

Dynamic Fault Tolerance in FPGAs via Partial Reconfiguration

John Emmert[†], Charles Stroud, Brandon Skaggs
Dept. of Electrical Engineering
University of Kentucky
[†]emmert@ieee.org

Miron Abramovici
Bell Labs - Lucent Technologies
Murray Hill, NJ

Abstract

*In this paper we present an on-line, multi-level fault tolerant (FT) technique for system functions and applications mapped to partially and dynamically reconfigurable FPGAs. Our method is based on the roving self testing areas (STARs) fault detection/location strategy presented in [1]. In STARs, the area under test uses partial reconfiguration properties to modify the configuration of the area under test without affecting the configuration of the system function and dynamic reconfiguration properties to allow uninterrupted execution of the system function while reconfiguration takes place. In this paper, we take this one step further. Once a fault (or multiple faults) is detected we dynamically reconfigure the working area application around the fault with no additional system function interruption (other than the interruption when a STAR moves to a new location). We also apply the concept of partially usable blocks to increase fault tolerance. Our method has been successfully implemented and demonstrated on the ORCA 2CA series FPGAs from Lucent Technologies.**

1. Introduction

Recent advances in fabrication technology and device architecture have resulted in tremendous growth in FPGAs, both in terms of density and performance. Currently commercially available devices can map up to three million or more gate equivalent designs[2][3]. Recent research has demonstrated the ability to clock FPGA circuits at 250 MHz[4]. As device sizes increase, the probability of faults occurring during the lifetime of an FPGA increases. Additionally, harsh environments also contribute to the degradation of FPGA parts. The fault tolerant (FT) problem for FPGAs is defined as reconfiguring an FPGA to operate correctly in the presence of faulty FPGA resources (programmable logic, interconnect, and configuration memory). When faulty blocks are located during execution, fault tolerance should allow the performance of the application to degrade gradually and gracefully. Of course, faults can be tolerated only if they are detected and accurately located. While most prior work treats fault-tolerance separately from fault detection and diagnosis, our approach is the first that integrates on-line test, diagnosis, and fault tolerance in

the same framework.

In this paper we present a multi-level, on-line FT technique for incrementally reconfiguring applications mapped to FPGA circuits. Our approach is specifically targeted toward faults that appear during the lifetime of the chip and are caused by aging (electromigration, corrosion, etc.) and/or environmental factors (temperature, humidity, vibrations, cosmic radiation, α -particles, etc.). For transient faults, our system will use a low overhead method such as concurrent error detection[5]. Additionally, our system detects dormant faults that do not affect the current system function which is an essential feature for high reliability systems[6]. There are also several *assumptions* associated with our method. The system operation will have short breaks to allow for moving our STARs to new locations, but it may not be taken off-line. The application timing has some slack to allow for additional routing delay due to roving and changes for fault bypassing. Finally, the FPGAs our method is applied to will have boundary scan and run-time partial reconfiguration features.

Our method uses the roving **Self-Testing AR**eas (STARs) approach for fault detection/location[1]. In this approach the STAR areas, small roving portions of the FPGA, are off-line and being tested while the rest of the device is on-line, continuing normal system operation. Run-time reconfiguration (RTR) and partial reconfiguration capabilities allow configurations used by STARs to be downloaded while the rest of the device continues normal, uninterrupted system operation. When testing a STAR is complete, the STAR moves to a new location (roving). This is accomplished by *copying* a portion of the working system function into the region previously occupied by the STAR (this step requires no interruption of the system function). Then the system clock is stopped for a few cycles while the *state* of the system function is transferred to the copied region, after which the system clock is restarted. Now the STAR occupies the now vacated portion of the working area (this step requires no interruption of the system function). This is illustrated in Figure 1a, where the horizontal and vertical STARs (H-STAR and V-STAR) are used for fault detection and diagnosis while the working area performs the intended system function.

When faults are detected, they are detected in a STAR. Since the system function is spatially separate from the STAR, it continues to execute when faults are detected.

* This material is based upon work supported by the DARPA ACS program under contract F33615-98-C-1318.

However, the configuration of the system function may be affected when the STAR moves off of the faults. Therefore, our FT method incrementally reconfigures the working area around faulty programmable resources before the working area is remapped during the STAR roving process. As a result, our method requires no additional down-time (beyond the clock cycles previously discussed for moving the STAR) of the system function executing in the working area of the FPGA. The paper is organized as follows. In Section 2 we briefly describe previous FT research for FPGAs. In Section 3 we provide a more detailed overview of the roving STARs approach to fault detection/location. In Section 4 we present the terminology and coordinate systems used in our problem formulation and solution descriptions. In Section 5 we formally define our FT problem, and in Section 6 we describe our multi-level FT technique and our current implementation. In Section 8 we present conclusions and directions for future work.

2. Related Research

In their FT technique, Kumar et al. used adaptive customization to avoid defective sections and artificially enhance a programmable gate array's yield [7]. Their technique was strictly off-line and fault tolerance was achieved by bypassing faulty programmable resources at the time the circuit was initially mapped to the FPGA. They described a two step process to achieve defect tolerance: test of an unprogrammed device to locate defective components and program the device to avoid the defective portions. They presented heuristics for adaptively programming an FPGA in the presence of faults.

Hatori et al. introduced redundancy as a FT method for FPGAs [8]. Their method used specialized selector circuitry to reconfigure FPGA circuits in the presence of faults. Similar to methods used for fault tolerance in SRAMs, an additional column of programmable logic blocks is added to the FPGA. In the event that a logic block was faulty, all functions mapped to the column where the faulty block was located were shifted toward the spare column. All subsequent columns of functions between the column with the faulty block and the spare column were also shifted a distance of one column toward the spare column. They did not

detect faults, only reconfigure around them. Their selector circuits required approximately 2% additional area overhead, and they required spare columns. Since they eliminated an entire column for one fault, their fault tolerance was low, and they estimated a performance degradation of 5% when compared to similar non-fault tolerant FPGAs.

Durand and Piquet proposed a FT FPGA architecture with the ability to repair itself in the presence of faults [9]. They used a special multiplexer with self repair and self diagnosis capabilities. Their special circuitry was capable of detecting logic failures during runtime, and the architecture had a limited capability to reconfigure itself dynamically. Their FT technique allocated two extra columns of cells, and in the event that a fault was detected, they bypassed the faulty column by shifting toward the spare column. Since the author's reported that more than complete logic redundancy was required for their method, the area overhead for fault tolerance exceeded 50%.

Narasimhan et al. developed a FT technique for PGAs or Wafer Scale Integrated Arrays [10][11]. They use a pebble shift algorithm to reconfigure around faulty blocks. Their method is flexible. It is not limited to one fault per row, column, or tile. However, it must be performed off-line, and it requires the application cease execution for reconfiguration.

Howard and Tyrrell described ideas for increasing yield on FPGAs [12]. Their FT idea was to insert extra bypassing columns or rows into the FPGA. They looked at both removing an entire row (column) if it contained a fault and removing only the faulty block from the row. Their methods were strictly for yield enhancement and the faults were bypassed at the time the configuration for the FPGA was downloaded. They also introduced global *long* lines to reduce the performance hit associated with reconfiguration.

Kelly and Ivey used redundancy to detect faults in applications mapped to FPGAs [13]. Their FT technique used a shift method to reconfigure in the presence of faults. They incorporated a *reconfiguration switch* to reconfigure and used normal place and route tools for mapping circuits to FPGAs. A step-based switch configuration algorithm was used to configure the special switches around faulty elements.

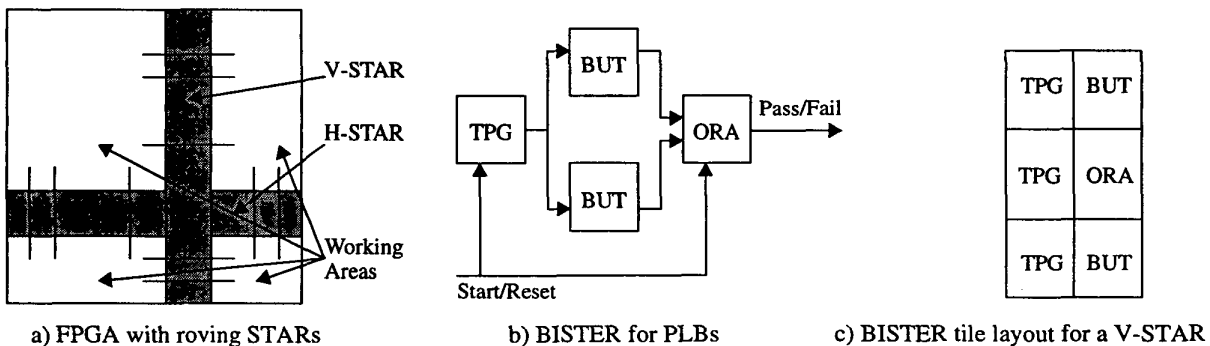


Figure 1. On-line roving self-test areas (STARs).

Cuddapah and Corba used Xilinx SRAM based FPGAs to demonstrate the FT capabilities of FPGAs [14]. In their study, they randomly picked logic blocks to be faulty. Then they reconfigured the circuit around these faults using commercially available tools. The main contributions of their work were an algorithm to determine fault coverage of a design and a definition of the fault recovery rate for any given design implemented in an SRAM-based FPGAs. Additionally, they demonstrated that fault recovery was feasible on FPGAs by other than modular redundant methods.

Dutt and Hanchek et al. developed a method to increase FPGA yield [15][16]. Their method uses node covering and reserved routing resources to replace the functionality of faulty cells. One row (column) of cells is reserved for spares. If a cell in any given column (row) goes bad, the functionality of all cells in the column (row) from the faulty cell to the spare cell is shifted toward the spare cell. Spare routing resources are used to eliminate overhead of re-routing the updated circuit placement. Relative to covering interconnect faults, they use a similar idea where spare interconnect resources are allocated as a grid, and when a fault occurs, it is replaced by an extra segment or grid of segments. The main advantage of this method is that it is very fast. Since the spare resources have already been allocated to cover a limited number of faults, the reconfiguration time is linear with respect to the number of faults. There are several issues with this methodology. The most significant problem is the limitation of the fault coverage. The number of faulty cells that can be covered is limited to one in each column (row). If two faults were found in the same row, this method breaks down. There is also an associated resource overhead. For an $N \times N$ array of cells, at least N cells must be left unused. There is also a routing overhead caused by the reservation of routing resources. Additionally, when cells are moved there is an associated performance degradation associated with increasing the length and number of programmable interconnect resources.

Mahapatra and Dutt proposed a dynamic method for allocating interconnect resources to bypass faults only after faults have been located [17]. If the new required segments conflict with the current usage of routing tracks, the layout is incrementally modified to make room for the new segments. This method is dynamic only with respect to the interconnect resources, as all spare programmable logic blocks (PLBs) are statically allocated.

Emmert and Bhatia developed a FT technique for incrementally reconfiguring FPGA mapped circuits around faulty programmable resources [18][19]. They used min-max grid matching, to match faulty PLB locations to unused spare PLB resources. Then they incorporated a shift methodology to shift PLBs between the faulty PLB and its matched spare location toward the spare location. For interconnect faults they introduced the idea of an embedded incremental router to route around faulty interconnect resources. The router uses read write capability of the

FPGA configuration memory to rip-up and reroute without a netlist. While their method for reconfiguring around faulty PLBs was applied to the Xilinx FPGA, their FT technique for interconnects was applied to an artificial FPGA architecture, not to a commercial FPGA.

Lach et al. developed low overhead FT systems [20]. Their basic approach was to partition a design into a number of tiles. Each tile was allocated a number of spare PLBs. In the event a fault in a tile was found, a spare PLB was used to replace the faulty PLB. Their reconfiguration approach provided multiple configurations for each tile. In the event that a fault was detected, the configuration associated with the fault was downloaded. The main drawback of this method is low fault tolerance. It is limited to one fault per tile. If multiple faults occur in close proximity (within a tile), this method breaks down. There is no ability to draw spares from other tiles to cover several faults in a single tile.

With the exception of Lach et al., all previous methods rely on reconfigurations around faults to be calculated online. This requires fast computations in order to maintain high availability for the fault tolerant system. The precompiled configurations of Lach et al. on the other hand significantly reduce system down time.

3. The Roving STARs

Since the normal system operation in adaptive computing systems involves reconfiguring FPGAs for different functions, we assume that the reconfiguration process is controlled by a module external to the FPGAs; typically this is an embedded microprocessor having some storage medium for the various FPGA configurations and partial configurations. Because the physical placement of the system functions in the FPGAs changes in time, this module keeps track of the functions currently implemented in different FPGAs and in different sections of the same FPGA, and maintains all configurations that may be used in the future. We extend the tasks of this processor to also control the test, diagnosis, and fault-tolerance functions, including their associated reconfigurations. We refer to this processor as the Test and Reconfiguration Controller (TREC). The TREC accesses the FPGA using its boundary scan interface so that this access is transparent to the normal system function (most current FPGAs support reconfiguration and access to the FPGA core via boundary scan). Partial reconfiguration allows localized configuration of the FPGA functionality. Therefore we eliminate any change of the system configuration when changing the STAR configuration for different test modes. Also, we minimize the amount of system reconfiguration required when the STARs move to new locations.

The RTR lets us change the configuration of the STAR area for different test modes without interfering with system operation. The TREC uses RTR to rove STARs across the FPGA and to reconfigure the STARs for different test and diagnosis operations. The TREC partially reconfigures and initiates Built-In Self-Test (BIST) within the STARs and

analyzes their test results at the conclusion of each BIST sequence[1]. If faults are detected, the TREC executes a diagnostic process based on dividing the set of suspected PLBs or wires and merging them into BIST circuit configurations which use fault-free resources. The TREC also keeps track of the status of FPGA resources, specifically any faulty resources, so that defective hardware can be avoided when part of the working area is relocated in a previous STAR. If a faulty PLB or a faulty wire may not be used, then the TREC determines configuration changes to the working area so that the defective resources are bypassed and replaced by fault-free ones; this implements the basic fault-tolerant function.

The floorplan of the FPGA consists of the working area containing the system function, a vertical STAR (V-STAR) consisting of two columns, and a horizontal STAR (H-STAR) consisting of two rows[1]. Depending on the positions of the two STARs, the working area may be contiguous, or it may be divided into two or four disjoint regions. Only the connections among the working PLBs and the I/O blocks of the FPGA are allowed to go through the STARs, using horizontal wire segments through V-STAR and vertical segments through H-STAR as illustrated in Figure 1a. The V-STAR roves across the FPGA and back again, while H-STAR roves up and down the FPGA. A STAR tests both the PLBs and the programmable interconnect in its area. While either STAR can be used to test the logic or PLBs, both STARs are required for testing the programmable interconnect[1].

We configure a STAR as several independent BIST structures which we refer to as BISTER tiles. The structure of a BISTER tile is shown in Figure 1b. The test pattern generator (TPG) applies pseudo-exhaustive test patterns to two identically configured programmable logic blocks under test (BUTs), whose outputs are compared by an output response analyzer (ORA). (Note that pseudo-exhaustive testing means every subcircuit is exhaustively tested.) The ORA latches and reports any mismatches as test failures. Direct comparison by the ORA avoids the excessive overhead involved in on-chip storing of expected responses, and also avoids possible errors introduced by signature aliasing caused by compressing responses into signatures. The BUTs are repeatedly configured to be tested in all their programmable modes of operation (such as adder/subtractor, flip-flops/latches, RAM, LUTs, etc.). Figure 1c shows a possible floorplan of the BISTER tile using 3 PLBs for the TPG, 1 PLB for the ORA, and 2 BUTs. Once the BUTs have been completely tested, the BISTER tile layout is rotated and the next pair of BUTs are tested in all of their modes of operation. We use a rotation strategy that repeatedly reconfigures the PLBs in a BISTER tile until each PLB has been tested twice, each time being compared with a different BUT. As a result of this strategy, our approach is capable of detecting almost any possible combination of faulty PLBs[1].

If all of the PLBs in a STAR have been tested and found

to be fault-free, the STAR roves to the next position by relocating a portion of the working area over the current STAR position. On the other hand, if faults are detected in the STAR, additional configurations are downloaded to diagnose the faulty PLB(s). We use an adaptive diagnostic strategy which downloads additional configurations to diagnose faulty PLBs based on the failing test results. In addition to locating single and multiple faulty PLBs, the purpose of the diagnostic configurations is to determine the faulty as well as the fault-free modes of operation of a defective PLB. Note that this diagnostic resolution allows a new level of fault tolerance, the partially usable block, that will be described in Section 6. Once the diagnostic process is complete, our FT technique reconfigures the working area around the diagnosed faults.

In summary, our approach provides an on-line method for detecting and locating faulty FPGA programmable resources. By removing the area under test from the working circuit and then roving this area under test across the FPGA, we allow the working circuit to continue executing the system function while testing and reconfiguration are underway. Therefore, if faults are found, they are found in the non-working area of the FPGA such that we can reconfigure without affecting the execution of the system function in the working area of the FPGA. We use pseudo-exhaustive tests so the FPGA is completely tested in all modes of operation. Our method covers faults for the entire FPGA including the programmable interconnect network[1]. However, the focus of this paper is limited to fault tolerance for the PLBs.

4. The Virtual FPGA

In this section we describe the coordinate systems (first the physical, then the virtual) used to keep track of the system application function during the roving process. We define the system or application function as being composed of one or more *logic cell functions*, where each logic cell function will fit in one PLB on the FPGA. We define the set of logic cell functions that make up the system function as F , where each element of F will fit in one PLB. The FPGA is composed of an $N \times N$ array of PLBs. We define the set of PLBs as P such that each element of P has a fixed physical location (defined by its row and column) on the FPGA. An example physical coordinate system for $N = 8$ is shown in Figure 2a where the H-STAR location is defined by its physical FPGA row $i \in \{1, 3, 5, \dots, N-1\}$ (the H-STAR uses rows $i=5$ and $i+1=6$), and the V-STAR is defined by a physical FPGA column $j \in \{1, 3, 5, \dots, N-1\}$ (the V-STAR uses columns $j=5$ and $j+1=6$).

During normal operation, the FPGA is divided into two disjoint sections: the test area (composed of the H-STAR and V-STAR) and the working area (where the system function is located). Since the locations of the H-STAR and V-STAR periodically change, so does the location of the working area. Depending on the location of V-STAR and H-STAR, the working area may be split by the STARs as

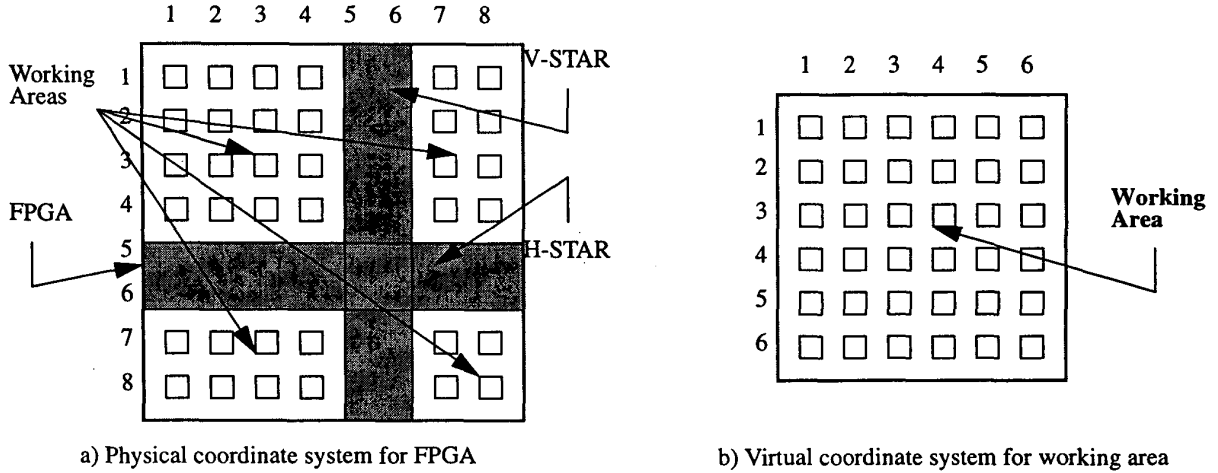


Figure 2. Physical and Virtual Coordinate System for FPGA

shown in Figure 2a. (It should be noted that the test and working area are not completely disjoint since some of the working area interconnects must go through the STARs.) The standard placement step of the mapping process for mapping system functions to FPGAs (FPGAs without STARs) will assign each logic cell function a specific location (or PLB) on the FPGA. For our application the roving STARs require that a single logic cell function could be mapped to several PLBs depending on the STAR locations. To simplify mapping and provide a framework for the system application that is independent of the STARs, we introduce the virtual coordinate system.

The virtual coordinate system describes the physical coordinate system without the STARs. It is composed of an $(N-2) \times (N-2)$ array of virtual logic cells (see example for $N=8$ in Figure 2b). Similar to the physical coordinate system, a single logic cell function can be mapped to a single virtual logic cell. While virtual logic cells may be separated by STARs in the physical coordinate system, they provide a reference framework by which standard place and route tools can map system logic functions to the working area of the FPGA. Additionally, the virtual coordinate system provides the necessary framework for our FT problem and solution. After placement by standard PAR, each element of F is assigned a location in the virtual coordinate system, and except when moved for fault tolerance, the virtual location of the logic cell function will not change. However, it should be noted that its physical location will change relative to the physical locations of the H-STAR and V-STAR. The virtual framework in Figure 2b defines the location of each system logic cell function in F relative to all other elements of F .

Therefore, given an element $p_i \in P$, to determine if an $f_j \in F$ occupies p_i , the converse is required. We must know which elements of F occupy a given physical PLB location. Depending on the location of the STARs, up to four different elements of F can occupy a given PLB. This concept is

demonstrated in Figure 3 where, depending on the locations of the STARs, up to four elements of F could occupy a given physical PLB.

5. Problem Formulation

Our logic FT problem is summarized as follows: given a set of faulty or partially faulty PLBs and a set of system logic cell functions, ensure that no system logic cell function ever falls on a faulty, unusable PLB. The faulty PLBs are elements of P and have locations defined by the physical coordinate system. The system logic cell functions are elements of F and have locations defined in the virtual coordinate system. It should be noted that elements of P may not be completely faulty and elements of F may not use all of the functionality available in a virtual logic cell.

A novel aspect of our FT concept is the **partially usable block (PUB)** which allows faulty PLBs to be safely used whenever the faults do not effect the intended operation of system function to be placed in that PLB. This is possible because the testing and diagnostic configurations are capable of identifying fault-free (as well as faulty) modes of operation of a faulty PLB. Reusing defective hardware resources leads to more graceful degradation and longer mission life by increasing the effective spare capacity. For example, a PLB with faulty flip-flops may still be safely used for combinational logic. Therefore, the FT problem is further complicated by the fact that a given faulty $p_i \in P$ may not be completely faulty, and whenever possible, the PUBs should be used so that performance of the system function will be minimally impacted by any required reconfiguration.

6. Proposed Current Implementation

In this section, we describe our FT approach. Unlike conventional FT systems, an important feature of our approach is that it does not interrupt the normal system operation after a fault has been detected and located. This is possible

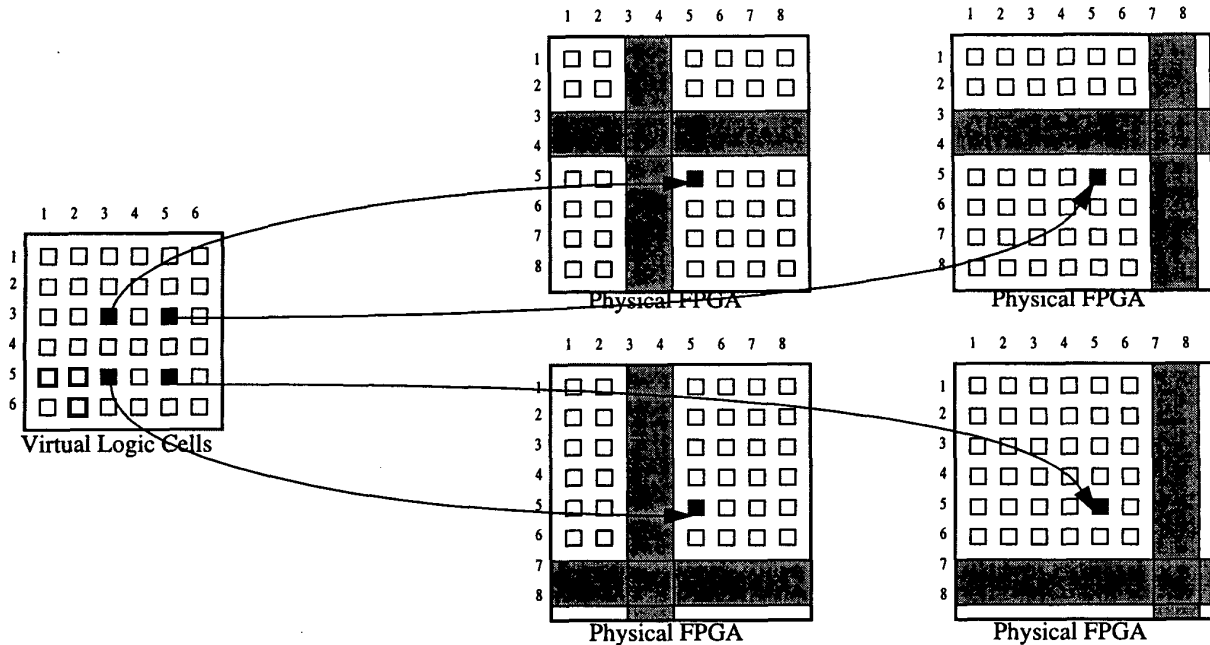


Figure 3. Example of virtual logic cells that can occupy a physical PLB

because the newly detected faults are found in a STAR and thus do not affect the current system logic. However, it should be noted that when a fault is detected, the operation of the system since the last time the same area was tested (and found fault-free) until the present time, could be corrupted. Depending on the application, normal operation at this point may consist of a roll-back operation that restores the state of the system to the last state saved before the fault occurred.

Relative to system logic, we provide three levels of fault tolerance as will be discussed in detail in Section 7. In general, at the first level of fault tolerance, *STAR Parking*, no system function reconfiguration is required. In this level we use roving STARs to temporarily cover the defective PLBs so that both the normal system operation and a scaled-down test process are allowed to continue. While in the first level of fault tolerance, the TREC is determining the configurations required for the second level of fault tolerance without requiring the system function to stop executing. In the second level of fault tolerance, STARs are allowed to move off of the defective PLBs. For this level, spare resources from the working area are used (only when required) in the system function so it avoids unusable defective PLBs as the STARs rove across the FPGA. This requires a spare allocation strategy that can be dynamically changed (by remapping the system function) in order to bring more spare PLBs into areas of the FPGA that contain more faults. The third level of fault tolerance takes effect when no more spare resources are available in the working area. In this level, spare resources used by the system function are stolen from the STARs. The rest of this section is divided into three parts. In Section 7 we describe in detail all three levels

of our approach. In Section 7.1 we describe some possible spare allocation strategies that can be used with the second level of our FT strategy (note: these strategies are *not* required for our FT method--other strategies are applicable). In Section 7.2 we describe our current implementation.

7. Multi-Level Fault Tolerant Approach

In this subsection we describe our multiple stages or levels of fault tolerance for faulty PLBs in the order that they are applied during on-line operation. The first two levels are the most advantageous relative to the performance of the system function and the ability to maintain complete testing capability. We start at the point where the roving STAR has detected a fault. Once a fault is detected, we first set a flag to notify the system application operational software, in case any system data check or roll back is desired or required. (Note: our STARs approach will detect faulty PLBs only in the region covered by the STARs. Therefore, we build the set of faulty PLBs incrementally. Our reconfiguration approach is also incremental since it handles elements of the set of faulty PLBs as they are found.) At this point we are applying the first possible level of our FT approach as described below.

Level 1: When one or more defective PLBs have been detected in the currently roving STAR, the system operation can continue without interruption. This is because the faults are located in the STAR, not the working area. This fact leads to our first level of fault tolerance **STAR parking**. In STAR parking, we always have at least one of the STARs over the newly detected faulty PLB(s) while the other contin-

ues to rove and test. In this stage no reconfiguration for fault tolerance of the system function is required. (However, reconfigurations for level 2 can be determined while in level 1.) We use the non-roving STAR to locate the exact PLB(s) that is(are) faulty and to diagnose if all or just part of the faulty PLB(s) is(are) faulty (if the faulty PLB(s) is(are) partially usable then we have potential PUBs). This approach on-line because the application or system function can continue to execute while fault detection/location and any required generation of fault tolerant configuration replacement sections (FTCRS) takes place. Once diagnosis is complete, TREC knows the extent of the fault in terms of the number and the position of the defective PLBs, and also their faulty and good modes of operation. So, we mark the appropriate portion(s) of the PLB(s) faulty.

While in level 1, we have two possibilities for roving: double STAR roving and single STAR roving. If the recently detected faulty PLBs are restricted to a 2x2 tile of PLBs, then *either* V-STAR *or* H-STAR could be used to cover the faulty PLBs while the other STAR continues to rove. This allows us to continue roving with both STARs; however, only one can rove while the other covers the newly detected faulty PLB(s). This provides us with complete logic test capability and slightly degraded interconnect test capability. (Only crosspoint switches that connect horizontal and vertical global routing resources will not be tested [1]). If the recently detected defective PLBs (multiple faulty PLBs outside of a 2x2 PLB area) can be covered only by one STAR, this STAR must remain *parked*. Testing will continue with only the other STAR roving. This process will still test all PLBs, but the interconnect resources that used to be tested by the now *parked* STAR will not be tested during this period. Note that testing during this period should ignore the current set of defective PLBs such that only new faults are processed.

Level 2: In level 2, the system function is reconfigured to avoid unusable faulty PLBs, and the STARs move off of the defective PLBs so full roving can continue. In level 2, the system function uses various configurations to avoid faulty unusable, defective PLBs. As the system function moves to an area where a totally faulty or partially usable PLB (PUB) exists, one of several things may occur. If a logic cell function falls on a faulty PLB that is a PUB, and the logic cell function can use the non-faulty part of the PUB, then the logic cell function is left where it is in the virtual coordinate system such that it stays mapped to the PUB. If the logic cell function falls on a defective PLB that it cannot use, it is shifted to its spare location in the virtual coordinate system. This will work as long as 1) spares are available and 2) no two faults need the same spare. For the second case, we perform additional

reconfiguration to use other spares available in the virtual coordinate system. For the second case where no more spares are available in the virtual coordinate system, we move to level three of fault tolerance or *STAR stealing*. Processing for the second level of fault tolerance is performed by the TREC while the system is in the first level of fault tolerance. (Future precompiled configurations for level two can also be calculated while in level two.)

In level two, defective resources are avoided by moving some of the system logic cell functions to avoid unusable faulty PLBs. For example, in level two let D be a newly diagnosed defective PLB. Recall that up to four different working PLBs may be hosted by D at different times (see Figure 3). If the good modes of operation of D allow it to correctly perform these functions, then D needs no replacement. For example, if D has faulty flip-flops, it may be used as a PUB whenever the logic cell function to be located in D is combinational or uses the LUT/RAM module as RAM. Note that a PUB extends the effective spare capacity of the FPGA, because it avoids using one of the fixed spares to bypass the fault. If all the newly diagnosed defective PLBs can serve as PUBs for all four roving directions, then no changes are needed, and the roving process is allowed to proceed. (Note that relocating a fixed spare over a defective PLB is always possible.)

In level two, there are several ways to determine the partial configurations required to avoid faults. For single faults occurring between long intervals of time, **precompiled** FTCRS can be stored to quickly map around faulty PLBs. For this implementation, we precompile a FTCRS for each system logic cell function mapping it to a spare virtual logic cell. The number of FTCRS is $O(m)$ where m is the number of system logic cell functions in F . By using the partial reconfiguration capability of FPGAs, the size of these FTCRS is kept small and therefore requires little memory storage relative to the amount required for storing complete FPGA configurations. For multiple faults when the FTCRS determined for the single faults interfere with one another (for example routing cross-overs) processing for FTCRS can be done **online** by the TREC. This takes longer than using the precompiled FTCRS, but it does not interfere with the system function execution since the a STAR is currently over the fault (see level 1). After several faults have occurred and several iterations of reconfiguration have taken place, it may become necessary to remap the entire system function and reallocate extra spare PLBs in the vicinity of high concentrations of defective PLBs. Processing for this is done off-line by the TREC, and it requires the entire system configuration file be reloaded.

Level 3: This level is referred to as **STAR stealing**. At

this level the fixed spares can no longer be used for fault-tolerance, most likely because all of them have already replaced defective PLBs. Then we use some of the roving spares provided by the two STARs by allocating them to replace defective PLBs. However, the PLBs taken out from the STAR will not be spares any longer. The extent of testing may be reduced with every new fault since some tiles will no longer be able contain BISTERS, and not enough fault-free PLBs may be available to allow accurate diagnosis. Also more interconnect resources will no longer be testable. If there is a choice from which STAR to take PLBs, we will try to maintain at least one STAR roving for as long as possible. It should be noted that the introduction of the concept and use of PUBs increases the spare capacity of the working area, thereby delaying entry into level 3 for as long as possible.

7.1 Spares

In this subsection we describe two possible spare allocation strategies for the working area of the FPGA that we are currently investigating. In the first strategy, we guarantee that for working area PLB utilization less than 80% (which is typical for most applications), every system logic cell function will be adjacent to at least one spare virtual logic cell. For the second strategy, for a working area PLB utilization less than 92%, every system logic cell function is no further than one PLB from a spare.

Initial placement and routing of the system logic is done in the virtual $(N-2) \times (N-2)$ array. For our first strategy, we will constrain the design so that at least 20% of the working area PLBs are left as fixed virtual spares. This is not excessive, since typical PLB utilization is around 80% for most applications. Higher utilizations make the design difficult to route. We can show that if this constraint is satisfied, we can always find a placement of the system logic so that every system logic cell function is adjacent to at least one spare virtual logic cell. Figure 4a illustrates such a placement for a 12x12 FPGA. To force standard design tools to generate

evenly distributed spares, we reserve the location of spare cells by preplacing dummy PLBs before executing the standard placement algorithm. In our implementation, for each system logic cell function, we select one of its adjacent spares as its preferred replacement. Note that the same spare may be designated as the preferred replacement for several working PLBs. In Figure 4, the following equations were used for determining if a position with virtual coordinate r, c in the working area should be reserved as a spare. If we let $\delta = r \pmod{5}$ and if $c \pmod{5} = (2 \times \delta + 1) \pmod{5}$ then working area location r, c is a spare location. Additional spares are needed on the edge of the working area (labeled with e in Figure 4 to ensure fringe PLBs are sufficiently close to a matching spare PLB in order to meet the defined distance criteria. Figure 4b demonstrates the same concepts for our second strategy. *It should be noted that with respect to performance these allocation strategies may not work for general FPGAs that don't use our STARs approach; however, with STARs, the system is already limited relative to the performance since the working area is often split by V-STAR and H-STAR.* The remainder of this section presents the three levels of fault tolerance in more details.

7.2 Current Implementation

We have successfully implemented system functions in the FPGA while reserving the H and V STARs for roving. With the system function running in the FPGA, we have used the partial reconfiguration capability of these FPGAs to download BISTER tiles for testing the logic resources in the STARs via the Boundary Scan interface. Once downloaded, we have executed the BIST sequence and retrieved the test results (both via the Boundary Scan interface) without interrupting operation of the system function. This process continues until we have rotated all of the BISTER tiles in the STARs to test all of the PLBs in all of their modes of operation. Running the Boundary Scan interface at about 66 KHz, the complete testing of the STAR (including all downloads of the BISTER tiles, execution of the BIST sequence, and retrieval of the test results) takes about

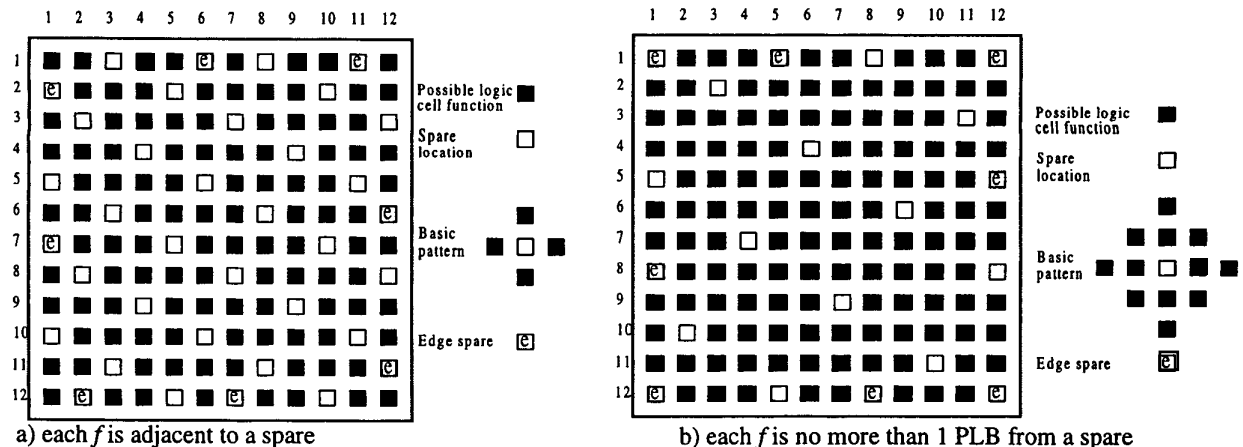


Figure 4. Spare allocation patterns

31 seconds. Once the STAR is completely tested, we momentarily stop the system clock in order to move the system function in the next two columns to the previous STAR area as well as the states of the memory elements in those two columns of PLBs containing the system function to free up the next STAR for testing before restarting the system clock. Currently, this process takes 3 to 4 seconds, but we are in the process of developing a faster roving process which should reduce this time to less than 1 second. In addition, this time that the system clock is stopped can be significantly reduced by increasing the clock frequency to the Boundary Scan interface to the maximum specified frequency of 10 MHz (instead of 66 KHz). This time can also be further reduced with FPGAs that facilitate partial reading of the configuration memory (to retrieve the states of memory elements) as well as the ability to write the states of memory elements during partial reconfiguration (neither of these features are supported in the ORCA 2CA series FPGA).

This process continues until the STAR has roved across the chip, which requires approximately 6 minutes for the ORCA 2C15A (where $N=20$) while running the Boundary Scan interface at 66 KHz. We have successfully roved the STAR across both good and faulty 2C15A devices and have detected faults in the defective devices as well as diagnose those faults that are within the PLBs. Via diagnosis of the faults we have been able to determine whether a given system function targeted for moving into the faulty PLB during the roving process can be used as a PUB or whether the system function must be moved to a spare PLB in order to avoid producing errors in the system operation. For example, in one case we identified the fault in a PLB to lie within a flip-flop such that system logic cell functions that only use combinational logic can use that PLB as a PUB while sequential logic functions (such as counters) must be moved to a spare PLB. The use of the PLB as a PUB as well as the shifting of system function to a spare PLB have both been accomplished without introducing errors in the system operation. Using the precompiled FTCRSs for each faulty PLB, fault tolerance is achieved with no additional interruptions to the system operation (other than the normal roving process) since the shift to the spare PLB is introduced during the roving process. The only time penalty is the additional execution time of the diagnostic configurations but, like during testing, the system function continues to operate without interruption so that only the fault latency is increased very slightly as a result of the diagnosis of a faulty PLB.

Not only does partial reconfiguration reduce the time required for downloading the BISTER tile configurations (and therefore the fault latency) but it also significantly reduces the amount of storage required for both the BISTER tile configurations as well as the FTCRSs. For example, the complete configuration memory for the 2C15A consists of about 225 Kbits. Partial configuration of the complete STAR requires about 23 Kbits. However, by performing

partial configuration for only those portions of the STAR that must be changed from one BISTER tile configuration to the next only requires between 3 Kbits and 10 Kbits, depending on how much of the STAR is being reconfigured with changing the mode of operation of the BUTs requiring the smallest number of bits while rotating the BISTER tile to begin testing a new set of BUTs requires the larger number of bits. There are 14 configurations (to change mode of operation in the BUTs) for each of the 8 rotations of the BISTER tile (where our BISTER tile is 4×2) so that only about 7% of the files are of the larger size while 93% of the files are the smaller size.

For our demonstration, we use a general purpose processor for the TREC. We have successfully implemented our FT approach using the TREC processor to control the levels of fault tolerance. The code used for the TREC is written in C++ and successfully compiled and executed both on a Pentium PC based platform running Windows 98 and Sun UltraSparc 10 workstation running Solaris. The TREC controls the FPGA configurations through the boundary scan interface attached to a serial port of the operating platform.

For level 1, the TREC parks the roving STAR over the fault(s) when a fault is detected. This fault is then diagnosed by the TREC and the location(s) of the fault(s) is(are) determined. In level 1, while the STAR is parked, the TREC currently diagnoses the fault. This information is used by the TREC to determine PUB usage and the configurations required for level 2.

The configurations used by level 2 of fault tolerance (level 2 is in effect when the STAR moves off of the faulty PLB) are determined using several pieces of information. First, if it is a single fault, the required sections of the system configurations affected by the fault are replaced by precompiled FTCRSs. If multiple faults are found, the TREC determines replacement FTCRS on-line. For example, if multiple FTCRSs are required and they are disjoint, we can use multiple precompiled FTCRSs for reconfiguring the system function. However, if multiple FTCRSs are required and they are not disjoint, then a new FTCRS must be generated while the STAR is parked that addresses the multiple faults. Additionally, the system logic cell functions affected by a detected and located fault are checked to see if they can make use of a PUB, before the system is reconfigured. For example, if the affected logic cell function is combinational and the fault is sequential, its configuration is left untouched. Additionally, while in level 2, the TREC downloads the correct configuration in the presence of faults.

Our design process for system functions can be accomplished manually or through use of synthesis using commercially available CAD tools. For our automated process, we start with an HDL description of the system function or application. We synthesize the design down to an EDIF netlist. Currently we use the Synopsys Design Compiler to synthesize designs and the Design Compiler automatically inserts predefined pin locations in the system's EDIF netlist file. Next we map the logic functions in

the EDIF file to PLBs using the ORCA Foundry Tools. At this point we have a nonplaced and nonrouted set of logic cell functions in a netlist file. We then insert preplaced STARs and spare PLBs into the netlist file. The spare PLBs are mapped to the virtual coordinate system. Next we place and route the system function using the ORCA commercial tools. Since the STARs are already placed, the ORCA tools place the system function in the virtual coordinate system. This provides good system performance given the preplaced spare PLBs. We use this placed and routed netlist file as input to the TREC to control downloading of FTCRS and to create all FTCRS.

The FTCRS are created using the placed and routed netlist file that includes empty locations for spares. In normal operation each element of F is mapped to its normal location in the virtual coordinate system. We precompile a set of FTCRS for each element of the set F . Each precompiled FTCRS is a placed and routed configuration replacement section that maps a single element of F to its spare location. Then when an element of F would fall on a faulty PLB in P that it can't use, the TREC replaces the normal route capability of the ORCA tools with the preplaced and routed FTCRS. For the case of multiple faults whose FTCRS are non-disjoint, the TREC uses the commercial tools to generate the FTCRS while the system function is on-line. Placement for the new FTCRS is limited to moving small numbers of logic cell functions. Routing for the new FTCRS is limited to only affected nets since we use the partial route capability of the ORCA tools. Since the changes are localized around the faults, the time for generating these FTCRSs is less than that of regenerating the system map of the entire system function. Also note, that while this generation process is taking place, the system function is still executing by using level 1 of our FT approach. This FTCRS generation capability also allows the TREC to create future FTCRS while in level 1 or level 2 of fault tolerance.

8. Conclusions

We have presented an on-line FT method suitable for use with the roving STARs fault detection/location method presented in [1]. Our method has been implemented and demonstrated in the ORCA 2CA series FPGA with several example system function circuits; however, the overall approach is suitable for other FPGAs. Our method currently handles faulty PLBs and is in the process of being expanded to handle faulty interconnects. Our approach is based on three levels of fault tolerance, and only moves system functions to spare locations when required since we introduce the concept and use of PUBs. As a result, we handle multiple faults without requiring the system function to stop execution. We minimize fault latency as well as the interruption of the system operation during the STAR roving process by using the partial reconfiguration capability of FPGAs.

9. References

- [1] M. Abramovici, C. Stroud, S. Wijesuriya, C. Hamilton, and V. Verma, "Using Roving STARs for On-Line Testing and Diagnosis of FPGAs in Fault-Tolerant Applications," *Proc. IEEE Int'l. Test Conf.*, pp. 973-982, 1999.
- [2] Xilinx, Inc., <http://www.xilinx.com>.
- [3] Altera Corp., <http://www.altera.com>.
- [4] W. Tsu, K. Macy, A. Joshi, R. Huang, J. Wawrzyniec, and A. DeHon, "High-Speed, Hierarchical Synchronous Reconfigurable Array," *Proc. ACM Int'l Symp. on FPGAs*, pp. 125-134, 1999.
- [5] C. Zeng, N. Saxena, and E. J. McCluskey, "Finite State Machine Synthesis with Concurrent Error Detection," *Proc. IEEE Int'l. Test Conf.*, pp. 672-679, 1999.
- [6] A. Steininger and Scherrer, "On the Necessity of On-Line BIST in Safety Critical Applications," *Proc. 29th Fault-Tolerant Computing Symp.*, pp. 208-215, 1999.
- [7] V. Kumar, A. Dahbura, F. Fischer, and P. Juola, "An Approach for the Yield Enhancement of Programmable Gate Arrays," *Proc. IEEE Int'l Conf. on CAD*, pp. 226-229, 1989.
- [8] F. Hatori, T. Sakurai, K. Sawada, M. Takahashi, M. Ichida, I. Yoshii, Y. Kawahara, T. Hibi, Y. Sacki, H. Muraga, and K. Kanzaki, "Introducing Redundancy in Field Programmable Gate Arrays," *Proc. IEEE Custom Integrated Circuits Conf.*, pp. 7.1.1-7.1.4, 1993.
- [9] S. Durand and C. Piguet, "FPGAs with Self-Repair Capabilities," *Proc. ACM Int'l Workshop on FPGAs*, pp. 1-6, 1994.
- [10] J. Narasimhan, K. Nakajima, C. Rim, and A. Daubura, "Yield Enhancement of Programmable ASIC Arrays by Reconfiguration of Circuit Placements," *IEEE Trans. on CAD*, Vol. 13, No. 8, pp. 976-986, August 1994.
- [11] J. Narasimhan, C. Rim, K. Nakajima, and A. Daubura, "Yield Enhancement of Wafer Scale Integrated Arrays," *Proc. IEEE Conf. on Wafer Scale Integration*, pp. 178-184, 1991.
- [12] N. Howard, A. Tyrrell, and N. P. Allison, "The Yield Enhancement of Field Programmable Gate Arrays," *IEEE Trans. on VLSI Systems*, Vol. 2, pp. 115-123, 1994.
- [13] J. Kelly and P. Ivey, "A Novel Approach to Defect Tolerant Design for SRAM Based FPGAs," *Proc. ACM Int'l Workshop on FPGAs*, pp. 7-11, 1994.
- [14] R. Cuddapah and M. Corba, *Reconfigurable Logic for Fault Tolerance*, Springer-Verlag, 1995.
- [15] F. Hanchek and S. Dutt, "Methodologies for Tolerating Logic and Interconnect Faults in FPGAs," *IEEE Trans. on Computers*, pp. 15-33, Jan. 1998.
- [16] S. Dutt and F. Hanchek, "REMOD: A New Methodology for Designing Fault-Tolerant Arithmetic Circuits," *IEEE Trans. on VLSI Systems*, Vol. 5, pp. 34-56, 1997.
- [17] N. Mahapatra and S. Dutt, "Efficient Network Flow Based Technique for Dynamic Fault Reconfiguration in FPGAs," *Proc. Fault Tolerant Computing Symp.*, pp. 122-129, 1999.
- [18] J. Emmert and D. Bhatia, "Reconfiguring FPGA Mapped Designs with Applications to Fault Tolerance and Reconfigurable Computing," *Lecture Notes on Comp. Sci.*, Vol. 1304, pp. 141-150, 1997.
- [19] J. Emmert and D. Bhatia, "Incremental Routing in FPGAs," *Proc. IEEE Int'l ASIC Conf.*, pp. 302-305, 1998.
- [20] J. Lach, W. Manione-Smith, and M. Potkonjak, "Efficiently Supporting Fault Tolerance in FPGAs," *Proc. ACM Int'l Symp. on FPGAs*, 1998.