Progress in Autonomous Fault Recovery of Field Programmable Gate Arrays

MATTHEW G. PARRIS, NASA Kennedy Space Center
CARTHIK A. SHARMA and RONALD F. DEMARA, University of Central Florida

The capabilities of current fault-handling techniques for Field Programmable Gate Arrays (FPGAs) develop a descriptive classification ranging from simple passive techniques to robust dynamic methods. Fault-handling methods not requiring modification of the FPGA device architecture or user intervention to recover from faults are examined and evaluated against overhead-based and sustainability-based performance metrics such as additional resource requirements, throughput reduction, fault capacity, and fault coverage. This classification alongside these performance metrics forms a standard for confident comparisons.

Categories and Subject Descriptors: B.8.1 [**Performance and Reliability**]: Reliability, Testing, and Fault Tolerance; B.7.0 [**Integrated Circuits**]: General; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search; A.1 [**General Literature**]: Introductory and Survey

General Terms: Design, Performance, Reliability

Additional Key Words and Phrases: FPGA, evolvable hardware, autonomous systems, self-test, reconfigurable architectures

ACM Reference Format:
Parris, M. G., Sharma, C. A., and Demara, R. F. 2011. Progress in autonomous fault recovery of field programmable gate arrays. ACM Comput. Surv. 43, 4, Article 31 (October 2011), 30 pages.
DOI = 10.1145/1978802.1978810 http://doi.acm.org/10.1145/1978802.1978810

1. INTRODUCTION

Field Programmable Gate Arrays (FPGAs) have found use among various applications in domains such as data processing, networks, automotive and other industrial fields. The reconfigurability of FPGAs decreases the time-to-market of applications that would otherwise require their functionality to be hard-wired by a manufacturer. Additionally, the ability to reconfigure their functionality in the field mitigates unforeseen design errors. Both of these characteristics make FPGAs an ideal target for spacecraft applications such as ground support equipment, reusable launch vehicles, sensor networks, planetary rovers, and deep space probes [Katz and Some 2003; Kizhner et al. 2007; Ratter 2004; Wells and Loo 2001]. In-flight devices encounter harsh environments of mechanical/acoustical stress during launch and high ionizing radiation and thermal stress while outside Earth's atmosphere. FPGAs must operate reliably for long mission durations with limited or no capabilities for diagnosis/replacement and little onboard capacity for spares. Mission sustainability realized by autonomous recovery of these

This research was supported in part by NASA Intelligent Systems NRA Contract NNA04CL07A.
Authors' addresses: M. G. Parris, NE-A3, Kennedy Space Center, FL 32899-0001; email: matthew.g.parris@nasa.gov; C. A. Sharma and R. F. Demara, School of Electrical Engineering and Computer Science, University of Central Florida, Box 162362, Orlando, FL 32816-2362.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2011 ACM 0360-0300/2011/10-ART31 $10.00
DOI 10.1145/1978802.1978810 http://doi.acm.org/10.1145/1978802.1978810

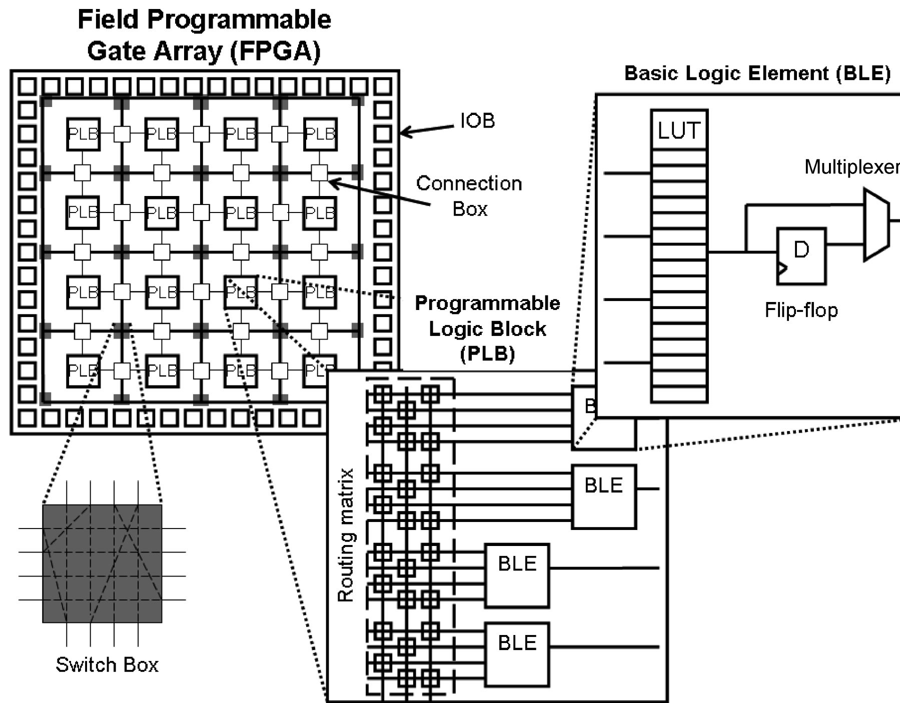

Fig. 1. Generic SRAM FPGA architecture.

reconfigurable devices is of particular interest to both in-flight applications and ground support equipment for space missions at NASA [Yui et al. 2003].

1.1. FPGA Architecture Overview

As indicated by its name, programmability is the primary benefit of FPGAs. Depending on the design of the device, a user programs anti-fuse cells or Static Random Access Memory (SRAM) cells within the FPGA. The anti-fuse cells store the application permanently, whereas the SRAM cells store the application temporarily, allowing reprogrammability. Since reprogrammability allows many more fault-handling techniques, this article focuses solely on SRAM FPGAs.

As shown in Figure 1, SRAM FPGA architectures are regular arrays of Programmable Logic Blocks (PLBs) among interconnect resources such as wire segments, connection boxes, and switch boxes [Trimberger 1993]. FPGA interconnect provides the means for multiple PLBs to realize complex logic functions. Connection boxes connect PLBs to wire segments, which in turn are connected to one another by switch boxes that allow various combinations of connections. The FPGA interconnect also joins PLBs to Input/Output Blocks (IOBs), which regulate the connections between the FPGA and external components.

The logic functionality of an application is realized by a combination of PLBs, each containing multiple Basic Logic Elements (BLEs). The BLE consists of: (1) a $2^n x1$ SRAM to store logic functions, where n is the number of inputs to the SRAM, (2) a flip-flop to store logic values, and (3) a multiplexer to select between the stored logic value or the SRAM. The most common SRAM size for logic functions is a 16×1 memory containing 4 inputs. In this configuration, the 16×1 SRAM behaves as a 4-input

function generator or a 4-input Look-Up Table (LUT), where the time to look up the result of the logic function is equal for all permutations of inputs.

1.2. Radiation-Induced Faults and Handling Techniques

When in the deep space environment, FPGAs are subject to cosmic rays and high-energy protons, which can cause malfunctions to occur in systems located on FPGAs. These radiation effects can be broadly classified into Total-Dose Effects and Single-Event Effects (SEEs). Total-Dose Effects describe cumulative long-term damage due to incident protons and electrons, and are described in detail by Dong-Mei et al. [2007]. SEEs are caused by the incidence of a single high-energy particle or photon. SEEs can be destructive, such as Single-Event Latchups (SELs), or nondestructive, as in the case of transient faults. Transient faults include Single-Event Upsets (SEUs) [Wirthlin et al. 2003], Multiple-Bit Upsets (MBUs), Single-Event Functional Interrupts (SEFIs), and Single-Event Transients (SETs). Adell and Allen [2008] provide a survey of technologies used for mitigating SEEs in FPGAs. Additionally, Bridgford et al. [2008] provide a glossary of terms and an overview of SEE mitigation technology for Xilinx FPGAs. This article discusses techniques to address the effects of nondestructive SEEs in SRAM-based FPGAs.

Radiation-hard describes resilience to either total-dose effects or SEEs at the device level. The configurations of anti-fuse FPGAs, for example, are radiation-hard since anti-fuse FPGAs do not depend upon SRAM cells to store their configurations. Radiation-tolerant, on the other hand, describes guaranteed performance up to a certain Total Ionizing Dose (TID) level or Linear Energy Transfer (LET) threshold. A TID of 300 krad (Si) and a Single-Event Upset (SEU) LET of 37 MeV-cm^2/mg are sufficient for the majority of space applications [Roosta 2004]. Consequently, FPGAs resistant to a TID of at least 300 krad (Si) have been labeled rad-hard [Actel 2005; Atmel 2007]. This label, however, can be misleading as memory cells and registers still remain vulnerable to SEUs and therefore must depend upon SEU mitigation techniques at the application level [Altera 2009; Baldacci et al. 2003; Bridgford et al. 2008]. Before the availability of radiation-tolerant SRAM FPGAs providing SEL immunity and performance characterization within heavy-ion environments [Xilinx 2008], designers of satellites and rovers had no serious alternative to the one-time programmable anti-fuse FPGA. If the inherent fault tolerance capability of anti-fuse FPGAs was insufficient, designers were restricted to employing passive fault-handling methods such as Triple Modular Redundancy (TMR) [Lyons and Vanderkulk 1962]. Due to the reconfigurable nature of SRAM FPGAs, radiation-tolerant SRAM FPGAs have enabled designers to consider other fault-handling methods such as the active fault-handling methods described by Sections 3 and 4 herein.

Fault avoidance strives to prevent malfunctions from occurring. This approach increases the probability that the system continues to function correctly throughout its operational life, thereby increasing the system's reliability. Implementing fault-avoidance tactics such as increasing radiation shielding can protect a system from single-event effects at the expense of additional weight. If those methods fail, however, fault-handling methodologies can respond to recover lost functionality. Whereas some fault-handling schemes maintain system operation while handling a fault, some fault-handling schemes require placing the system offline to recover from a fault, thereby decreasing the system's availability. This limited decrease in availability, however, can increase overall reliability.

1.3. Focus of this Survey Article

This survey focuses on fault-handling methods that modify an FPGA's configuration during runtime to address transient and permanent faults. Whereas some methods

Table I. Fault-Handling Characteristics and Considerations

	Metric	Description
Overhead	*Physical Resources*	additional amount of resources required due to fault-handling strategy
	Throughput Reduction	reduced rate of computations due to fault recovery
	Detection Latency	amount of time required detect and/or locate a single fault
	Recovery Time	amount of time system is offline to completely recover from a single fault
Sustainability	*Fault Exploitation*	ability to utilize defective resources
	Recovery Granularity	smallest component in which a fault may be handled
	Fault Capacity	number of fault-free resource units required for system functionality with a single additional fault
	Fault Coverage	handling of faults in various FPGA components
	Critical Components	external fault-handling components relied upon as fault free

incorporate fault detection and isolation techniques, these capabilities are not required for consideration by this survey. Since SRAM FPGAs can be: (1) radiation-tolerant, (2) reconfigured, and (3) partially reconfigured with the remaining portion remaining operational, research has also begun to focus on exploiting these capabilities for use in environments where human intervention is either undesirable or impossible. Section 2 classifies such fault-handling methods, which are described by Sections 3, 4, and 5. Table I lists various considerations addressed in detail by Section 5.

As listed in Table I, FPGA autonomous fault recovery strategies are described in this survey in terms of several fundamental processing *overhead* and mission *sustainability* characteristics. With respect to processing overhead, both the space complexity and time complexity of the existing recovery strategies can vary significantly. The principal space complexity metric are the additional *physical resources* which must either be reserved as spares or are otherwise utilized actively to support the underlying fault-handling mechanism. On the other hand, measures of time complexity incurred are the amount of *throughput reduction* as a side-effect of fault recovery, the *detection latency* measured as the time required to isolate the fault to the level of granularity which is covered by the fault-handling mechanism, and the *recovery time* which accounts for the cumulative unavailability of throughput during the recovery process. Meanwhile, the sustainability metrics will be used to assess the quality of the recovery which is achieved. Some FPGA fault-handling techniques attempt to increase long-term mission sustainability through *fault exploitation* strategies which effectively recycle the partially disabled resources as floating partial spares. Depending on the particular strategy used, the *granularity of recovery* can vary widely from a fixed number of columns or fixed-sized rectangular regions, down to individual logic elements without restriction. The *fault capacity* and *coverage* provided refer to measures of redundancy and logic/interconnect resource coverage, respectively. Some strategies provide explicit coverage for the latter type of resources while others provide only logic resource coverage, or implicit coverage for some interconnect resources. Finally, all strategies reviewed in this survey rely on one or more *critical components*, sometimes referred to as *golden elements* [Garvie and Thompson 2004], that are required to be operational in order for the recovery strategy to operate effectively. As discussed in subsequent sections, many strategies that tend to excel with respect to sustainability characteristics often do so at the expense of increased overhead characteristics.

2. CLASSIFICATION OF FAULT-HANDLING METHODS

Fault-handling methods can be broadly classified based on the *provider* of the method into *manufacturer-provided* methods and *user-provided* methods [Cheatham et al. 2006]. Furthermore, these methods can be classified based on whether the technique relies on active or passive fault-handling strategies. Of particular interest to this work

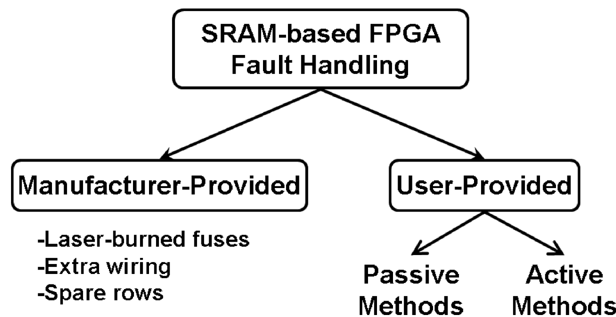

Fig. 2. Classification of FPGA fault-handling methods.

are the user-provided active fault-handling strategies. These can be classified based on the allocation, type, and level of redundant resources. In particular, a subset of the active fault-handling methods relies on a priori allocation of resources, both spare computational resources and spare designs. Lastly, the *dynamic* family of techniques is surveyed, and these techniques can be classified based on whether the technique requires the device to be taken offline for the recovery to be completed.

As suggested by Cheatham et al. [2006], Figure 2 divides fault-handling approaches into two categories based on the provider of the method. Manufacturer-provided fault recovery techniques [Cheatham et al. 2006; Doumar and Ito 2003] address faults at the level of the device, allowing manufacturers to increase the production yield of their FPGAs. These techniques typically require modifications to the current FPGA architectures that end-users cannot perform. Once the manufacturer modifies the architecture for the consumer, the device can tolerate faults from the manufacturing process or faults occurring during the life of the device.

User-provided methods, however, depend upon the end-user for implementation. These higher-level approaches use the configuration bitstream of the FPGA to integrate redundancy within a user's application. By viewing the FPGA as an array of abstract resources, these techniques may select certain resources for the implementation of desired functionality, such as resources exhibiting fault-free behavior. Whereas manufacturer-provided methods typically attempt to address all faults, user-provided techniques may consider the functionality of the circuit to discern between dormant faults and those manifested in the output. This higher-level approach can determine whether fault recovery should occur immediately or at a more convenient time.

The classification presented herein further separates user-provided fault-handling methods into two categories based on whether an FPGA's configuration will change at runtime [Parris 2008]. Passive methods embed processes into the user's application that mask faults from the system output. Techniques such as TMR are quick to respond and recover from faults due to the explicit redundancy inherent to the processes. Speed, however, does come at the cost of increased resource usage and power. Even when a system operates without any faults, the overhead for redundancy is continuously present. In addition to this constant overhead, these methods are not able to change the configuration of the FPGA. A fixed configuration limits the reliability of a system throughout its operational life. For example, a passive method may tolerate one fault and not return to its original redundancy level. This reduced reliability increases the chance of a second fault causing a system malfunction.

Active methods strive to increase reliability and sustainability by modifying the configuration of the FPGA to adapt to faults. As such, these methods cannot be realized on anti-fuse FPGAs. Reconfiguring the device allows a system to remove accumulated

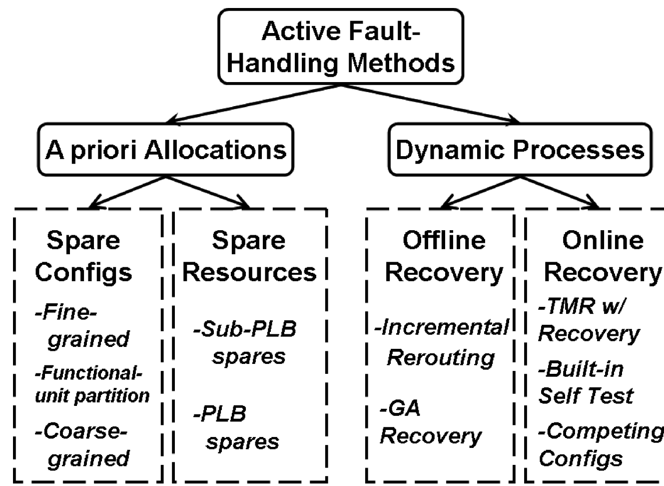

Fig. 3. Overview of active fault-handling methods.

SEUs and avoid the utilization of permanently faulty resources. External processors, which cost additional space, typically determine how to recover from the fault. These methods also require additional time either to reconfigure the FPGA or to generate the new configuration. Figure 3 illustrates two classes—a priori allocations and dynamic processes—described by Sections 3 and 4, respectively.

3. A PRIORI RESOURCE ALLOCATIONS

A priori allocations assign spare resources during design-time, independent of fault locations detected during runtime. These methods take advantage of the regularity of the FPGA's architecture by implementing redundancy structures. Since typical FPGA applications do not utilize 100% of the resources, the size of standby spares is reduced from entire FPGAs to unused resources within the FPGA. These techniques may recover from a fault by utilizing design-time compiled *spare configurations,* or remapping and rerouting techniques utilizing *spare resources*. Spare configuration-based methods must provide sufficient configurations whereas spare resource-based methods must allocate sufficient resources to facilitate a recovery without incurring unreasonable overheads. These two types of a priori allocations are addressed in Sections 3.1 and 3.2, respectively.

3.1. Spare Configuration Methods

Methods that utilize spare configurations require the user to generate alternative FPGA configurations during design-time to account for faults that may occur. Accounting for every possible fault at the lowest levels of the device, and therefore generating a configuration to account for each permutation of faults, is not practical. Instead, methods may consider a group of low-level resources as a logical partition of the FPGA. Then, FPGA configurations may be generated to account for faults occurring within each of these partitions. The methods outlined next select various granularities for partitioning the FPGA.

3.1.1. Fine-Grained Partitioning. Lach et al. [1998] implement a fine-grained partitioning technique where *tiles*—groups of logic and local interconnect resources—are formed. The goal of the tiling technique is to partition FPGA resources in such a way that at least one spare PLB is included within each tile to form Atomic Fault-Tolerant Blocks

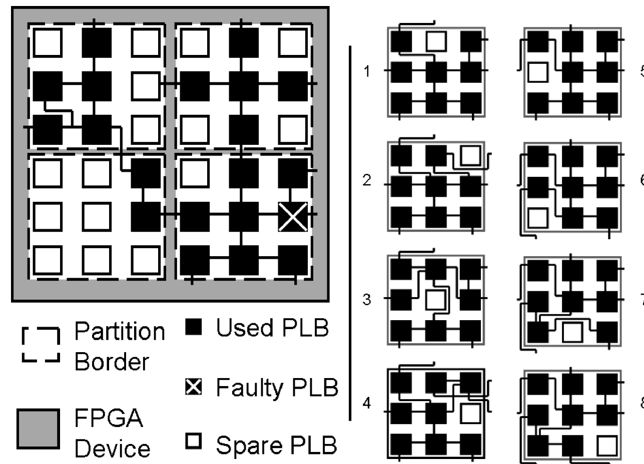

Fig. 4. Alternate configurations for a faulty 3 × 3 partition located in bottom right-hand corner region.

(AFTBs). Since each AFTB contains at least one spare PLB, each tile is able to tolerate at least one PLB fault.

Alternate fine-grained configurations generated during design-time and stored in external memory provide the ability to tolerate faults at runtime. In order to realize a significant reduction in storage space, each configuration is implemented as a partial configuration as opposed to a full configuration. The Xilinx Virtex-4 architecture, for example, allows two-dimensional partial configurations with a minimum height of 16 Configurable Logic Blocks (CLBs) [Lysaght et al. 2006].

During design-time, tiling implements multiple arrangements of logic resources within a tile as separate configurations such that each PLB within a tile is represented as a spare in at least one configuration. As seen in Figure 4, the bottom-right tile in the FPGA produces eight alternate configurations. To tolerate a fault during runtime, the system implements the configuration of the faulty tile that renders the faulty PLB as a spare, effectively bypassing the fault. Figure 4 depicts configuration 4 as one such alternate to bypass the fault which is located in the bottom right-hand corner region of the FPGA. Fixed inter-tile interfaces between alternate configurations render the arrangement of each tile logically independent.

Lach et al. [1998] report that this technique requires 2–10% additional logic resources when implemented on nine Microelectronics Center of North Carolina (MCNC) benchmark circuits. Additionally, this technique resulted in a throughput reduction between 14% and 45% of original performance. Given the granularity of resources used by this method, fine-grained configurations only consider interconnect resources that are local to the partition. In some cases, however, a nonlocal interconnect fault may be interpreted as a unique PLB fault and an appropriate configuration may bypass the interconnect fault.

3.1.2. Functional-Unit Partitioning. Since Triple Modular Redundancy (TMR) performs a majority vote of three modules, the voted output remains correct even if a single module is defective. Thus, TMR is a passive fault-handling technique widely used to mitigate permanent and transient faults. Whereas TMR can tolerate one faulty module, a fault occurring in a second module may produce a faulty functional output. Thus, TMR is limited in its fault capacity.

To increase system reliability, Zhang et al. [2006] combine TMR with Standby (TMRSB) to create a functional-unit spare configuration method. Shown by Figure 5,

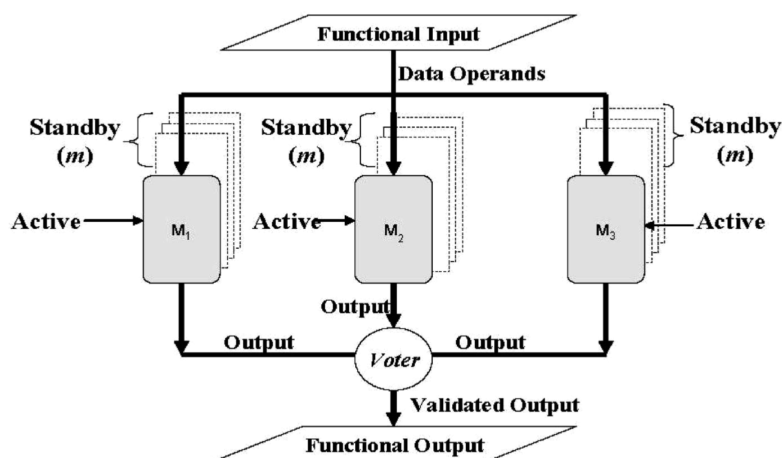

Fig. 5. Triple modular redundancy with standby configurations [Zhang et al. 2006].

each module of the TMR arrangement contains standby configurations that are available at runtime. At design-time, each of these configurations is created to utilize varying FPGA resources. Upon detecting a fault within one of the modules, a standby configuration not utilizing a faulty resource is selected and implemented to bypass the fault. TMRSB exploits the ability of TMR to remain online with two functional modules while the defective module undergoes a functional recovery. Such runtime recovery of modules increases the reliability of TMR by allowing another fault to occur in a second module while maintaining a correct functional output. The process repeats until all standby configurations are exhausted.

While this method does require 200% additional logic and interconnect resources, the detection latency is negligible since a fault occurring within one of the three modules is detected immediately. When generating alternating configurations, a user may control any throughput reduction to satisfy application requirements. At a minimum, this method can provide fault coverage for the triplicate logic and interconnect resources utilized by the method. The reliability is further increased when alternative configurations can bypass faulty logic and interconnect resources.

3.1.3. Coarse-Grained Partitioning. Mitra et al. [2004] present a coarse-grained fault-handling technique that reserves one or more columns of unused PLBs to tolerate faults. At design-time, multiple configurations are generated, each of which reserves spare columns in a distinct area of the FPGA. Once a fault occurs and is located, the system implements a configuration that covers the fault(s) with its spare columns. If the fault location is not available, then all configurations may be implemented and tested one at a time until a configuration provides a functional application.

Designers may partition the FPGA in one of two ways. If the application is small with respect to the FPGA device, then a *nonoverlapping* method can be considered. The nonoverlapping scheme separates the FPGA into columns, where one column contains the entire application. The remaining columns are not used by the application and are reserved as spares. As seen in Figure 6(a), this method generates three distinct configurations, each of which utilizes nonoverlapping FPGA resources. More generally, the number of generated configurations is $m + 1$, where the number of tolerable faulty columns equals m.

For larger applications, Figure 6(b) displays a configuration that separates the FPGA application into columns while reserving at least one column as spare. Alternate

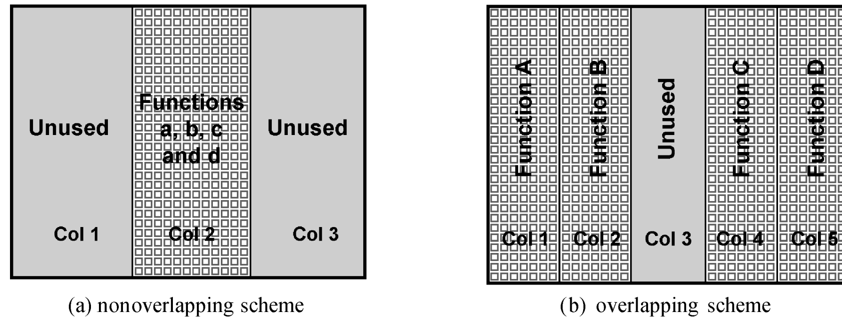

Fig. 6. Coarse-grained partitioning schemes for an FPGA.

configurations are generated during design-time so that within each configuration, a different column becomes the spare column. In the case of one spare column and four columns containing the application, five distinct configurations are generated. More generally, the number of generated configurations is $\frac{(k+m)!}{k!m!}$, where k is the number of columns containing the application. This scheme is *overlapping* since the various generated configurations utilize overlapping FPGA resources. Unlike the nonoverlapping scheme, some configurations, such as Figure 6(b), may require horizontal routing resources within the spare column to connect the separated logic resources.

Since the technique is coarse-grained, its fault coverage in both logic and interconnection resources is implicit. Whereas logic resources are unused in a spare column in the overlapping scheme, interconnection resources are utilized in the spare column to link the two disjointed functional areas. As such, multiple alternate configurations of the same spare column must be generated to enable the technique to bypass faulty interconnect resources located within the spare column. Depending on the size and type of application, a user can choose the number of logic and interconnect resources to reserve as spares, in addition to selecting between nonoverlapping and overlapping methods. When implementing the coarse-grained method on four MCNC benchmark circuits, the critical path was extended, resulting in a throughput reduction between 11% and 18% of the original circuit.

3.2. Spare Resource Methods

Spare resource methods strategically implement spare resources into the design of the application to tolerate faults. Spare configurations only account for a small subset of all possible means of utilizing spares, which is determined during the design phase prior to any faults occurring. Spare resource methods, however, delay this determination until after a fault is located at runtime. Whereas this strategy may enable an FPGA to handle more fault patterns, determining how to best utilize the spare resources at runtime is an overhead. To mitigate this overhead, spare resource methods incorporate redundancy structures to minimize the time required to generate a useful configuration.

3.2.1. Sub-PLB Spares. Typical FPGA architectures implement logic functions with Look-Up Tables (LUTs). As shown in Figure 7, Basic Logic Elements (BLEs) combine each LUT with a flip-flop and output multiplexer to enable sequential logic implementation. PLBs, in turn, contain multiple BLEs as in the Virtex-4 architecture, which contains eight BLEs per PLB.

By implementing ten benchmark circuits from the MCNC suite [Yang 1991], the RAW benchmark suite [Babb et al. 1997] and a benchmark circuit generator [Hutton et al. 1997], Lakamraju and Tessier [2000] found that, on average, 40% of the utilized

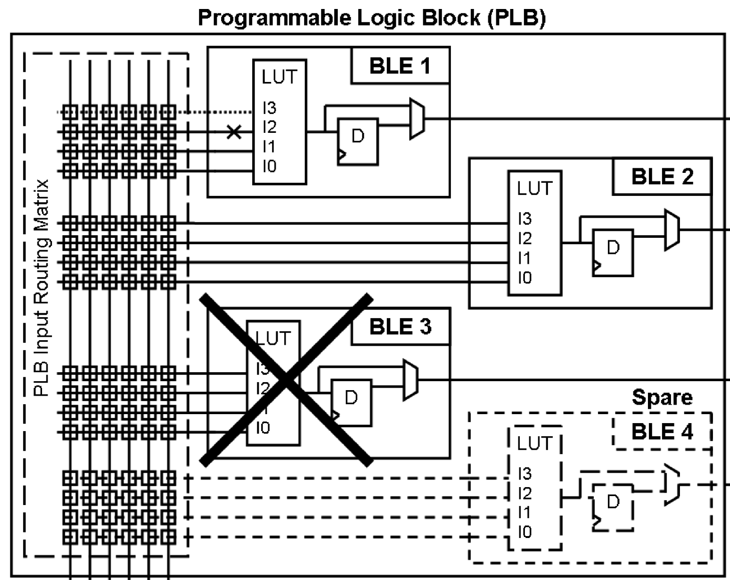

Fig. 7. PLB recovery strategies using sub-PLB spares.

4-input LUTs contained one or more spare input. This suggests that an FPGA application contains inherent spares at a finer granularity than the PLB level as previously discussed. This PLB recovery strategy reserves spare BLEs and implements a hierarchy of fault-handling strategies to take advantage of these spare resources, beginning with the finest granularity: LUT input swap, BLE swap, PLB I/O swap, incremental reroute, and complete reroute.

Given the identification of a faulty LUT input by a fault detection technique, the sub-PLB fault-handling method first attempts to swap the faulty resource with a spare input of the same LUT. Figure 7 shows input I2 of BLE1 as a faulty LUT input that may be swapped with a spare LUT input such as input I3 to avoid the fault. After swapping the LUT inputs, the contents of the LUT are modified to compensate for the input change. Whereas Figure 7 depicts a full PLB input routing matrix, some FPGA architectures contain only a partial routing matrix, restricting the number of PLB inputs to which a given LUT input may connect. For these architectures, the LUT input swapping method must consider whether the spare LUT input has access to the same PLB inputs as the faulty LUT input, to prevent rerouting. If spare LUT inputs with similar connections are available, this method is ideal as it does not require logical or connection changes outside of the BLE. If a spare LUT input is not available, then the entire BLE is considered faulty.

When a BLE is considered faulty, as is the case with BLE 3 in Figure 7, it is swapped with the reserved spare shown as BLE 4. In the case of partial routing matrices, the BLE swapping method needs to ensure the spare BLE has access to the same PLB inputs as the faulty BLE to prevent rerouting. Figure 7 shows that BLE 3 can swap with BLE 4 because of the similarity in connectivity, thus the change only affects the PLB and not the remainder of the circuit. If a spare BLE is not available, then the entire PLB is considered faulty and *incremental rerouting* is required. Incremental rerouting is discussed further in Section 4.1.1. Similar to the LUT input swap, faulty PLB input/output wires may be swapped with spare wires that contain similar connections. If a spare PLB input/output wire is not available, then incremental rerouting is required.

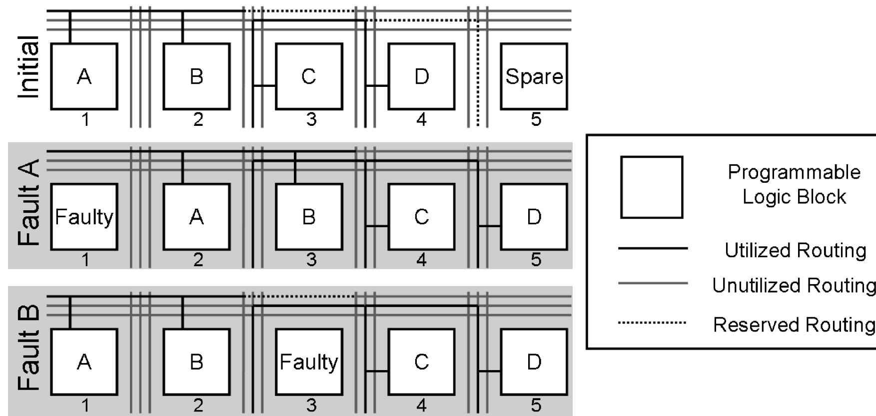

Fig. 8. Fault scenarios with spare PLBs [Hanchek and Dutt 1998].

Lakamraju and Tessier [2000] report that 24% of the fault scenarios requiring incremental reroute could be handled using the BLE swap by reserving at least one BLE as spare per PLB. Whereas BLE swap is quicker than incremental rerouting, reserving BLEs requires an additional 8% of logic resources when using 8 BLEs per PLB and 21% of logic resources using 4 BLEs per PLB. In addition to requiring a detection method that can isolate a fault to a BLE, sub-PLB spares require a custom place-and-route application to perform the BLE swap to bypass faults.

3.2.2. PLB Spares. To tolerate logic faults within a PLB, Hanchek and Dutt [1998] allocate the rightmost PLB of each row as spare. In the case of a fault, a string of PLBs beginning with the faulty PLB is shifted one PLB to the right. More formally, this technique is *node-covering*, which allocates a cover PLB to each PLB. In the case of a fault occurring in a PLB, its cover replaces the functionality of the faulty PLB to avoid the fault. This covering continues within a row in a cascading fashion until the spare PLB at the end of the row is reached. For a PLB to become a cover, it must duplicate the: (1) logic functionality and (2) connectivity of the original PLB to other PLBs. Hanchek and Dutt [1998] ensure that cover cells duplicate connectivity by incorporating reserved wire segments during the design process. Whereas end-users may choose to implement the node-covering strategy for tolerating logic faults, the authors specify that handling interconnect faults requires modification to the FPGA architecture, and thus is intended for manufacturer yield enhancement. Such modifications are outside the scope of this survey.

In Figure 8, some wire segments are utilized by the initial configuration whereas other wire segments are reserved, such as the one above location 3. As is the case with Fault Scenario A, this reserved segment becomes utilized by PLB B by shifting into location 3. Likewise, the two segments above and to the right of location 4 become utilized by the PLBs to the left. Additionally, a design may contain inherent reserved segments where some utilized wire segments of the initial configuration also serve as reserved wire segments in a fault scenario. This is seen in Fault Scenario A where PLB B allows its utilized wire segment above location 2 to be used by PLB A. During design-time, a custom tool determines the necessary reserved routing segments to enable the FPGA to tolerate one faulty PLB per row. Two heuristics that increase the efficiency of routing include segment reuse and preferred routing direction. Segment reuse allows a utilized net and a reserved net to map to the same wire segment if the utilized net will move off of the wire segment with the shifting the PLBs, therefore

freeing up a wire segment for the reserved net. For nets that cross the FPGA, preferred routing direction encourages the router to extend such nets to the right, horizontally, as far as possible before extending the net in either vertical direction. Providing longer continuous horizontal segments allows greater opportunities for a design to contain inherent reserved segments, as discussed earlier.

Since the design process has ensured that the cover cells can duplicate functionality and connectivity, the routing phase of the place-and-route process is finalized during design-time. To avoid a faulty PLB within a row, an end-user only needs to replace the PLBs by shifting a row of PLBs into a fault-free configuration, which requires a custom placer application. The time to modify an existing configuration by replacing a row of PLBs is, however, significantly less than the time required either to generate a new configuration from scratch or to incrementally reroute an existing configuration. Depending on the size of the target FPGA, logic resource overhead can range from 1–41% of the FPGA, as explained by Section 5.1.1. Based on 18 benchmark circuits provided by MCNC and other sources, Hanchek and Dutt [1998] report that an additional 9% to 50% of resource utilization is required to reserve interconnect resources. Implementing the fault-handling method for a single fault can extend the length of the critical path, causing a throughput reduction between 0% and 14% of the original application.

4. DYNAMIC RECOVERY PROCESSES

Methods using dynamic processes aim to allocate spare resources or otherwise modify the configuration during runtime, after detecting a fault. Whereas these approaches offer the flexibility of adapting to specific fault scenarios, additional time is necessary to generate appropriate configurations to recover from the specific faults. *Offline* recovery methods require the FPGA's removal from operational status to complete the refurbishment. *Online* recovery methods endeavor to maintain some degree of data throughput during the fault recovery operation, increasing the system's availability. Sections 4.1 and 4.2, respectively, address these two types of active runtime dynamic methods.

4.1. Offline Recovery Methods

4.1.1. Incremental Rerouting Algorithms. The node-covering technique discussed in Section 3.2.2 avoids a fault by replacing a circuit into design-time allocated spares using design-time reserved wire segments. Dutt et al. [1999] expand this method by dynamically allocating reserved wire segments during runtime instead of design-time. Runtime reserved wire segments allow the method to utilize unused resources in addition to the spares allocated during design-time.

Emmert and Bhatia [2000] present a similar incremental rerouting approach that does not require design-time allocated spare resources. The fault recovery method assumes an FPGA to contain PLBs not utilized by the application, thus exploiting unused fault-free resources to replace faulty resources. When a logic or interconnection fault is detected by some external method, incremental rerouting calculates the new logic netlist to avoid the faulty resource. The method reads the configuration memory to determine the current netlist and implements the incremental changes through partial reconfiguration.

A string of PLBs is created, starting with the faulty PLB and ending with the PLB adjacent to the spare resource. Figure 9 shows one such string, starting with PLB 25, including PLB 20, and ending with PLB 15. To avoid the fault, the string of PLBs shifts away from the faulty resource and towards the spare resource. In the case of node-covering, every row has a spare resource so the string of PLBs within the row

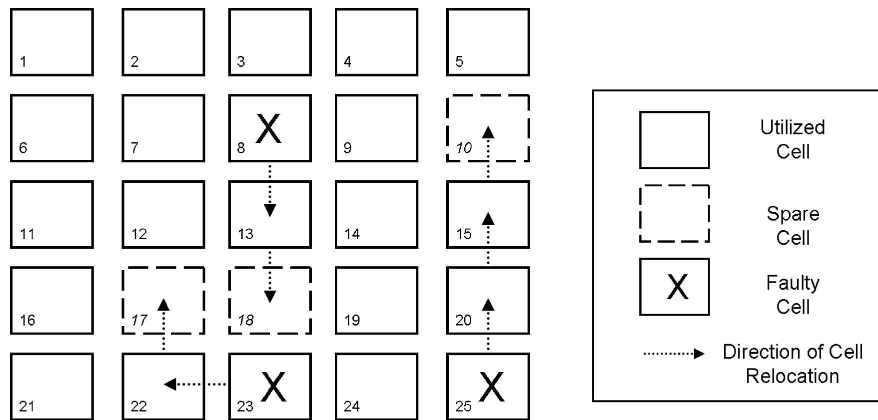

Fig. 9. One possible minimax fault-handling strategy for a 5 × 5 array.

simply shifts to the right, leaving the faulty resource unused. Since this method does not allocate a spare resource for every row, the string of PLBs may extend into multiple rows to reach a spare PLB, as shown in Figure 9.

This approach uses minimax Grid Matching (MGM) to determine the optimum replacement of faulty PLBs. Minimax is an algorithm that minimizes the maximum Manhattan distance, L, between the faulty PLB and an unused, fault-free PLB. Beginning with $L = 1$, Figure 9 shows that the faulty cell 23 is adjacent to the spare cell 18 and thus a match, but faulty cells 8 and 25 do not have adjacent spares and thus no matches. Incrementing L to two, faulty cell 23 matches cell 17 while maintaining its match to cell 18. Additionally, faulty cell 8 matches cell 18 and cell 10, whereas faulty cell 25 still has no matching spare. Incrementing L to three, faulty cell 23 acquires no new matches, faulty cell 8 acquires cell 17 as a match, and faulty cell 25 matches cell 10 and cell 18. Since all cells have a match at Manhattan distance $L = 3$, one match is then chosen for each faulty cell. Figure 9 depicts one such possibility for the three faulty PLBs, where, for example, the logic in cell 23 shifts to cell 22 and the logic in cell 22 shifts to the spare cell 17.

Replacing PLBs requires the wire segments of the moving PLBs to be rerouted. The configuration memory of the FPGA is read to determine which nets are affected by the replaced PLBs. All faulty nets and those that solely connect the moved PLBs are *ripped-up* [Emmert and Bhatia 2000] while those that connect other unmoved PLBs remain unchanged. A greedy algorithm then incrementally reroutes each of the dual-terminal nets to reestablish the application's original functionality. Initially, the algorithm only uses spare interconnection resources within the direct routing path, but may enlarge its scope to encompass wider routing paths for unroutable nets. Lakamraju and Tessier [2000] expand this work by utilizing historical node-cost information from previous routing attempts to increase the probability of routing success.

Whereas this method does not require reserved spare resources, strategically placing spares throughout the design will improve the probability of successfully handling a fault. By incrementally rerouting various circuits, Emmert and Bhatia [2000] observe a throughput reduction between 2% and 35% of the original performance prior to a fault. Utilizing the test circuits, up to 32 PLBs on average could be replaced before the algorithm encounters an unroutable net. While observed as a diminishing return, enlarging the scope of possible routing paths can increase the number of PLBs that may be moved prior to encountering an unroutable net by as much as 25%.

4.1.2. Genetic Algorithm Recovery. Genetic Algorithms (GAs) are inspired by evolutionary behavior of biological systems to produce solutions to computational problems [Mitchell 1996]. Suitable for complex search spaces, GAs have proven valuable in a wide range of multimodal or discontinuous optimization problems. Previous research has investigated the capability of GAs to design digital circuits [Miller et al. 1997] and recover from runtime faults [Keymeulen et al. 2000]. Vigander [2001] proposes the use of GAs to recover faulty FPGA circuits. As a proof of concept, Vigander [2001] implements extrinsic evolution, utilizing a simulated feed-forward model of the FPGA device. Lohn et al. [2003], however, propose intrinsic evolution where the FPGA hardware is included in the evolutionary loop. In the experiments, the logic and interconnect configurations are represented as genetic chromosomes.

The evolution process begins with initializing a population of candidate solutions. These initial solutions contain different physical implementations of the same functional circuit. The optimal size of the population is determined through experiments to ensure that there is sufficient diversity in the representations. In order for evolutionary recovery to be effective, the chromosomes should provide adequate resource coverage. However, the overhead involved in storing the configurations and the initial effort required by some methods to create the alternate configurations are some of the limiting factors determining the population size. As examples of population sizes, the FPTA experiment [Keymeulen et al. 2000] utilizes a population size between 128 and 200 for the experiments. For evolutionary design, as a result of experimenting with varying population sizes, Miller et al. [1997] note that evolving small populations over very large number of generations gives the best results. In the experiment, they vary the population size from 10 to 240 and note that a population size between 15 and 60 yields the best performance. Vigander [2001] utilizes a population size of 50 in the evolutionary repair experiments, and Oreifej et al. [2006] utilize a population size of 25 designs. DeMara and Zhang [2005] utilize a population of 20 fault-free configurations for the consensus-based evaluation technique. In general, the optimal population size to be used in evolutionary design and repair experiments has to be determined via experimentation, subject to a storage and performance criterion.

Once a fault occurs, the GA evaluates the candidate solutions from the initial population against a set of test vectors to determine the functionality of each. Based on performance, a fitness function assigns values to each candidate solution, revealing which are most affected by the fault. If none of the available configurations provides the desired functionality, then genetic operators create a new population of diverse candidate solutions from the previous configurations. Configurations having a higher fitness value are more likely to be selected and combine with other configurations by application of the *crossover* genetic operator. Additionally, the *mutation* genetic operator injects random variations in the newly created candidate solutions. Vigander [2001] also makes use of a *cell swap* operator that allows the functionality and connectivity of a faulty cell to be interchanged with that of a spare cell. The GA evaluates the newly created solutions and replaces poorer performers from the old population with better performers in the current population to create a new generation of candidate solutions. This evolutionary process repeats, terminating either when an optimal solution is discovered or after a specific number of generations.

Vigander [2001] reported difficulty in recovering full functionality of a faulty 4-bit multiplier. Experiments included increasing the number of resources available to the GA and increasing the number of generations. During extrinsic experiments, Lohn et al. [2003] were successful in recovering full functionality of a faulty quadrature decoder within 623 generations. Moreover, the newly generated configuration exploited the induced fault, where removing the stuck-at fault caused the circuit to lose its full

functionality. Lohn et al. [2003] also report that initial intrinsic experiments reduced the evolutionary process from hours to minutes.

4.1.3. Augmented Genetic Algorithm Recovery. To decrease the amount of time required to recover from a fault, Oreifej et al. [2006] augment the genetic algorithm fault-handling concept with a Combinatorial Group Testing (CGT) fault isolation technique [Sharma 2008]. A group testing algorithm identifies subsets of resources, and schedules a minimal number of tests with the goal of identifying the faulty resource. The algorithm is adaptive, so the suspect resources are rearranged based on the results of preceding tests.

The functional-equivalent, yet physically distinct configurations form the population upon which the GA operates. CGT evaluates each configuration for correct functionality. If a configuration manifests a faulty output, then the resources used by that configuration are considered suspect. Various overlapping subsets of resources are used by the configurations. CGT tests multiple configurations and accumulates the number of times each resource is considered suspect through a history matrix. Under CGT-based fault isolation, let $\mathbf{R}$ denote the set of all resources $r_i(x,y) \in \mathbf{R}$ under test as specified by their (x,y) coordinates. A set of functionally-equivalent logic configurations, $\mathbf{C}$, consists of subsets $\mathbf{c}_i$, $0 \leq i \leq p$, where p quantifies the size of a *population* of *design configurations.* Each configuration realizes the combinatorial logic required for the application. The *discrepancy function* $D(\mathbf{T'}, \mathbf{c}_j)$ yields a set of all outputs that are not equal to the correct output, as realized when tests comprising the syndrome, $\mathbf{T'}$ are applied to configuration $\mathbf{c}_j$. Tests $\mathbf{T'} \subset \mathbf{T}$ on a subset $\mathbf{c}_j$ are *positive* if and only if $D(\mathbf{T'},\mathbf{c}_j) \neq \{\}$, and *negative* otherwise. The history matrix, H, keeps track of the discrepancy counts of the resources. All elements in the H matrix are initialized to zero. As a stage of tests proceeds, for each test t_i for which $D(t_i, c_j) \neq \{\}$, all H matrix entries $H(x,y)$ are incremented by one where (x,y) are the coordinates of all $r_i(x,y) \in c_j$. Over time, the maximal elements in H identify suspect resources by their coordinates. Under a single-fault assumption, fault isolation is complete when a unique maximum can be identified in H. The GA, in turn, uses the fault location information to avoid faulty resources while evolving a configuration for fault recovery. Oreifej et al. [2006] report that the number of generations needed to isolate a single fault is 0.11% of the total needed to realize a functionally correct circuit. This additional temporal overhead is justified, as utilizing fault location information can reduce the number of generations needed to recover full functionality by up to 38%.

4.2. Online Recovery Methods

4.2.1. TMR with Single-Module Recovery. In Section 3.1.2, faults in TMR arrangements were handled with a priori, design-time configurations. When these a priori configurations cannot restore required functionality, genetic algorithms may utilize them as an initial population within the evolutionary process to generate a suitable alternative [Garvie and Thompson 2004; Ross and Hall 2006; Shanthi et al. 2002; Vigander 2001]. As shown by Figure 10, genetic operators and reconfiguration are invoked when a defective module is detected. At design time, Ross and Hall [2006] produce a population of diverse configurations for implementation. At runtime, three of these configurations are implemented and monitored for discrepancies. Agreeing outputs indicate that the modules are functioning correctly whereas discrepancies indicate that defective resources are utilized by at least one of the configurations. A simple mutation genetic operator is applied to defective modules and the performance or fitness of the new individual is evaluated. The process repeats until the fault is occluded.

Similar to the CGT fault isolation technique, Shanthi et al. [2002] utilize a deterministic approach in identifying faulty resources. By monitoring the resources within

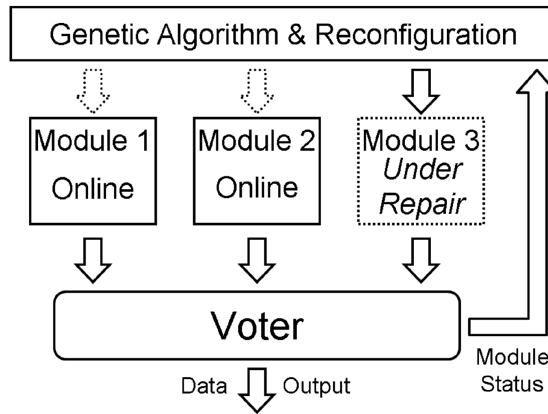

Fig. 10. Single-module recovery in TMR arrangement.

each configuration, resources utilized by viable modules gain confidence whereas resources utilized by faulty modules gain suspicion. This information allows for fault handling by excluding defective resources from configurations. Additionally, differing configurations can be rotated to reveal dormant faults in unused resources.

Scrubbing is a fault-handling technique commonly used to replace affected FPGA configuration memory cells with faultless configuration data. Scrubbing depends upon reading back the configuration memory cells and detecting faults by comparing them to the original configuration. Upon isolating a fault, the FPGA can recover the correct bitstream through reconfiguration. The Mars Exploration Rovers successfully implemented this method to mitigate SEUs while in transit to Mars [Ratter 2004]. An advantage of coupling scrubbing with partial reconfiguration is that the reconfiguration process can occur without interrupting the normal operation of other parts of the system [Carmichael et al. 2000; Yui et al. 2003].

Instead of selecting from a diverse population, Garvie and Thompson [2004] implement three identical modules. The commonality between configurations permits a *lazy scrubbing* technique to address transient faults. Lazy scrubbing considers the majority vote of the three configurations as the original configuration when scrubbing a faulty module. Of course, lazy scrubbing only applies when a genetic algorithm has not modified the original configurations to tolerate a permanent fault. To address permanent faults, a $(1 + 1)$ evolutionary strategy [Schwefel and Rudolph 1995] provides a minimal genetic algorithm, which produces one genetically modified offspring from a parent and chooses the configuration with the better fitness. To mitigate the possibility of a misevaluated offspring replacing a superior parent, a history window of past mutations is retained to enable rollback to the superior individual. Normal FPGA operational inputs provide the test vectors to evaluate the fitness of newly formed individuals. To determine correct values, an individual's output is compared to the output of the voter. An individual's fitness evaluation is complete when it has received all possible input combinations.

By implementing a full-adder circuit in a TMR arrangement, Ross and Hall [2006] successfully handle a fault with an average of 760 cycles. Garvie and Thompson [2004] utilize the cm42a combinational circuit of the MCNC suite [Yang 1991] and achieve restoration of full functionality within 800,000 generations, an equivalent of two minutes of simulated evolution time. Whereas the a priori, design-time configurations utilized for the initial population can be created with specific performance

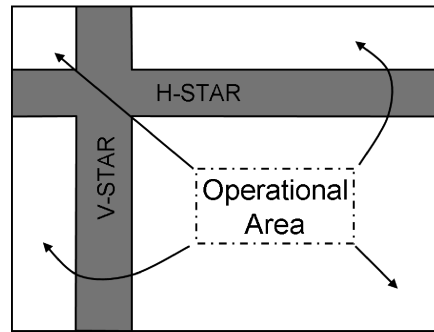

Fig. 11. Roving STARs within an FPGA.

characteristics, the throughput delay of circuits generated by GAs is indeterminate. Future genetic algorithms may consider including throughput delay as part of a circuit's evaluation to satisfy performance requirements.

4.2.2. Online Built-in Self-Test. Emmert et al. [2007] present an approach that pseudo-exhaustively tests, diagnoses, and reconfigures resources of the FPGA to restore lost functionality due to permanent faults. The application logic handles transient faults through a concurrent error detection technique and by periodically saving and restoring the system's state through checkpointing. As shown in Figure 11, this method partitions the FPGA into a noncontiguous operational area and a Self-Testing ARea (STAR), consisting of a horizontal STAR and a vertical STAR. Such an organization allows normal functionality to occur within the operational area while Built-In Self-Tests (BISTs) and fault diagnosis occur within the STARs. Whereas other BIST methods may utilize external testing resources assumed fault-free, the resources-under-test also implement the Test-Pattern Generator (TPG) and the Output Response Analyzer (ORA).

To provide fault coverage of the entire FPGA, the STARs incrementally rove across the FPGA, each time exchanging tested resources for the adjacent, untested resources in the operational area. The H-STAR roves top to bottom then bottom to top while the V-STAR roves left to right then right to left. Whereas one STAR could test and diagnose PLBs, two STARs are required to test and diagnose programmable interconnect: the H-STAR for horizontal routing resources and the V-STAR for vertical routing resources. Where they intersect, the two STARs may concurrently test both horizontal and vertical routing resources and the connections between them. Since faults can occur in either used or unused resources with equal probability, roving STARs provide testing for all resources. Uncovering dormant faults in unused resources prevents them from being allocated as spares to replace faulty operational resources.

In addition to facilitating testing, diagnosis, and reconfigurations, a Test and Reconfiguration Controller (TREC) is responsible for roving the STARs across the FPGA. The TREC is implemented as an embedded or external microprocessor that communicates to the FPGA through the boundary-scan interface. All possible configurations of the STARs are processed during design-time and stored by the TREC for partial reconfiguration during runtime. Relocating the STARs through partial reconfiguration only affects the logic and routing resources within the STAR's current and new locations. When a STAR's next location includes sequential logic, the TREC pauses the system clock until the logic is completely relocated. On an ORCA 2C15 FPGA—a 20×20 PLB array—the authors report that the clock must be stopped for approximately 250 μ s. Instead of stopping the system clock, Gericota et al. [2008] provide an

active replication method which replicates logic resources without halting the system clock, thereby increasing the availability of the FPGA. In the first of two phases of active replication, the internal configuration of the PLB is copied to a spare PLB and the inputs of both PLBs are placed in parallel. If the PLB makes use of a clock-enable, then additional logic is temporarily employed so the replication process need not wait for the clock-enable to be active. When the clock-enable is not active, the additional logic copies the internal state of synchronous logic from the replicated PLB to the spare. If the clock-enable becomes active during the replication process, the additional logic instead allows the spare PLB to update its own synchronous components. Since the inputs are connected in parallel, the output of the replicated PLB is equal to that of the spare PLB. The second phase is then implemented where the outputs of the replicated PLB and the spare PLB are placed in parallel, and after one clock pulse, both the outputs and inputs are disconnected from the replicated PLB. Without halting system operations, the replicated PLB then becomes a STAR for testing and allows system operations to continue.

Roving STARs support a three-level strategy to handling permanent faults. In the first level, a STAR detects a fault and remains in the same position to cover the fault. Since a STAR contains only offline logic and routing resources, testing and diagnosing time is not at a premium; the application continues to operate normally while the TREC tests and diagnoses the fault. After diagnosing the fault, the TREC determines if the fault will affect the functionality that will soon occupy the faulty resources upon moving the STAR. If the fault will not affect the new configuration's functionality—such as only affecting resources that will be unused or spare—then the application's output will not articulate the fault and no action is required. If the fault will affect the new configuration's functionality, then the TREC generates a Fault-Bypassing Roving Configuration (FABRIC) to incrementally reroute the new configuration so that the fault will not affect its functionality. Whereas some FABRICs may be compiled during design-time, most fault scenarios will dictate compiling them online while the STAR covers the fault. While one STAR covers a fault for testing and diagnosis, the second STAR may continue roving the FPGA searching for faults. The second-level strategy then applies the FABRIC that either was compiled during design-time or was generated during the first-level strategy. Replacing a faulty resource with a spare one through a FABRIC thus releases the STAR covering the fault to continue roving the FPGA.

If the fault affects functionality and no spare resources are available to bypass the fault, then the third strategy is invoked. As a last resort, the TREC has an option to perform *STAR stealing*, which reallocates resources from a STAR to the operational area to bypass the fault. Removing resources from a STAR immobilizes it from roving the FPGA. Whereas the second STAR can test all PLBs in an FPGA with an immobile STAR, only half of the routing resources can be tested. In some situations, however, a mobile STAR may intersect and forfeit its resources to an immobile STAR, which releases the other STAR to rove the FPGA and test the remaining routing resources.

Testing and diagnosis occurs within a STAR by utilizing the resources of the STAR through partial reconfiguration. The TREC configures a TPG, an ORA, and either two Blocks Under Test (BUT) for a PLB test or two Wires Under Test (WUT) for an interconnect test. Since no resource may be assumed to be fault-free, the TPG, BUTs/WUTs, and ORA are rotated through common resources of the STAR. The TREC maintains the results for all test configurations so that the common faulty resources can be identified between the two parallel BUTs or WUTs and the rotation of resources.

Whereas this survey focuses on fault-handling methods rather than detection techniques, another proposal regarding the testing phase of the online BIST method merits mention. While using the same concepts of the roving STAR, Dutt et al. [2008] provide more accurate BIST strategies that increase the percentage of faults correctly

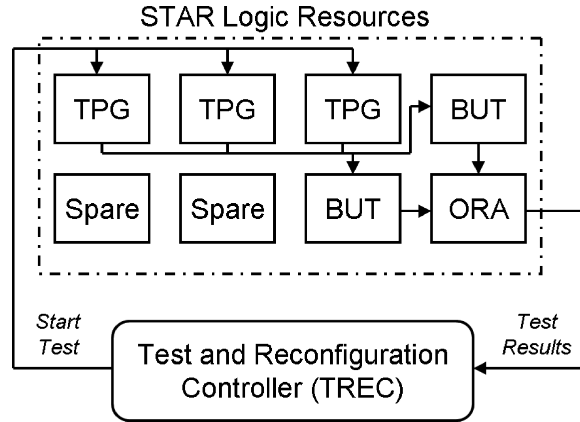

Fig. 12. 4×2 programmable logic block BIST.

diagnosed. Additionally, their method can decrease detection latency by switching from pseudo-exhaustive BISTs to functional-based BISTs (Figure 12).

By inserting a STAR in the working area, Emmert et al. [2007] observe a worst-case throughput reduction of 16%. The authors additionally report a detection latency of 1.34s using the maximum boundary scan frequency of 10 MHz on the ORCA OR2C15A FPGA, which is a 20×20 PLB array with four LUTs per PLB. To scale the fault detection roving time to larger FPGA architectures, this survey calculates the number of Effective Cycles per PLB (ECPP), which is defined to include the average PLB testing time plus the average STAR reconfiguration and moving time. Eq. (1) expresses the roving time as a function of: (1) N/2 positions required for a full sweep of a STAR [Abramovici et al. 2004], (2) the number of PLBs in a row/column, (3) the ECPP, and (4) the boundary scan frequency. For the ORCA OR2C15A, these values become 20/2 vertical STAR positions, 20 columns, 10 MHz, 20/2 horizontal STAR positions, 20 rows and 10 MHz. Solving the following expression for ECCP estimates 33,500 effective cycles are dedicated to a single PLB during one full sweep.

$$1.34\text{s} = \left(\frac{20}{2}\right)\left(20 \cdot \frac{\text{ECPP}}{10\text{ MHz}}\right) + \left(\frac{20}{2}\right)\left(20 \cdot \frac{\text{ECPP}}{10\text{ MHz}}\right) \tag{1}$$

To estimate an upper bound for the detection latency, the largest device in the Xilinx Virtex-4 family—the XC4VLX200—is used, which is a 116×192 PLB array containing eight LUTs per PLB and allows a boundary scan clock frequency up to 50 MHz. Accounting for increases in array size and configuration frequency, the detection latency scales to 17s as shown next.

$$t_{latency} = \left(\frac{116}{2}\right)\left(116 \cdot \frac{33{,}500}{50\text{ MHz}}\right) + \left(\frac{192}{2}\right)\left(192 \cdot \frac{33{,}500}{50\text{ MHz}}\right) \tag{2}$$

This expression, however, does not account for the two-fold increase in LUTs per PLB that also must be tested and reconfigured. Since the functional-based BIST strategies proposed by Dutt et al. [2008] indicate up to a four-fold decrease in detection latency from pseudo-exhaustive BIST strategies supplied by Emmert et al. [2007], the additional time required for a two-fold increase in LUTs per PLB may be completely offset. Another way to address the long average latencies required to detect errors is by using a hybrid of CED on a fine-grained scale along with the desired fault location technique. This combines the benefits of immediate fault detection of CED with efficiency and robustness of other fault location strategies. In the most straightforward

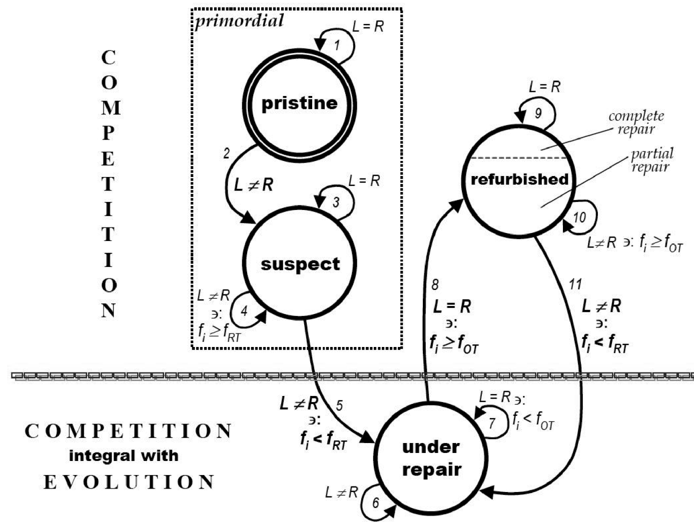

Fig. 13. States of an individual during its lifetime [DeMara and Zhang 2005].

case, combinational logical blocks are considered without memory elements, otherwise a checkpointing and rollback strategy is needed to recover proper state information [Huang et al. 2001].

4.2.3. Consensus-Based Evaluation of Competing Configurations. Whereas previous online Genetic Algorithm (GA) methods utilize an *N*-MR voting element, the Competitive Runtime Reconfiguration (CRR) proposed by DeMara and Zhang [2005] handles faults through a pairwise functional output comparison. Similar to previous GA methods, each of the two individuals is a unique configuration on the target FPGA exhibiting the desired functionality. CRR divides the FPGA into two mutually exclusive regions, allocating the *left half* configuration to one individual and the *right half* configuration to another individual in the population of alternate configurations. This detection method realizes a traditional Concurrent Error Detection (CED) arrangement that allocates mutually exclusive resources for each individual. The comparison results in either a discrepancy or a match between half-configuration outputs, which detects any single resource fault with certainty. This indicates the presence or absence of a FPGA resource fault for all inputs that articulate the fault when applied to a combinational logic module or a pipeline stage consisting of combinational logic.

The left and right individuals of the pairwise comparison are selected from their respective left and right populations to maintain resource exclusivity. Functionally identical, yet physically distinct, *Pristine* individuals developed at design-time compose the initial population. As Figure 13 shows, the left and right individuals remain Pristine as long as the left and right individuals exhibit matching outputs. Additionally, the fitness values of both individuals are increased to encourage selection of individuals exhibiting correct behavior. Upon detecting a discrepant output, however, the fitness state of both individuals are demoted and labeled as *Suspect*. Furthermore, the fitness values of both individuals are decreased to discourage selection of individuals exhibiting discrepant behavior. Over many pairings and evaluations, the fitness value of individuals utilizing faulty resources, and therefore their probability for selection, will be decreased regardless of pairing. Moreover, nonfaulty individuals that were previously paired with faulty individuals will eventually be exonerated.

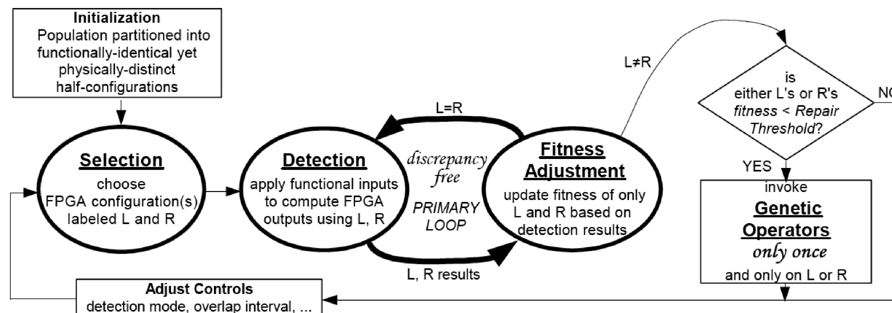

Fig. 14. Procedural flow for competing configurations [DeMara and Zhang 2005].

Figure 13 shows that the fitness state of individual i, which has been labeled as `Suspect`, is further demoted when its fitness (f_i) drops below the *repair threshold* (f_{RT}). Genetic operators are applied to the *`Under Repair`* individual, until its fitness rises above the *operational threshold* (f_{OT}). Selecting an operational threshold greater than the repair threshold increases confidence that the individual is, in fact, *`Refurbished`*. Further matching pairings with the `Refurbished` individual can result in either a *partial* or *complete regeneration* of lost functionality. Nonetheless, if the individual exhibits further discrepant behavior, its fitness state is returned to Under Repair and genetic operators are reapplied.

Figure 14 shows the CRR processes of *selection*, *detection*, *fitness adjustment*, and *evolution*. These processes identify individuals utilizing faulty resources and refurbish those individuals in the midst of the fault. The selection process determines the two individuals that will occupy the left and right regions. Typically, one of the halves is reserved as a "control" configuration where fault-free operational individuals, such as `Pristine`, `Suspect`, and `Refurbished` in that order, are always preferred. The other half supersedes these operational individuals with `Under Repair` individuals at a rate equal to the reintroduction rate. Genetically modified `Under Repair` individuals compete by being reintroduced into the operational throughput. The reintroduction rate can be adjusted to achieve a desired recovery goodput during the recovery process. This assumes that alternative configurations with fault-free behavior over a window of recent inputs remain available or have already been refurbished within the population.

Applying an input to the left and right individuals invokes the fitness adjustment process. As previously discussed, matching outputs results in increases to the fitness values of both individuals. Discrepant outputs decrease the fitness value and, consequently, the probability that either individual is selected again, with a steeper gradient. This process negatively or positively reinforces certain individuals by decreasing or increasing their fitness appropriately. If an individual's fitness is less than the repair threshold, a single application of genetic operators such as crossover and mutation are performed with a random `Pristine` individual. The checking logic is embedded in the individual and is dependent on the other half. Thus, if the checking logic in one of the halves experiences a fault, it will propagate to the other half, causing the fitness of the individuals to decrease. Additionally, the genetic operators may recover from faults in the checking logic. This implements a check-the-checker concept to enhance its fault tolerance. Variation of the reintroduction rate then allows control over how frequently the genetically modified offspring are allowed to compete with the rest of the population.

CRR exploits the normal operational inputs of the FPGA to evaluate the fitness of individuals. To establish confidence in an individual's fitness, more than one input is

evaluated for each individual; the more inputs evaluated, the greater the confidence. Since this method is not an exhaustive evaluation, CRR utilizes an *evaluation window* that specifies the number of inputs needed to gain a certain confidence in the individual's fitness. Over many pairings and fitness evaluations, CRR eventually forms a consensus from a population of individuals for a customized fault-specific recovery. For a 3×3 multiplier circuit, a configuration restoring full functionality was realized within a few thousand evaluations. During the recovery period when possible alternatives were being introduced to the application, data throughput averaged 87%, requiring 14% of the computations to be redundant. Fault exploitation was supported as experiments demonstrated LUTs exhibiting a single stuck-at-zero input being recycled as a function of three viable input variables after recovery. The recovery granularity observed was variable down to the resolution of a single LUT.

5. COMPARISON OF METHODS

This section details the metrics identified by Table I in Section 1.3. The metrics distinguish the various design philosophies along with their advantages and disadvantages. The metrics are separated into two categories: overhead-related metrics and sustainability-related metrics. The sections that follow evaluate each fault-handling method against these metrics and, where needed, clarify how these evaluations are obtained.

5.1. Overhead-Related Metrics

All fault-handling methods incur various overheads. This may include redundant physical resources, external storage necessary for configurations generated during design-time, or additional processing time associated with the fault-handling strategy. In order to simplify comparisons, estimates for values not explicitly provided by the authors of the methods are derived for comparison based on the Xilinx Virtex-4 device family as elaborated next.

5.1.1. Physical Resource Overhead. The *physical resource overhead* metric describes the amount of resources that an end-user must reserve to implement the fault-handling technique, in addition to that required by the original application design. Overheads for both logic and interconnect resources are listed in Table II. Some fault-handling methods, such as the coarse-grained method, partition entire areas of the FPGA and thus do not differentiate between logic and interconnect resource requirements. Resource overheads reported as a percentage of the application requirements are values supplied by the respective authors for those methods. Overheads reported as a percentage of the FPGA resources, however, are estimates based on the fault-handling strategy using the largest Virtex-4 device—XC4VLX200, 192×116 array—as a lower bound and the smallest Virtex-4 device—XC4VLX15, 64×24 array—as an upper bound [Xilinx 2007].

5.1.2. Throughput Reduction. Faults may occur in PLBs that are located along a critical timing path of the application. Reconfiguring the application to function correctly in the presence of such a fault may extend the length of the critical path, thereby increasing signal propagation delay. *Throughput reduction* is the penalty incurred by the application in the course of recovering from a single fault, expressed as a percentage of the application's original clock speed. Table II lists throughput reduction values that are reported by the respective methods. Methods utilizing a stochastic recovery process such as genetic algorithms are unable to provide tight bounds for throughput reduction.

Table II. Overhead-Related Metrics

			Metrics				
			Physical Resource Overhead		Throughput reduction	Detection Latency	Recovery time for single fault
			Logic	Inter-connect			
A priori Resource Allocations	Spare Configurations	Fine-grained	2–10% of application	None	14–45%	Not Addressed	[*44μs–64ms*]
		Functional-unit partitioning	200% of application		Not Provided	Negligible	None
		Coarse-grained	4–50% of FPGA		0–14%	Not Addressed	*64ms*
	Spare Resources	Sub-PLB Spares	8–20% of application	None	Not Provided	Not Addressed	Place&Route + *64ms*
		PLB Spares	1–41% of FPGA	9–50% of application	0–15%	Not Addressed	Place + *64ms*
Dynamic Recovery Processes	Offline Recovery	Incremental Rerouting	Subset of unutilized resources on FPGA		2–35%	Not Addressed	[2–12s] + *64ms*
		GA Recovery	Subset of unutilized resources on FPGA		Indeterminate	Not Addressed	Unbounded
		Augmented GA Recovery	Subset of unutilized resources on FPGA		Indeterminate	12 generations	38% decrease from GA Recovery
	Online Recovery	TMR w/ Single Module Recovery	200% of application		Indeterminate	Negligible	None
		Online BIST	4–11% of FPGA		0–16%	0–17s	None
		Competing Config.	100% of application		Indeterminate	Negligible	[popSize*44 μs –popSize**26ms*] or Unbounded

5.1.3. Detection Latency. Some of the surveyed methods incorporate fault detection and location techniques towards providing an integrated fault-handling solution. Others, such as the sub-PLB spare technique, however, rely on external fault detection mechanisms. For techniques that provide fault detection strategies, *detection latency* specifies the amount of time required for the fault-handling method to detect and/or locate the fault.

5.1.4. Recovery Time. The recovery time of a fault-handling method is the time required to restore complete functionality to the application implemented on the FPGA, as measured starting from the time the fault is detected and/or located. Since all fault-handling methods discussed address faults through FPGA reconfiguration, a portion of the time taken to recover from a single fault is due to the time taken to reconfigure the device, namely *configuration time*. Configuration times are calculated for the largest Xilinx Virtex-4 device, XC4VLX200, as an upper bound. Using the Virtex SelectMAP byte-wide parallel, continuous loading interface, configuration times are calculated using the following equation derived from Xilinx [2009]. We have

$$t_{config} = (bytes + 3) \cdot \frac{1}{f_{cclk}}, \tag{3}$$

where *bytes* is the size of the configuration file in bytes and f_{cclk} is the frequency of the configuration clock. A full-device bitstream for the XC4VLX200 is 6.12MB and thus requires 64ms using a 100 MHz configuration clock.

Some methods, such as fine-grained methods, rely on reconfiguring the FPGA with a partial configuration bitstream. For such methods, configuration times are calculated based on the size of the partial configuration bitstream. The height of a configuration frame for the Virtex-4 family is 16 PLBs [Lysaght et al. 2006]. Thus, the smallest possible partial configuration frame has a height of 16 PLBs and a width of one PLB. This partial bitstream is 4.2KB bytes in size whereas a bitstream for half of the device is 2.5MB in size. These bitstreams require 44μs and 26ms, respectively, for configuration using a 100 MHz configuration clock. Since partial configuration bitstreams do not reconfigure many parts of the device such as input/output buffers, the half-device bitstream is less than half of the full-device bitstream. The configuration times for the techniques that leverage partial reconfiguration are estimated based on the sizes of the smallest and largest partial reconfiguration bitstream the method uses.

5.1.5. Summary. The authors of the fine-grained approach describe the AFTBs as consisting of a set of PLBs and the interconnect resources associated with them. As listed in Table II, the method does not address faults in the interconnect resources directly dedicated to specific PLBs, since these appear as faults in the PLBs. Therefore, the physical resource overhead for the method is only listed as the sum of logic resources (PLBs) required to configure the logic in the desired manner. In general, the a priori methods listed in Table II have a physical resource overhead directly proportional to the size of the spare configuration or the granularity of the a priori resource allocation. Due to this, these methods involve planning at the design stage, and there are limitations concerning the portability of a developed scheme across applications. Except for the functional-unit partitioning system developed by the TMRSB method described in Section 3.1.2, none of the surveyed a priori methods has integrated fault detection or location capability. Significantly, for the TMRSB method, the physical resource overhead is 200% of the size of the application. Also, for this method, the throughput reduction varies depending on the configuration selected to replace the fault-affected configuration. Though a quantitative analysis of the throughput reduction incurred by the technique can be attempted, the authors do not provide estimates for the throughput reduction across the standby configurations, as is the case with the sub-PLB spares-based technique. Since the outputs for the system are based on majority voting of the outputs of individual functional units, the system provides for continuous operation at some degraded level of throughput even in the presence of a fault. As compared to methods relying on spare configurations, the spare resource-based techniques incur overheads related to placement and/or routing during fault recovery. However, unlike the spare configuration-based methods, these techniques do not require additional design-time effort to anticipate and accommodate expected runtime fault scenarios.

GA-based and other dynamic offline techniques do not define a fixed resource overhead. These techniques leverage redundancy at varying levels of granularity—as low as spare PLB I/O pins and as high as a set of spare PLBs—in order to realize a response to faults. All the online recovery methods except the online BIST method provide for immediate fault detection. Due to the roving nature of the STARs in the online BIST method, the fault detection latency can be as high as 17s. On the other hand, this method provides quick recovery once the fault is identified. This is also true of TMR with single-module-recovery methods, which provides for the device to remain functional in the presence of a fault at the cost of 200% physical resource overhead. It is not possible to estimate throughput reduction for dynamic methods that rely on evolutionary techniques due to their inherently stochastic nature. Of the dynamic

methods, the TMR with single module recovery method provides for immediate fault resolution since the majority output from the voter guarantees sustained throughput. However, until the faulty module is refurbished, the system remains susceptible to a fault in either of the other two fault-free modules. The recovery time for online BIST is factored into the detection latency since tests are conducted on inactive PLBs. For GA-based methods, the recovery time is unbounded. In order to recover from a fault, the GA must generate a configuration, or, in the case of competing configurations, find a preexisting fault-free configuration through a process of reconfiguring the device.

5.2. Sustainability Metrics

Sustainability metrics provide a qualitative comparison of the benefits provided by the various methods. In conjunction with the overheads, these metrics provide insight into the trade-offs involved in choosing a specific method.

5.2.1. Fault Exploitation. By nature, all fault-handling methods are able to bypass faulty resources. Methods that are capable of *fault exploitation* can reuse residual functionality in fault-affected elements. Such methods increase the effective size of the resource pool by virtue of recycling faulty resources. For example, authors of the online BIST method state that if an LUT has a cell stuck-at-0 fault the LUT can still be used to implement a function that requires a 0 to be in that cell.

5.2.2. Recovery Granularity. Recovery granularity defines the smallest set of FPGA resources in which faults can be handled. The set of resources identified as being fault-affected is excluded from further utilization by the methods that are not capable of fault exploitation.

5.2.3. Fault Capacity. Each method is capable of handling a varying number and types of faults. *Fault capacity* defines the number of fault-free resource units required to ensure that the system continues to function if there is a single additional fault, irrespective of the location of the faults. The fault capacity for the surveyed methods is listed in Table III and discussed subsequently.

5.2.4. Fault Coverage. Fault coverage defines the ability of a method to handle faults in various components of the FPGA. Some of the surveyed methods can distinguish between transient and permanent faults. All methods surveyed handle faults occurring within logic resources. None of the surveyed methods handles faults occurring within I/O blocks. Only some of the methods surveyed, however, address faults occurring within interconnect resources. Since the interconnect resources can compose up to 90% of the FPGA area [Hanchek and Dutt 1998], the probability of a fault occurring within interconnect is higher than that of an occurrence within PLB logic resources. As such, interconnect fault coverage is an important metric outlined by Table III.

5.2.5. Critical Components. Critical components are those FPGA components that are essential for fault-free operation of the fault-handling method. If the fault-handling techniques do not provide coverage for these components, a single fault in one of these can cause the failure of the fault-handling technique, thus defining a single point of failure. Table III provides a list of components that are critical to each technique. FPGAs such as the Virtex-4 XC4VFX140 include embedded microprocessors or can realize such hardware equivalents with its PLB logic and interconnect. Some techniques may incorporate external procedures such as a custom place-and-route mechanism within the device by utilizing these device capabilities. While such instantiations are not external to the actual device, they are critical components if the technique does not provide coverage for faults in the components used to realize them.

Table III. Sustainability Metrics

			Metrics				
			Fault Exploitation	Recovery Granularity	Fault Capacity	Interconnect Fault Coverage	Critical Components
A-priori Resource Allocations	Spare Configurations	Fine-grained		PLB	One PLB per tile	Implicit, inside partitions	Config memory
		Functional-unit		Variable	Three configurations	Implicit, inside partitions	Voter, Config memory
		Coarse-grained		Column(s) of PLBs	Column(s) of PLBs	Implicit, inside partitions	Config memory
	Spare Resources	Sub-PLB Spares		Lookup Table	One LUT input or One BLE per PLB	None	Custom placer and router
		PLB Spares		PLB	One PLB per row	None	Custom placer
Dynamic Recovery Processes	Offline Recovery	Incremental Rerouting		PLB	One PLB	Explicit	Custom placer and router
		GA Recovery	√	Variable	Indeterminate	Implicit	Processor and Config memory
		Augmented GA Recovery	√	Variable	Indeterminate	Implicit	Processor and Config memory
	Online Recovery	TMR w/Single Module Recovery	√	Variable	Three configurations	Implicit	Voter, Processor and Config memory
		Online BIST	√	Lookup Table	One H-STAR & One V-STAR	Explicit	Processor and Config memory
		Competing Configurations	√	Variable	Two configurations	Implicit	Processor and Config memory

5.2.6. Summary. A priori allocations depend on occluding faulty resources for creating recovery configurations. The dynamic methods bypass the faulty resource partially or completely. In addition, these techniques are also able to leverage residual functionality in the fault-affected resources. The recovery granularity for the a priori allocations is proportional to the level at which redundant resources are allocated at design time. Thus, the metric also determines the net decrease in the amount of hardware redundancy per fault for a priori allocations. The GA-based methods provide a variable level of fault granularity, as they refurbish fault-affected configurations without requiring specific details regarding the precise location of the fault. Additionally, these methods generate refurbished configurations by leveraging residual functionality in PLBs in a manner that is transparent to the user. For example, a LUT with a stuck-at-one fault on an input pin might still be utilized in an evolved design which synthesizes a logic expression using that input.

The fault capacity of a technique indicates the expected lifetime of the device. The fine-grained approach has a fault capacity limited by the number of redundant PLBs in each tile. This method has a capacity for $(n-1)$ faults, where n is the number of redundant PLBs incorporated in each tile at design time. After $(n-1)$ faults have occurred, complete functionality cannot be guaranteed for any additional faults, if all the preceding faults occurred in PLBs in the same tile. The functional-unit method is

capable of tolerating additional faults as long as three fully functional configurations exist. The coarse-grained technique has a fault capacity limited by the number of re-programmable columns; however, each such column may consist of multiple columns of FPGA PLBs. Spare resource-based techniques can tolerate additional faults as limited by the number and the location of spare resources. For example, the PLB spare-based technique allocates a fixed number of spare PLBs per row. With n spare PLBs in each row, a maximum of $(n - 1)$ faults can occur before fault-free operation cannot be guaranteed in the event of an additional fault. The authors of the online BIST technique state that the method can tolerate one additional fault as long as there is a pair of fault-free H-STAR and V-STAR. For GA-based techniques, the fault capacity is listed as indeterminate, since these methods refurbish resources for reuse at a rate dependant on the evolutionary algorithm, and parameters controlled by the designer of the algorithm. These techniques provide for control over the useful throughput that can be obtained at repair-time, at some trade-off for slower refurbishment.

All the surveyed methods provide for coverage of faults in logic elements. Some also provide coverage for faults in the interconnect resources: either explicitly by diagnosing and occluding such faults, or implicitly by realizing recovery configurations that provide functional recovery in spite of faulty interconnect. The critical components column in Table III provides assistance in selecting the most suitable fault recovery method based on the target application that has to be implemented on the FPGA. The points-of-failure in each recovery method can be overcome by employing a matrix of multiple techniques that provide for complete fault coverage. This, in conjunction with external fault-handling mechanisms for the resources not covered by the techniques described here, can provide a high level of system fault tolerance. As an example, the competing configurations method can be extended by using a TMR-enabled memory subsystem consisting of triplicate memory and a voter.

6. CONCLUSION

This work provides an overview of the characteristics and advantages of runtime fault-handling methods for transient and permanent faults in SRAM-based FPGAs. Several fault tolerance techniques that rely on custom-built architectures and devices are not covered in this survey. Examples of such approaches include embryonics [Ortega-Sanchez et al. 2000], distributed embedded systems [Dave and Jha 1997], and several self-adaptive systems as described in Mesquita [2008]. Since commercial SRAM-based FPGAs continue to play a vital role in the design of systems, fault-handling methods for such devices remain a topic of active research interest.

From this study, a general consensus emerges that the fault-tolerant technique adopted for a particular application should choose between a specific set of performance and sustainability trade-offs. Selection of a particular fault-tolerant methodology implies weighing overheads incurred against specific sustainability requirements such as those of long missions where multiple faults may occur and unit-level spares are unavailable. As listed in Table I, the overhead metrics include physical resource overheads, throughput reduction, latency in detection, and recovery time. For mission-critical applications, one or more of these overheads might dominate the selection of the fault tolerance methodology. For example, real-time processing and control circuit implementations may place a higher premium on maintaining throughput within tolerance limits, but applications used for postprocessing data might not be critically affected by increased detection latency or recovery time, by virtue of being able to adapt the processing rate of the throughput data.

The a priori allocations are limited by the total number of consecutive randomly located faults they can handle, but tend to provide low detection latency and faster fault recovery. Methods such as those based on evolutionary algorithms provide the potential

for tolerating an increased number of faults, but at the cost of increased recovery time. Significant trade-offs include design footprint size, postfault recovery time, spare-resource overhead, and, for online systems, availability during recovery. As discussed in Section 4.2.1, transient faults are typically addressed by some scrubbing scheme. Whereas some fault-handling methods distinguish between transient and permanent faults by explicitly incorporating scrubbing or rollback [Emmert et al. 2007; Garvie and Thompson 2004; Ratter 2004], others treat every fault as if it were permanent. In these cases, applying a new configuration to avoid the perceived permanent fault essentially accomplishes the same effect of scrubbing by reestablishing a functionally correct configuration and reevaluating the last incorrect computation. Whereas the method restores functionality affected by the fault, it may still consider the resource faulty, unnecessarily reducing the redundancy index.

As most methods suggest the system be in an offline state during the entire fault recovery process, a particular fault may only articulate itself in a small percentage of the output space. In such a situation, an application with low sensitivity to faulty inputs may benefit from the faulty system remaining in an operational state during the fault recovery process. For instance, Sharma et al. [2007] illustrate *recovery goodput,* which measures the percentage of correct or useful outputs provided during the recovery process. Whereas recovery goodput measurements are largely a result of the type of fault and application, most fault-handling methods do not consider goodput during fault recovery. The previously discussed competing configurations method can maintain a required level of goodput by adjusting the rate at which configurations under repair are implemented on the FPGA.

There exist several issues that need to be addressed with regards to selecting a complete fault-handling technique for SRAM-based FPGAs. Though there are several approaches for tolerating faults in the interconnect resources, the choices are limited when it comes to online isolation of such faults. Thus, the integration of interconnect-fault and logic-fault strategies remains a major challenge. The challenge in extending the approach to sequential logic circuits is primarily one of being able to formulate a strategy for evaluating the fitness of alternative designs. A general strategy to enable sustained recovery of large sequential circuits without exhaustive resource testing remains to be addressed.

REFERENCES

Abramovici, M., Stroud, C. E., and Emmert, J. M. 2004. Online BIST and BIST-based diagnosis of FPGA logic blocks. *IEEE Trans. VLSI Syst. 12*, 12, 1284–1294.

Actel. 2005. Radiation-Hardened FPGAs datasheet. http://www.actel.com/documents/RadHard_DS.pdf.

Adell, P. and Allen, G. 2008. Assessing and mitigating radiation effects in Xilinx FPGAs. *JPL Publ. 08-09*.

Altera. 2009. Stratix IV device handbook. http://www.altera.com/literature/hb/stratix-iv/stratix4_handbook.pdf.

Atmel. 2007. Rad hard reprogrammable FPGA ATF280E. Atmel datasheet 7750.

Babb, J., Frank, M., Lee, V., Waingold, E., Barua, R., Taylor, M., Kim, J., Devabhaktuni, S., and Agarwal, A. 1997. The RAW benchmark suite: Computation structures for general purpose computing. In *Proceedings of the 5th Annual IEEE Symposium on FPGAs for Custom Computing Machines*. 134–143.

Baldacci, S., Zolesi, V., Cuzzocrea, F., and Ramacciotti, T. 2003. SEU tolerant controls for a space application based on dynamically reconfigurable FPGA. In *Proceedings of the Military and Aerospace Programmable Logic Devices (MAPLD) Workshop*.

Bridgford, B., Carmichael, C., and Tseng, C. W. 2008. Single-event upset mitigation selection guide. Xilinx Application Note 987.

Carmichael, C., Caffrey, M., and Salazar, A. 2000. Correcting single-event upsets through virtex partial configuration. Xilinx Application Note 216.

Cheatham, J. A., Emmert, J. M., and Baumgart, S. 2006. A survey of fault tolerant methodologies for FPGAs. *ACM Trans. Des. Autom. Electron. Syst. 11*, 2, 501–533.

DAVE, B. P. AND JHA, N. K. 1997. COFTA: Hardware-software co-synthesis of heterogeneous distributed embedded system architectures for low overhead fault tolerance. In *Proceedings of the 27th Annual International Symposium on Fault-Tolerant Computing*. 339–348.

DEMARA, R. F. AND ZHANG, K. 2005. Autonomous FPGA fault handling through competitive runtime reconfiguration. In *Proceedings of the NASA/DoD Conference on Evolvable Hardware*. 109–116.

DONG-MEI, L., ZHI-HUA, W., LI-YING, H., AND QIU-JING, G. 2007. Study of total ionizing dose radiation effects on enclosed gate transistors in a commercial CMOS technology. *Chinese Phys. 16*, 12, 3760–3765.

DOUMAR, A. AND ITO, H. 2003. Detecting, diagnosing, and tolerating faults in SRAM-based field programmable gate arrays: A survey. *IEEE Trans. VLSI Syst. 11*, 3, 386–405.

DUTT, S., SHANMUGAVEL, V., AND TRIMBERGER, S. 1999. Efficient incremental rerouting for fault reconfiguration in field programmable gate arrays. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. 173–176.

DUTT, S., VERMA, V., AND SUTHAR, V. 2008. Built-In-self-test of FPGAs with provable diagnosabilities and high diagnostic coverage with application to online testing. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst. 27*, 2, 309–326.

EMMERT, J. M. AND BHATIA, D. K. 2000. A fault tolerant technique for FPGAs. *J. Electron. Test. 16*, 6, 591–606.

EMMERT, J. M., STROUD, C. E., AND ABRAMOVICI, M. 2007. Online fault tolerance for FPGA logic blocks. *IEEE Trans. VLSI Syst. 15*, 2, 216–226.

GARVIE, M. AND THOMPSON, A. 2004. Scrubbing away transients and jiggling around the permanent: Long survival of FPGA systems through evolutionary self-repair. In *Proceedings of the IEEE International On-Line Testing Symposium*. 155–160.

GERICOTA, M. G., ALVES, G. R., SILVA, M. L., AND FERREIRA, J. M. 2008. Reliability and availability in reconfigurable computing: A basis for a common solution. *IEEE Trans. VLSI Syst. 16*, 11, 1545–1558.

HANCHEK, F. AND DUTT, S. 1998. Methodologies for tolerating cell and interconnect faults in FPGAs. *IEEE Trans. Comput. 47*, 1, 15–33.

HUANG, W.-J., MITRA, S., AND MCCLUSKEY, E. J. 2001. Fast run-time fault location in dependable FPGAs. CRC Tech. rep. 01-5, Center for Reliable Computing, Department of Electrical Engineering and Computer Science, Stanford University. May.

HUTTON, M., ROSE, J., AND CORNEIL, D. 1997. Generation of synthetic sequential benchmark circuits. In *Proceedings of the ACM 5th International Symposium on Field-Programmable Gate Arrays*.

KATZ, D. S. AND SOME, R. R. 2003. NASA advances robotic space exploration. *Computer*. 52–61.

KEYMEULEN, D., ZEBULUM, R. S., JIN, Y., AND STOICA, A. A. S. A. 2000. Fault-Tolerant evolvable hardware using field-programmable transistor arrays. *IEEE Trans. Reliabil. 49*, 3, 305–316.

KIZHNER, S., PATEL, U. D., AND VOOTUKURU, M. 2007. On representative spaceflight instrument and associated instrument sensor web framework. In *Proceedings of the IEEE Aerospace Conference*. U. D. Patel Ed., 1–10.

LACH, J., MANGIONE-SMITH, W. H., AND POTKONJAK, M. 1998. Low overhead fault-tolerant FPGA systems. *IEEE Trans. VLSI Syst. 6*, 2, 212–221.

LAKAMRAJU, V. AND TESSIER, R. 2000. Tolerating operational faults in cluster-based FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 187–194.

LOHN, J., LARCHEV, G., AND DEMARA, R. 2003. Evolutionary fault recovery in a Virtex FPGA using a representation that incorporates routing. In *Proceedings of the Parallel and Distributed Processing Symposium*.

LYONS, R. E. AND VANDERKULK, W. 1962. The use of triple-modular redundancy to improve computer reliability. *IBM J. Res. Devel. 6*, 2, 200–209.

LYSAGHT, P., BLODGET, B., MASON, J., YOUNG, J. A. Y. J., AND BRIDGFORD, B. A. B. B. 2006. Enhanced architectures, design methodologies and CAD tools for dynamic reconfiguration of xilinx FPGAs. In *Proceedings of the International Conference on Field Programmable Logic and Applications*. 1–6.

MESQUITA, A. 2008. Introduction to evolvable hardware: A practical guide for designing self-adaptive systems. *Genetic Program. Evolv. Mach. 9*, 3, 275–277.

MILLER, J. F., THOMSON, P., AND FOGARTY, T. 1997. Designing electronic circuits using evolutionary algorithms. Arithmetic circuits: A case study. In *Genetic Algorithms and Evolution Strategies in Engineering and Computer Science*, D. Quagliarella, J. Periaux, C. Poloni and G. Winter, Eds., Wiley, 105–131.

MITCHELL, M. 1996. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA.

MITRA, S., HUANG, W. J., SAXENA, N. R., YU, S., AND MCCLUSKEY, E. J. 2004. Reconfigurable architecture for autonomous self-repair. *IEEE Des. Test Comput. 21*, 3, 228–240.

OREIFEJ, R. S., SHARMA, C. A., AND DEMARA, R. F. 2006. Expediting GA-based evolution using group testing techniques for reconfigurable hardware. In *Proceedings of the IEEE International Conference on Reconfigurable Computing and FPGAs*. 1–8.

ORTEGA-SANCHEZ, C., MANGE, D., SMITH, S., AND TYRRELL, A. 2000. Embryonics: A bio-inspired cellular architecture with fault-tolerant properties. *Genetic Program. Evolv. Mach. 1*, 3, 187–215.

PARRIS, M. G. 2008. Optimizing dynamic logic realizations for partial reconfiguration of field programmable gate arrays. Masters thesis, School of Electrical Engineering and Computer Science, University of Central Florida.

RATTER, D. 2004. FPGAs on mars. *Xcell J. 50*, 8–11.

ROOSTA, R. 2004. A comparison of radiation-hard and radiation-tolerant FPGAs for space applications. NASA Electronic Parts and Packaging (NEPP) Program JPL D-31228.

ROSS, R. AND HALL, R. 2006. A FPGA simulation using asexual genetic algorithms for integrated self-repair. In *Proceedings of the 1st NASA/ESA Conference on Adaptive Hardware and Systems*. 301–304.

SCHWEFEL, H.-P. AND RUDOLPH, G. 1995. Contemporary evolution strategies. In *Advances in Artificial Life*. Springer, 891–907.

SHANTHI, A. P., VIJAYAN, B., RAJENDRAN, M., VELUSWAMI, S., AND PARTHASARATHI, R. 2002. GA based on-line testing and recovery for critical digital systems. In *Proceedings of the HiPC Workshop on Soft Computing*. 81–89.

SHARMA, C. A. 2008. Sustainable fault-handling of reconfigurable logic using throughput-driven assessment. Doctoral dissertation, School of Electrical Engineering and Computer Science, University of Central Florida.

SHARMA, C. A., DEMARA, R. F., AND SARVI, A. 2007. Self-healing reconfigurable logic using autonomous group testing. *ACM Trans. Auton. Adapt. Syst.*.

TRIMBERGER, S. 1993. A reprogrammable gate array and applications. *Proc. IEEE 81*, 7, 1030–1041.

VIGANDER, S. 2001. Evolutionary fault repair of electronics in space applications. Masters thesis, Department of Computer and Information Science, Norwegian University of Science and Technology (NTNU), Trondheim, Norway. 50.

WELLS, B. E. AND LOO, S. M. 2001. On the use of distributed reconfigurable hardware in launch control avionics. In *Proceedings of the 20th Digital Avionics Systems*.

WIRTHLIN, M., JOHNSON, E., ROLLINS, N., CAFFREY, M., AND GRAHAM, P. 2003. The reliability of FPGA circuit designs in the presence of radiation induced configuration upsets. In *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. 133–142.

XILINX. 2007. Virtex-4 family overview. Xilinx Data Sheet 112.

XILINX. 2008. Radiation-Tolerant virtex-4 QPro-V family overview. Xilinx Data Sheet 653.

XILINX. 2009. Virtex-4 FPGA configuration user guide. Xilinx User Guide 071.

YANG, S. 1991. Logic synthesis and optimization benchmarks user guide version 3.0. Tech. rep. Microelectronics Center of North Carolina.

YUI, C. C., SWIFT, G. M., AND CARMICHAEL, C. 2003. SEU mitigation of xilinx virtex II FPGAs for critical flight applications. In *Proceedings of the IEEE Nuclear and Space Radiation Effects Conference*.

ZHANG, K., BEDETTE, G., AND DEMARA, R. F. 2006. Triple modular redundancy with standby (TMRSB) supporting dynamic resource reconfiguration. In *Proceedings of the IEEE Systems Readiness Technology Conference*. 690–696.

Received April 2009; revised October 2009; accepted December 2009