We also assume that the output z is either directly observed or compacted using some signature analysis technique with negligible aliasing.

The given interconnection of LUTs forms a natural partition (segmentation) of the circuit under consideration. Each LUT forms a segment. Note that, due to the reconfiguration capability of the LUTs, we do not require any multiplexer or sensitized partitioning. An LUT can be configured into *pass* modes such that it passes a single input signal to the output. For example, for a 3 input LUT with inputs a, b and c and output w, the LUT will be configured as shown in Table 4.1 to pass the input a to output w.

Table 4.1 Configuration to pass input *a* to *w*.

abc	000	001	010	011	100	101	110	111
w	0	0	0	0	1	1	1	1

In Session 1, LUT1 of Fig. 4.1 is tested exhaustively, by applying all possible patterns of length 3 at the inputs a, b and c and configuring LUT3 in the pass mode to pass input w to the output z — we can then observe the response of output z. Similarly, in Session 2, in order to test LUT2 exhaustively, we apply all possible patterns of length 3 to the inputs d, e, f and configure LUT3 in the pass mode to pass input x to output z. Finally, in Session 3, for testing LUT3 exhaustively, we apply all possible patterns to input a, d and g and configure LUT1 and LUT2 in pass mode to pass signals on inputs a and d to outputs w and x, respectively. A De-Bruijn counter [McCluskey 86] can be used to generate all possible patterns of a given length (3 in this case).

Now, let us consider the responses obtained by applying pseudo-exhaustive tests to the three LUTs under 3 possible cases: (i) LUT1 is faulty, (ii) LUT2 is faulty and (iii) LUT3 is faulty. This is shown in Table 4.2. We call this table a *fault table*. An entry C (Correct) means that the response of z is the same as the fault free response. An entry E (Error) means that the response of z is faulty. An entry U (Unknown) means that the response of z may be faulty or fault-free depending on the fault.

Table 4.2 Fault table.

Faulty	Test Sessions		
Unit	LUT1	LUT2	LUT3
LUT1	E	C	U
LUT2	C	E	U
LUT3	U	U	E

When LUT1 is faulty and we apply exhaustive patterns to LUT1, obviously, the response at z will be an E. When LUT1 is faulty and we test LUT2 exhaustively, the response must be the same as fault-free response because there is no chance that the inputs or output of LUT2 get *corrupt* due to the interference of a faulty LUT. However, when LUT1 is faulty, the response obtained during exhaustive testing of LUT3 may or may not be faulty. For example, if the location in LUT1, corresponding to $a = 0$, $b = 0$ and $c = 0$ is only faulty (say stuck at 1), and while testing LUT3, we apply $b = 1$ and $c = 1$ (and LUT1 configured in pass mode), then the response obtained at z will be fault-free. However, if the location of LUT1 corresponding to $a = 1$, $b = 1$ and $c = 1$ becomes faulty then the response of z, while testing LUT3, can become faulty. The case of LUT2 being faulty is similar.

Now consider the case where LUT3 is faulty. It is obvious that the we obtain an erroneous response on z in Session 3. However, the response on z may be faulty or fault-free, when we test LUT1 and LUT2 in Session 1 and Session 2. Hence, the third row of Table 4.2 has entries U in the first two columns. A response is *consistent* with a particular row i of the fault-table if and only if for each test session, if row i contains an E (or C) then the response also contains an E (or C) for the same test session. Now, suppose that the response from the three sessions is $<E, C, E>$. This is consistent with the first and the third rows of Table 4.2. Hence, we have to decide whether LUT1 or LUT3 is faulty.

If we restrict ourselves to cell stuck faults (not necessarily single), i.e., the cells in the LUTs can be stuck at 0 or 1, then there is a straightforward way of finding out whether LUT1 or LUT3 is faulty. We first reconfigure LUT3 in all 0's mode; i.e., 0s are written to all the locations of LUT3. LUT1 is configured in the pass mode to pass the signal on input a to w. Similarly, LUT2 passes the signal on input *d* to *x*. We exhaustively test LUT3 in this configuration. Next, we reconfigure LUT3 in all 1's mode and perform exhaustive testing of LUT3. If no faulty response is generated in any one of these test sessions, then it means that, with respect to cell stuck faults, LUT3 is not faulty — otherwise it is faulty. This is because, if LUT3 is fault-free, then the response out of LUT3 is always correct, because all its locations contain the same value (0 or 1). If LUT1 or LUT2 is faulty, the result will be that some patterns applied to the inputs of LUT3 will be corrupt. Nevertheless, the response on output *z* will still be correct.

We can perform the same procedure for LUT1. Table 4.3 shows the corresponding fault table. As we can see, the two rows in Table 4.3 are distinct. Hence, we can locate the faulty LUT with respect to cell stuck faults. This second step introduces some adaptivity in our algorithm.

Table 4.3 Fault table.

Faulty	Test Sessions		
Unit	LUT1	LUT2	LUT3
LUT1	E	C	C
LUT3	U	U	E

Our basic algorithm (the first step) is shown in Algorithm 1. The definition of post-order traversal, mentioned in Step 4 of Algorithm 1 can be obtained from [Cormen 89]. Algorithm 2 presents the technique to locate the faulty LUT with respect to cell-stuck faults on the path that contains the faulty LUT. In the next sub-section (Sec. 4.A), we derive some bounds on the reconfiguration complexity associated with our technique and prove the correctness.

4.A Reconfiguration complexity and correctness

In this section, we will first derive bounds on the reconfiguration complexity of the diagnosis procedure for a fanout-free network of LUTs, described in Sec. 4 (Algorithm 1). For the following discussion, we introduce the following notation. Suppose that there are *n* LUTs in the given network. *p* of these LUTs are driven only by primary inputs and are called *input* LUTs. The remaining LUTs are called *internal* LUTs. Let there be *m* internal LUTs. Then, $n = m + p$. Now, the number of inputs of LUT_i is written as f_i.

Theorem 1: For a fanout-free single output network of LUTs, an upper bound on the total number of reconfigurations required in Algorithm 1 is given by:
$U_1 = (\Sigma_{i \in \text{ internal LUTs}} f_i) + 2m + p$.

Proof: Let us consider the following situation. In the worst case, each internal LUT has to be configured to pass each of its inputs to the output — this contributes to the term $(\Sigma_{i \in \text{ internal LUTs}} f_i)$ in the above expression. Now, each internal LUT has to be configured in the normal functional mode to get tested pseudo-exhaustively. Moreover, after getting tested pseudo-exhaustively, the same LUT has to be reconfigured into a pass mode to test the LUTs which are on a path from that LUT to the output. This contributes to the term $2m$ in the above expression. Finally, the term *p* comes from the fact that after the input LUTs are tested pseudo-exhaustively, they should be reconfigured in a pass mode to pass test patterns to the inputs of internal LUTs. The bound can also be written as $U_2 = n\text{-}1 + 2m + p$. The formula for U_2 can be derived from the formula for U_1 by observing the fact that if an input of an internal LUT is connected to a primary input, then we need not consider that particular input in the first expression in the formula for U_1. Thus, for a LUT network having no internal LUT whose input is connected to a primary input, $U_1 = U_2$. Otherwise, typically $U_2 < U_1$. Q.E.D.

ALGORITHM 1
Input: A fanout-free network of LUTs for a single-output combinational logic circuit
Output: A reconfiguration schedule for testing each LUT exhaustively

Procedure:
```
1. Configure input LUTs (LUTs whose inputs are connected to primary inputs only) in normal functional mode
2. Configure each internal LUT to pass its leftmost input (connected to a LUT output) to the output (pass mode)
3. while (all LUTs not visited)
4.      Consider the next LUT j in the post-order traversal of the network
5.      If LUT j is not an input LUT, reconfigure it into functional mode
6.      If output of LUT j is connected to input k of LUT m, reconfigure LUT m to pass input k to its output
7.      Test LUT j exhaustively
8.      Reconfigure LUT j to pass one of its inputs to the outputs
9. endwhile
end
```

ALGORITHM 2
Input: A path *p* in the LUT network containing the faulty LUT
Output: The faulty LUT

Procedure:
```
1. Consider the LUT i on p whose output is the primary output
2. Configure LUT i to contain 0 in all locations
3. Test LUT i exhaustively
4. Configure LUT i to contain 1 in all locations
5. Test LUT i exhaustively
6. If any of the responses is faulty
        LUT i is faulty, break
7. Consider the LUT j on p which is a direct predecessor of LUT i
8. Configure LUT i to pass its input that is connected to the output of LUT j, to its output
9. Repeat lines 2-8 for LUT j (i taking the value of j) until we have considered all LUTs on p.
end
```

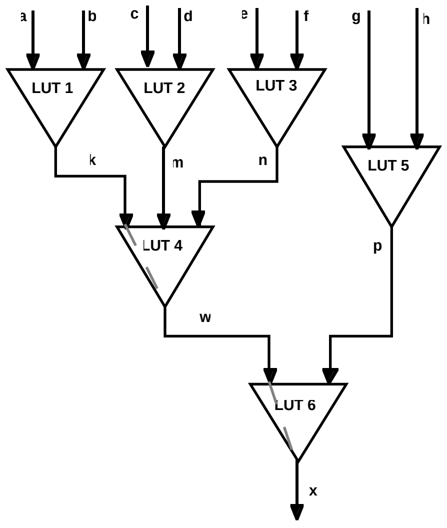

Figure 4.2 An example to illustrate proof of Theorem 1.

Typically, f_i is constant (say, f). Then, the reconfiguration complexity is O(*n*), where *n* is the number of LUTs in the network. Moreover, the actual number of reconfigurations required can be reduced, if we take the advantage of sensitized partitioning — however, that increases the overhead of the software to control the diagnosis. We explain the proof of Theorem 1 with the help of the following example (Fig. 4.2). There are six LUTs which are interconnected. There are 4 input LUTs and 2 internal LUTs — i.e., $n = 6$, $p = 4$ and $m = 2$. Initially, LUT1, LUT2, LUT3 and LUT5 are configured in their functional mode, while LUT4 is configured to pass *k* to *w* and LUT6 is configured to pass *w* to *x*. In this configuration, we can test LUT1 exhaustively. Next, LUT4 has to be configured to pass *m* to *w* to test LUT2 and *n* to *w* to test LUT3. Thus, LUT4 requires 3 reconfigurations. Next, LUT1, LUT2 and LUT3 must be configured in the pass mode to test LUT4 exhaustively (in that case LUT4 has to be reconfigured into functional mode). Finally, for testing LUT6, we need to reconfigure LUT4 and LUT 5 in pass mode and LUT6 in normal functional mode. Thus, the total number of reconfigurations required is $(3 + 2) + 2 \times 2 + 4 = 13$. The corresponding test schedule is LUT1, LUT2, LUT3, LUT4, LUT5 and LUT6. Note that, in this case, $n\text{-}1 + 2m + p$ also evaluates to $6\text{-}1 + 2 \times 2 + 4 = 13$.

Theorem 2: After the execution of the first step (Algorithm 1), we can either locate the fault or isolate a path where the fault lies.

Proof: Suppose that at the end of the first iteration we have two candidate LUTs, LUT1 and LUT2 which may be faulty and also, LUT1 and LUT2 do not lie on the same path from the primary input to the output. This implies, there was no *interference* by LUT2 when LUT1 was tested and vice-versa. This, in turn, implies that both LUT1 and LUT2 must be faulty (because no other LUTs are faulty). However, our basic assumption was that there can exist a single faulty LUT. Q.E.D.

Since after the execution of the first step, we get information about the path containing the faulty LUT, we can directly apply Algorithm 2.

Table 4.4 Fault table.

Faulty Unit	Test Sessions					
	LUT1	LUT2	LUT3	…	LUT*n-1*	LUT*n*
LUT1	E	C	C	…	C	C
LUT2	U	E	C	…	C	C
LUT3	U	U	E	…	C	C
…	…	…	…	…	…	…
LUT*n-1*	U	U	U	…	E	C
LUT*n*	U	U	U	…	U	E

Theorem 3: Algorithm 2 can locate the faulty LUT on a path with respect to cell-stuck faults.

Proof: Consider a path *p* of LUTs LUT1, LUT2, ..., LUT*n*, where the output of LUT*n* is the primary output. If LUT*n* is faulty with respect to cell stuck faults, we can detect that when we test LUT*n* in all 0's and all 1's mode. However, if LUT*n* is not faulty (i.e., some other LUT on *p* is faulty), the response out of LUT*n* is always the correct response (since all locations of LUT*n* contain the same value). For the case where LUT*n-1* is not faulty and LUT*n* is faulty, the response of LUT*n-1* may get corrupt. Hence, in a fault table, the corresponding entry will be a U. When LUT*n-1* is faulty, we will obtain a faulty response and the entry in the fault table will be E. However, if neither LUT*n* nor LUT*n-1* is faulty, then we will always obtain a correct response when we test LUT*n-1*. Thus, the fault table is shown in Table 4.4. We find that each row the fault table is distinct, and hence, we can locate the faulty LUT. Q.E.D.

5. Fault location: general LUT networks

In this section, we extend our basic faulty LUT location technique to general LUT networks. However, we restrict ourselves to only single-output networks. For multi-output LUT networks, we can extract the cones from each individual output to the primary inputs and apply the fault-location procedure for each cone. For a general LUT network, there can exist multiple paths from an LUT output to the primary output and also from the primary inputs to the LUT inputs. For the current work, we have chosen a simple *heuristic* for choosing one single path when there are a number of possible choices. The heuristic is to use the shortest path from the primary inputs to the LUT input and from the LUT output to the primary output. The basic intuition behind using this heuristic is that, by choosing the shortest paths we reduce the probability that the response from (patterns to) a fault-free LUT gets corrupt due to the presence of a faulty LUT on the path from the LUT output (primary input) to the primary output (LUT input). Also, the shortest path can be computed easily using well-known algorithms [Cormen 89] in time polynomial in the number of LUTs in the network.

The existence of multiple paths from an LUT output (primary input) to the primary output (LUT input) has some added advantages. Consider a particular LUT *i*

Table 6.1 Experimental Results.

Design	# LUTs	# reconfiguration sessions	Total # LUT reconfigurations	Average # LUT reconfigurations per session
example1	640	640	1525	2.38
example2	41	41	90	2.19
example3	587	587	1310	2.23
example4	172	172	513	2.98
example5	64	64	176	2.75
example6	56	56	139	2.48
example7	176	176	513	2.91
example8	680	680	1590	2.34
example9	141	141	423	3
example10	107	107	299	2.79
example11	70	70	184	2.62
example12	82	82	219	2.67
example13	70	70	188	2.68
example14	88	88	263	2.98
example15	88	88	253	2.87
example16	86	86	237	2.75
example17	96	96	279	2.9
example18	104	104	302	2.9
example19	101	101	293	2.9
example20	108	108	318	2.94
example21	118	118	364	3.08
example22	71	71	196	2.76
example23	89	89	245	2.75
example24	194	194	577	2.94
example25	84	84	215	2.55

having m distinct paths from its output to the primary output. Suppose that, after the first step, the faulty LUT could not be located but it was found that a path p containing LUT i is faulty. If we test the LUT exhaustively and propagate the results along each of the m paths by configuring the corresponding LUTs, then, in the second step, we have to examine only the LUTs which lie on *all* these m paths. Thus, there is a high possibility that the faulty LUT is located in the first step (after considering all the paths from LUT i), or the second step will be applied to a very small set of LUTs. This enhances the quality of the algorithm. However, this scheme may be expensive because it spends more time during the first step. For our implementation, we considered only a single path (the shortest path) instead of considering all possible paths. Although we considered only LUT faults, some interconnect faults (stuck faults on LUT inputs or outputs) are also covered by our technique.

6. Experimental results

In Sec. 4.A, we have derived bounds on the total number of reconfigurations, over all reconfiguration sessions for fanout-free LUT networks. Each time an LUT is tested, some LUTs have to be reconfigured. This partial reconfiguration constitutes a *reconfiguration session*. However, it is difficult to derive similar bounds for general LUT networks. So, we have conducted experiments on a set of general LUT networks. We generated the LUT networks from the MCNC combinational logic benchmarks by mapping each individual output function to FPGAs. We used the *act_map* command in Sis [Sentovich 92] to generate the networks and treated the blocks as LUTs. Since this command targets Actel FPGAs, we also generated LUT networks for Xilinx XC3100 FPGAs using the Synopsys FPGA compiler [Synopsys 98]. Table 6.1 reports the total number of reconfigurations over all reconfiguration sessions for some of these networks. Note that, the number of reconfiguration sessions is always the same as the number of LUTs in the given network.

In Table 6.1, we have also reported the average number of LUTs reconfigured per reconfiguration session which is computed by dividing the entry in column 4 of the table by the entry in the column 3. This particular metric is extremely important in order to measure the reconfiguration complexity of any fault location technique. In conventional FPGAs (with no partial reconfiguration capability), the major goal of any such technique should be to minimize the number of reconfiguration sessions because in each such session all the logic blocks in the FPGA are configured even if we want to change the configuration of a single logic block. However, for FPGAs with partial reconfiguration capability, since we can reconfigure a single logic block without having to reconfigure the remaining logic blocks, the metric of reconfiguration complexity should be the total number of reconfigurations (each reconfiguration implies reconfiguration of a single logic block) performed during the execution of the entire technique. As shown in Table 6.1, the total number of reconfigurations is equal to 2 to 3 times the total number of LUTs in the networks; i.e., on average, around 2 to 3 LUTs need to be reconfigured in each reconfiguration session. This shows the effectiveness of our technique when partially reconfigurable FPGAs (e.g. the Xilinx 6200 and the Atmel AT6000 families) are used in the system.

7. Conclusions

In this paper, a novel fault location technique for FPGA-based reconfigurable systems is presented. This technique takes advantage of partial reconfiguration when it is available. The method uses the pseudo-exhaustive BIST technique and is hence thorough. Moreover, the technique is in-place and preserves the original interconnection structure of the LUTs. The technique has an extra advantage that it is not necessary to bring the whole system down while fault location is carried out. It is a two-step approach, where the first step does not use any specific fault model. However, cell-stuck fault model is used for the cases where the second step is needed. Moreover, the technique is adaptive during the second step. For fanout-free LUT networks, it can be proved that the upper bound on the total number of LUT reconfigurations is linear in the number of LUTs. Experimental results show that this number is linear (2 to 3 times) in the total number of LUTs even for general LUT networks. This implies that, in a system with partial reconfiguration capability, only a few LUTs need to be reconfigured during each session. These reconfiguration steps can be generated during the compilation of the original design.

The technique has a disadvantage that all the LUTs in the network have to be tested during the first step of our technique (Algorithm 1). However, this disadvantage can be handled by making the first step of our technique adaptive. In that case, we stop when we obtain a faulty response during the first step (Algorithm 1) and apply Algorithm 2 on the path along which we observed the response of the LUT that produced faulty response. For multi-output circuits, our technique can be further enhanced to allow exhaustive test of multiple LUTs at the same time along independent paths.

8. Acknowledgments

This work was supported in part by the Ballistic Missile Defense Organization, Innovative Science and Technology (BMDO/IST) Directorate and administered through the Department of Navy, Office of Naval Research under Grant No. N00014-92-J-1782, the Advanced Research Projects Agency Contract No. DABT-94-C-0045 and the Defense Advanced Research Projects Agency (DARPA) under Contract No. DABT63-97-C-0024. The authors would like to thank Dr. Nirmal Saxena of Stanford CRC for his helpful comments regarding this work.

9. References

[Atmel 97] Atmel Corp., *AT6000 Series Configuration*, 1997.

[Cormen 89] Cormen, T. H., C. E. Leiserson and R. L. Rivest, *Introduction to Algorithms*, The MIT Press and McGraw-Hill Book Company, 1989.

[Friedman 73] Friedman, A.D., “Easily Testable Iterative Systems,” *IEEE Trans. Comp.*, Vol. C.22, No. 12, pp. 1061-1064, Dec. 1973.

[Huang 96] Huang, W.K., X.T. Chen, and F. Lombardi, “On the Diagnosis of Programmable Interconnect Systems: Theory and Application,” *IEEE VLSI Test Symp.*, pp. 204-209, 1996.

[Inoue 98] Inoue, T., S. Miyazaki and H. Fujiwara, "Universal Fault Diagnosis for Lookup Table FPGAs," *IEEE Design and Test of Computers*, pp. 39-44, January-March, 1998.

[Jordan 93] Jordan, C., and W.P. Marnane, “Incoming Inspection of FPGAs,” *Euro. Test Conf.*, pp. 371-377, 1993.

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

[Liu 95] Liu, T., F. Lombardi, and J. Salinas, "Diagnosis of Interconnects and FPICs Using a Structured Walking-1 Approach," *IEEE VLSI Test Symposium*, 1995, pp. 256-261.

[Lombardi 96] Lombardi, F., D. Ashen, X. Chen, and W.K. Huang, “Diagnosing Programmable Interconnect Systems for FPGAs,” *Int’l Symp. on FPGAs*, pp. 100-106, 1996.

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

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

[McCluskey 85] McCluskey, E.J., “Built-In Self-Test Techniques,” *IEEE Design and Test of Computers*, pp. 21-28, April 1985.

[McCluskey 86] McCluskey, E. J., *Logic Design Principles with Emphasis on Testable Semicustom Circuits*, Prentice-Hall, Eaglewood Cliffs, NJ, USA, 1986.

[Renovell 97] Renovell, M., J. Figueras, and Y. Zorian, "Test of RAM-Based FPGA: Methodology and Application to the Interconnect," *IEEE VLSI Test Symp.*, pp. 230-237, 1997.

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

[Rupp 98] Rupp, C. R., et al., “The NAPA Adaptive Processing Architecture,” *Proc. IEEE Symp. FCCM*, 1998.

[Saxena 98] Saxena, N. R., and E. J. McCluskey, “Dependable Adaptive Computing Systems: The ROAR Project,” To appear in *Proc. IEEE Int’l Conf. on Systems, Man and Cybernetics*, October 1998.

[Sentovich 92] Sentovich, E. M., *et. al.*, “SIS: A System for Sequential Circuit Synthesis,” ERL *Memo. No. UCB/ERL M92/41*, Department of EECS, UC Berkeley, 1992.

[Stroud 96] Stroud, C., S. Konala, P. Chen, and M. Abramovici, "Built-In Self-Test of Logic Blocks in FPGAs (Finally, A Free Lunch: BIST Without Overhead!)," *IEEE VLSI Test Symposium*, 1996, pp. 387-392.

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

[Synopsys 98] *Synopsys Design/FPGA Compiler Version 1998.02*, Synopsys, February, 1998.

[Udell 87] Udell, J. G. Jr. and E. J. McCluskey, "Efficient Circuit Segmentation for Pseudo Exhaustive Test," *Proc. ICCAD*, pp. 148-151, 1987.

[Wang 97] Wang, S. J., et al., "Test and diagnosis of faulty logic blocks in FPGAs," *Proc. ICCAD*, pp. 722-727, 1997.

[Xilinx 96] Xilinx Inc., *The Programmable Logic Data Book*, 1996.