SUSTAINABLE FAULT-HANDLING OF RECONFIGURABLE LOGIC USING THROUGHPUT-DRIVEN ASSESSMENT

by

CARTHIK ANAND SHARMA
B.Tech. Kakatiya University, 2001
M.S. University of Central Florida, 2004

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Engineering
in the School of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Summer Term
2008

Major Professor: Ronald F. DeMara

ABSTRACT

A sustainable *Evolvable Hardware (EH)* system is developed for SRAM-based reconfigurable *Field Programmable Gate Arrays (FPGAs)* using outlier detection and group testing-based assessment principles. The fault diagnosis methods presented herein leverage throughput-driven, relative fitness assessment to maintain resource viability autonomously. Group testing-based techniques are developed for adaptive input-driven fault isolation in FPGAs, without the need for exhaustive testing or coding-based evaluation. The techniques maintain the device operational, and when possible generate validated outputs throughout the repair process.

Adaptive fault isolation methods based on discrepancy-enabled pair-wise comparisons are developed. By observing the discrepancy characteristics of multiple *Concurrent Error Detection (CED)* configurations, a method for robust detection of faults is developed based on pairwise parallel evaluation using Discrepancy Mirror logic. The results from the analytical FPGA model are demonstrated via a self-healing, self-organizing evolvable hardware system. Reconfigurability of the SRAM-based FPGA is leveraged to identify logic resource faults which are successively excluded by group testing using alternate device configurations. This simplifies the system architect's role to definition of functionality using a high-level *Hardware Description Language (HDL)* and system-level performance versus availability operating point. System availability, throughput, and mean time to isolate faults are monitored and maintained using an *Observer-Controller* model. Results are demonstrated using a *Data Encryption Standard*

(DES) core that occupies approximately 305 FPGA slices on a Xilinx Virtex-II Pro FPGA. With a single simulated stuck-at-fault, the system identifies a completely validated replacement configuration within three to five positive tests. The approach demonstrates a readily-implemented yet robust organic hardware application framework featuring a high degree of autonomous self-control.

I dedicate this to all those who believed in me, listened to me, and helped me. I wish to thank my grand mother Mrs. G. Bhageerathi Ammal, father Mr. V. Narayan, and sister Mrs. Gina Kartik for their phenomenal support in the 25 years I have spent in school. I also wish to thank Dr. DeMara for his patient guidance and technically-sound advice, Prerona Chakravarty for the mental and moral support, Kening Zhang for being the best colleague possible, and all my roommates past and present for their kindness.

ACKNOWLEDGMENTS

The research presented in this dissertation was supported in part by NASA Intelligent Systems NRA Contract NNA04CL07A.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS/ABBREVIATIONS

ADAS	Advanced Data Acquisition System
AGT	Autonomous Group Testing
BIST	Built-In Self Test
CBE	Consensus Based Evaluation
CED	Concurrent Error Detection
CGT	Combinatorial Group Testing
CLB	Configurable Logic Block
CRC	Cyclic Redundancy Check
CRR	Competitive Runtime Reconfiguration
DES	Data Encryption Standard
DV	Discrepancy Value
EH	Evolvable Hardware
FIAT	Fault Insertion and Analysis Tool
FPGA	Field Programmable Gate Array
GA	Genetic Algorithm
HDL	Hardware Description Language
LUT	Look-Up Table
MTBF	Mean Time Between Failures
MTTR	Mean Time To Recover
TMR	Triple Modular Redundancy
VLSI	Very Large Scale Integrated

CHAPTER 1: INTRODUCTION

Reliable and efficient detection, isolation, and handling of failures within electronic circuits are fundamental issues in the design of dependable devices. With production exceeding 100 million units per year, SRAM-based FPGA devices are frequently used in a wide range of embedded applications requiring high levels of reliability and availability.

1.1. Need for Evolvable Hardware Regeneration Methods and Group Testing-based Fault Diagnosis

Reconfigurable devices, such as FPGAs, enable new fault handling techniques based on *evolvable hardware regeneration*. Evolvable hardware regeneration techniques use the principle of biological evolution to handle faults. Using evolutionary techniques such as genetic algorithms and cellular automata, the existing redundant hardware resources are reused or rewired to occlude the fault. The repair process can take place *online* when the hardware is in active use, or *offline* when the regeneration occurs as part of a process outside the normal computation dataflow.

Such techniques are highly relevant to many embedded device applications, including remote sensing, applications in hazardous environments, and space missions. For instance, deep space satellites such as Stardust contain over 100 FPGA devices [1] while NASA terrestrial applications routinely employ FPGAs extensively for tasks ranging from launch control to signal processing. SRAM-based FPGAs are of significant

importance due to their high density, unlimited reprogrammability, and growing use in mission-critical/safety-impacting applications.

Depending on the application, these devices encounter harsh environments of mechanical/acoustical stress, high ionizing radiation, and thermal stress. Simultaneously, they are required to operate reliably for long durations with limited or absent capabilities for diagnosis/replacement in the case of remote applications. For example, in Aerospace Technology, Space Science, and Earth Science enterprises, the impact from increased safety and autonomy for FPGAs is highly relevant. On-going research at Ames [2] and JPL [3] has focused specifically on employing the reconfigurability inherent in various field programmable devices to increase their reliability and autonomy using evolutionary mechanisms.

Ground-based applications of FPGAs such as data acquisition devices and instrumentation systems seek to incorporate self-repair capabilities and provide extended calibration cycles. One such application is Kennedy Space Center's *Advanced Data Acquisition System (ADAS)* [4]. ADAS is a signal acquisition and processing system for launch control measurements typical of real-time NASA applications that heavily utilize FPGAs and have high reliability, availability, and maintainability requirements. Some target components that will benefit from evolvable hardware repair include Analog Signal Modules, Digital Signal/Control Modules, and Power Management Modules.

There is the need to integrate multiple phases of the fault handling process in an integrated manner. Further, this should ideally be done while maintaining the uptime,

and availability of the reconfigurable device. Evolutionary mechanisms can actively restore mission-critical functionality in SRAM-based FPGA devices. They provide an attractive alternative to device redundancy for resolving permanent degradation due to radiation-induced stuck-at-faults, thermal fatigue, oxide breakdown, electro-migration, and other failures. Potential benefits include recovery without the increased weight and size normally associated with spares. Without regeneration, spare capacity is finite. Therefore, an evolutionary fault handling strategy that relies upon resource recycling by means of leveraging the reconfigurability of FPGAs is required. Regeneration also provides for graceful degradation of performance with time, where resources are constantly recycled with minimal impact on system availability. The capability to recycle resources at a variable rate, as afforded by evolutionary mechanisms provides the capability to delay refurbishment to maintain required availability and throughput requirements. Such a strategy would rely upon fault isolation to accelerate the evolutionary repair. However, failures need not be precisely diagnosed due to automatic evaluation of FPGA residual functionality through intrinsic assessment using a specified fitness function.

Evolutionary mechanisms rely upon efficient fault detection and isolation schemes. Fault detection triggers the regeneration operation. Robust fault detection techniques are required to detect fault and failures with a low latency. Fault location methods provide inputs to the repair mechanism which accelerate the repair process, and reduce the search space of candidate solutions to the fault scenario. The fault isolation technique identified

in this work is one such method for isolating faults with low latency and minimal overheads.

1.2. Fault Handling in Reconfigurable Devices

An operational *failure* occurs when the service delivered deviates from its as-built specification. A resource *fault* is the cause of such failures. *Fault handling* refers to the entire process by which potential or actual failures are dealt with. Ideally, fault handling maintains failure-free functionality.

The process of improving fault handling typically involves *detection, isolation, diagnosis,* and *repair*. The detection phase consists of identifying the presence of a fault in the device. A fault is said to be *detected* when the effects of a corresponding failure is observed. Depending on whether the inputs applied manifest an observable failure, the fault is either be *perpetually articulated* or *intermittently articulated*. The articulation of the fault, and hence its potential for detection, relies on the mapping of the functional design to the physical resources. Once a failure has been detected, it may be possible to isolate the faulty resources. Fault *location* or *isolation* determines the physical location of the faulty components. The granularity of isolation may vary, depending on the architecture, the algorithm, and the isolation tools available. Fault *diagnosis* thus deals with the determination of the symptoms and the reason behind the observed failure. A *symptom* is an observable effect of a fault. Failures are among the most easily observed symptoms of a fault and are the basis for the isolation methods developed in the proposed research. The diagnosis phase may involve obtaining the response of the device to an

exhaustive set of inputs using a tool designed solely for performing diagnostic tests. The last phase consist of fault repair, wherein the effects of the fault are ameliorated to reduce the occurrence or impact of future failures.

The particular fault handling approach can be classified on the basis of when the faults are accounted for in the development cycle. *Design Time* approaches place the emphasis on *Fault Avoidance* strategies through design strategies that avoid the occurrence of faults. *Execution Time* or *Run Time* approaches tackle the problem by using *Fault Tolerance* and *Fault Evasion* methods. A *Fault-Tolerant system* is characterized by its ability to provide uninterrupted service, conforming to the desired levels of reliability even in the presence of faults. A *dependable* or *reliable system* is one which offers a level of service that is characterized by its *availability* or readiness for use when desired.

Embedded fault-handling techniques can also be broadly categorized as diagnostic-based [5], coding-based [6], or redundancy-based [7], depending on the method used to implement fault-handling. Diagnostic-based techniques execute a supplemental procedure that applies a test vector to a subset of the physical resources. While diagnostics offer a compact approach, they can suffer from unavailability of throughput during testing, a large detection latency, and intractability of search as the number of physical resources and their piecewise interactions grow large [8]. Coding-based techniques map the input values to an alternate representation to enforce constraints on the validity of the outputs. Such encodings based on parity, CRC, Berger, and other codes can be effective for data storage and transmission [9]. However, they preclude the occurrence of failures that might map one valid codeword onto another, and thus their

general applicability for FPGA logic resources is limited. To avoid such limitations, embedded techniques frequently rely on component or system-level redundancy.

Fault detection methods are central to fault handling strategies. Fault detection can be carried out by a mechanism outside the domain of the system under observation. In some cases it is not feasible to have a separate supervisory system in addition to the system under test. For such a system to be fault tolerant, it is imperative that the fault detection tool or system used be fault-tolerant as well, since it will be a part of the system under observation. To maintain acceptable availability levels, reduction of the fault detection latency is essential. An evolutionary hardware repair strategy can use the information provided by the fault isolation strategy to speed up the repair process. In CRR, accurate knowledge of the physical location of the fault can provide useful inputs to the repair algorithm. The fault detection and isolation strategy used should ideally be capable of identifying and locating faults without requiring special test inputs, or an interruption in the normal data throughput. The hardware resources used by the detector should be minimal, in order to reduce the number of points of failure, and to conserve floor space. The detector should be *fault-secure* meaning it does not propagate incorrect outputs in the presence of a fault. Section 2.2 provides a detailed overview of selected fault detection strategies.

1.3. Individual and Population-Centric Fault Assessment

Traditional approaches to fault-detection typically rely on coding-based schemes or redundancy using a single voter, comparator, or error detector. Those fault checkers

possess a single point-of-failure exposure involving the detector elements, or must rely upon special test-vectors or data encodings to isolate them. Detector components in the reliability path have been referred to as golden elements [10] because the fault-handling strategy relies on them to be fault-free. Also, significantly, previous methods test individual configurations or resource units to evaluate their *fitness*. While such individual-fitness centric methods provide fault coverage on the device level, they do not lend to an adaptive, evolving system.

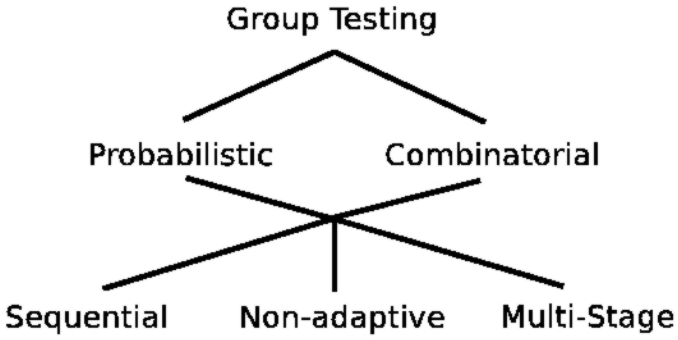
In a redundant system, the problem of fault detection can be simplified by the fact that if there are no faults, then the outputs of the redundant elements should be identical. An observed deviation from the this property would imply that the disagreement is a result of a failure in at least one of the redundant components. Natural laws of competition, as seen in biological evolution can be applied to improve the performance of electronic circuits. In fault-detection, a deviation from the normal behavior, as determined by comparison with another individual design, signifies a state of decreased fitness, as a result of the manifestation of a hardware fault.

The idea of competition can also be extended to the repair problem, using competitive pairing as a fitness evaluation technique. Traditional GAs use an absolute measure of fitness for the individuals to search for improved solutions. In this work, the fitness of an individual design configuration depends on relative measures computed over a period of time. The proposed fitness assessment process involves accumulation of discrepancies across multiple random pairings with other individuals from the population. Such a population-based approach greatly simplifies the process of fault diagnosis, and uses the

fact that the circuit under test continues to operate for the duration of its useful lifetime to accrue information about the performance of competing individuals. By keeping the method of fault isolation simple, the cost of repair is reduced and amortized over time, thus providing a fault-secure system without acceptable overhead.

1.4. Group Testing Techniques and Applications to Fault Tolerance

Group testing is a field of mathematics concerned with the development of efficient algorithms to identify defective members from a large population. The origin of group testing is attributed to Robert Dorfman who proposed the first application during World War II. He devised a scheme for testing blood samples from millions of United States army draftees for cases of syphilis [11]. He proposed that the blood samples be pooled for testing, in order to reduce the number of tests required and the associated cost and effort. If a pool of samples tested positive for syphilis, then the samples that contributed to the pool would be subject to individual testing. Though this idea of testing groups to identify faulty units was not practically implemented at the time, it gained currency and has been the subject of intensive research since. The monograph [12] provides a detailed look into the current state of group testing applications. The fundamental group testing problem is to identify a subset Q of defective items from a set P, by conducting the minimum number of tests on v – subsets of P. A test seeks to identify whether a particular v – subset is defective, as shown by a positive outcome of the test [11]. Group testing algorithms are classified as shown in Figure 1.1.



Probabilistic group testing theory assumes a known probability p of an item being defective, and uses it to guide the isolation process. In *Combinatorial Group Testing (CGT)*, it is often assumed that D is the subset of defective items among S items whereby $p = |D| = |S|$. In *sequential* group testing algorithms, tests are conducted in succession so that the results of previous tests are known to guide the current test. In a *non-adaptive* test, the tests are pre-designed and executed in parallel, without cognition of the result of other tests. In a *multi-stage* algorithm, successive stages of tests utilize informative from previous stages, and tests in a particular stage are executed in parallel. Testing is conducted using a checker or a detector which tests subgroups comprising items from S. A group testing algorithm is *reasonable* if it contains no test whose outcome can be predicted from outcomes of other tests conducted either previously or simultaneously. To minimize the number of tests required to identify the defectives, it is sufficient to consider only reasonable algorithms as otherwise the algorithm would be sub-optimal with respect to this criteria. However, it is not necessary to restrict use to only reasonable algorithms as there many be me practical advantageous to the fault handling process when more general techniques are used. This is especially the case when FPGAs must be

supported on long missions without reducing availability due to the need to execute additional tests.

CGT techniques have been applied to DNA library screening [13] and more recently to hardware fault detection [14]. Efficient algorithms designed for reconfigurable architectures that are capable of solving the fault isolation problem are particularly useful in NASA applications.

1.5. Contributions of this Dissertation

Improving the fault tolerance of reconfigurable devices is a fundamental issue to be considered while using such devices in failure-prone environments. This dissertation develops a strategy for the integration of multiple phases of the fault handling process for reconfigurable devices. While traditional approaches to these problems rely on unique instances of dedicated hardware elements, this dissertation investigates a new technique based on *iterative pairwise comparison* and *functional regeneration*. Under the proposed approach, an initial population consisting of a set of functionally identical (same input-output behavior), yet physically distinct (alternative design or place-and-route realizations) FPGA configurations are produced at design time. The performance of these configurations is evaluated by comparing them in pairs. The result of the pairwise comparisons are then utilized to realize a fault location strategy. The fault location information obtained can then be used to guide the hardware regeneration process. Evolutionary repair techniques inspired by *Genetic Algorithms (GAs)* are used to realize the repair. The methods presented here provide, for the first time, a fault isolation

strategy that works in conjunction with an evolutionary refurbishment mechanism. Significantly, the group testing-based isolation strategy presented here does not require the device to be taken completely offline, or for the resources to be tested exhaustively. This dissertation provides an example of how fault isolation can be achieved while maintaining the system's availability as measured by its goodput.

The competitive evolutionary method presented here leverages information contained in a population of alternatives to enable the refurbishment of faulty configurations. In the context that functional elements are groupings of the underlying physical resources, this research proposes utilization of *Combinatorial Group Testing (CGT)* methods to analyze the expected performance. A comprehensive toolkit for injecting stuck-at faults in FPGA logic for the purpose of evaluating group testing algorithms is developed. This is used to demonstrate the efficiency of CGT techniques in fault isolation. CGT methods are used to develop algorithms for isolating faults using the minimal number of pairings to establish optimality bounds. Further, analytical equation which describe the bounds of the system are derived.

CHAPTER 2: PREVIOUS WORK

Fault tolerance techniques include both Fault Avoidance and Fault Handling approaches. *Fault Avoidance* strives to prevent malfunctions from occurring. This approach increases the probability that the system is functioning 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. If those methods fail, however, *Fault Handling* methodologies can respond to or recover lost functionality. Whereas some fault handling schemes maintain system operation, some fault handling schemes require removing the system offline to recover from a fault, thereby decreasing the system's *availability*. This limited decrease in availability, however, can increase overall reliability.

Hardware failures in FPGA occur variously due to device degradation over age, or due to environmental factors. Ionization, electromigration, hot carrier effects, and other device degenerative effects may cause device faults in the FPGAs used by such applications. In all of the above scenarios, these devices are mandated to operate reliably for long mission durations with limited or absent capabilities for diagnosis/replacement and little onboard capacity for spares. Specifically, when in a space environment, FPGAs are subject to the effects of high-energy particles or radiation. Cosmic rays and high-energy protons can cause malfunctions to occur in systems located on FPGAs. These malfunctions may be a result of Single-Event Latch-ups (SELs) or Single-Event Upsets (SEUs). SEUs are transient in nature, inverting bits stored in memory cells or registers, whereas SELs may

be permanent by inducing high operating current into sensitive devices. While all FPGAs containing memory cells or registers are vulnerable to SEUs, anti-fuse FPGAs are particularly resilient since they do not depend upon SRAM cells to store its configuration. Reconfigurable FPGAs, on the other hand, store its configuration in SRAM cells, which increases the risk to SEUs. Over the years, designers have developed methods for SRAM FPGAs to allow reconfigurability in space applications while mitigated the risk of SEUs.

Radiation-hard SRAM FPGAs have fulfilled the rising demand for FPGAs in space applications. Before their availability, designers of satellites and rovers had no serious alternative to the one-time programmable anti-fuse FPGA. If the inherent fault handling capability of anti-fuse FPGAs was not sufficient, designers were restricted to employing Design-time Redundancy methods. Due to the reconfigurable nature of SRAM FPGAs, radiation-hard SRAM FPGAs have allowed designers to consider other fault handling methods- namely Run-time Fault Handling methods.

2.1. Taxonomy and Nomenclature of FPGA Fault Tolerance Techniques

Figure 1.1 primarily divides Fault Handling approaches into two categories based on its method of implementation [15]. *Architecture-based* fault recovery techniques [16] 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. Concealing the

fault through the underlying fabric of the FPGA is advantageous; users need not know of the occurring hardware faults. Despite making faults transparent to the user, the ability of these methods to tolerate faults is limited in both type and number.

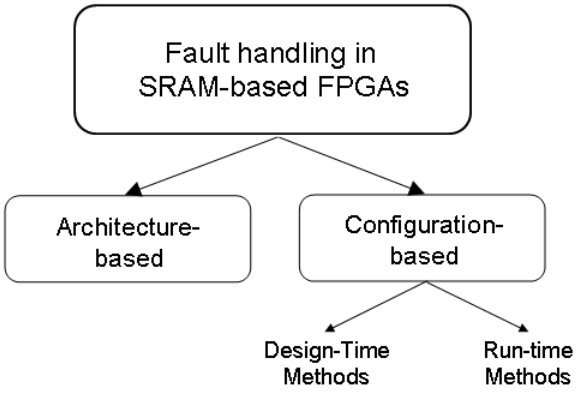


Figure 2.1: Classification of FPGA Fault Handling Methods

Configuration-based methods, however, depend upon the end-user for implementation. These higher-level approaches use the configuration bitstream of the FPGA to integrate redundancy with a user's application. By viewing the FPGA as an array of abstract resources, these techniques may select certain resources for implementation, such as those exhibiting fault-free behavior. Whereas Architecture-based methods typically attempt to address all faults, Configuration-based 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.

Figure 2.1 further separates Configuration-based Fault Handling methods into two categories based on whether an FPGA's configuration will change at run-time. *Design-time Redundancy* methods embed processes into the user's application that mask faults from the system output. These methods are quick to respond and recover from faults due to the explicit redundancy inherent to the processes. This 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 Design-time redundancy method may tolerate one fault and not return to its original redundancy index. This reduced reliability increases the chance of a second fault causing a system malfunction.

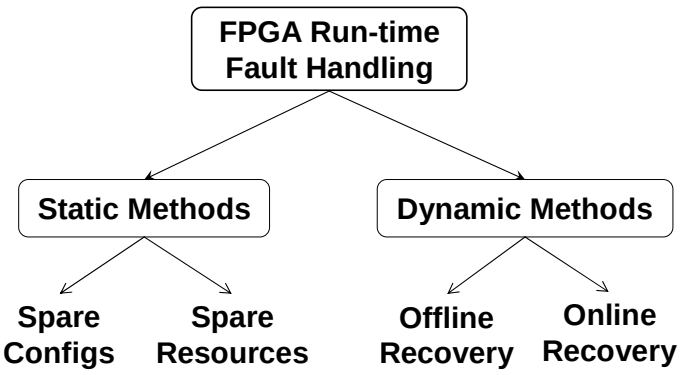


Figure 2.2: Overview of Run-time Fault Handling Methods

Run-time Fault Handling methods strive to increase reliability and *Sustainability* by modifying the configuration of the FPGA to adapt to faults. This allows a system to

remove accumulated SEUs and avoid permanently faulty resources to reclaim its lost functionality. In addition, Run-time schemes can transform faulty resources into constructive components by incorporating stuck-at faulty behavior into the circuit's functionality. 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.

Within Run-time Fault Handling, Figure 2.2 illustrates two classes: *Static* and *Dynamic* methods. Of these, Dynamic fault handling methods are the primary focus of this work. Section 2.2 describes and compares the existing Static Run-time techniques and Section 2.3 addresses the Dynamic Run-time approaches in relation to the concepts used in this work.

2.2. Static Run-time Fault Handling Methods

Static methods may recover from a fault utilizing design-time compiled *spare configurations* or re-mapping and rerouting techniques utilizing *spare resources*. The resource allocation and/or pre-designed configurations are independent of the location an nature of faults detected during run-time. These methods take advantage of the regularity of the FPGA's architecture to implement redundancy structures or for designing alternate configurations. Spare configuration methods must provide sufficient configurations and require storage space overhead for these, whereas spare resource methods must allocate sufficient resources to facilitate a repair.

Spare Configuration-based approaches rely on a population of alternate configurations that each use a different set of logical resources to respond to faults. These can be created either at *design-time*, or at *runtime*, after the fault has occurred. The pre-compiled configuration based technique [17] creates alternative configurations at design time that use different equivalent columns of FPGA resources. In their non-overlapping scheme, which has the least resource overhead, a total of $C(k+m, m) = (k+m)! / (m!k!)$ configurations are required to tolerate faults in m columns, where k is the number of columns in the base configuration. The required design-time effort for this approach is high, as it requires manual modification of the design to fit into column sets. Also, the number of horizontal routes available to the designer is reduced by the resources consumed by the approach. The fitness-based and population-based evolutionary hardware approaches for *Field Programmable Transistor Arrays (FPTAs)* proposed by Keymeulen et al. [18] creates alternative configurations for anticipated faults and at runtime for observed faults respectively. This method provides good resource coverage and passive runtime operation, however system uptime is impacted severely by failure occurrence. Also, additional external computational capacity is required to implement the genetic algorithm that creates the population-based solution at runtime.

Spare Resource-based methods such as the one proposed by lach et al [19] rely on the availability of standby resources of varying granularity to address faults. Lach's deterministic approach provided redundant resources at design time. This approach segments the FPGA into static tiles at design time with a known functionality, some

redundant resources, and a pre-designed alternate configuration. Spare tiles can be selected when needed, but their functionality is predetermined and thus limited. Dutt et al[20] provide an incremental re-routing method for increased flexibility to tolerate fault on-the-fly. In this method, the FPGA is initially routed without any extra interconnects for reconfiguration. The technique relies on node-covering in which reconfiguration is achieved by constructing replacement chains of cells from faulty cells to spare or unused cells. Using a cost-directed depth-first search strategy, they minimize the overheads involved in rerouting interconnects when responding to faults. Other innovative methods to tolerate faults using spare resources include Lakamraju and Tessier's[21] intra-cluster repair. The authors approach fault tolerance for cluster-based FPGA which group multiple LUT/FF pairs together in clusters. Their method that takes advantage of logical redundancy in such clusters by replacing fault LUT inputs and logic resources unused in the original design mapping by defining methods for *LUT Input Exchange* and *Basic Logic Element exchange*. All these re-routing strategies that involve spare resources require the device to be offline, and the support of an external system to complete the re-routing procedure.

2.3. Dynamic Run-time Fault Handling Methods

Dynamic methods aim to allocate spare resources or otherwise modify the configuration during run-time after detecting the fault. Whereas these approaches offer the flexibility of adapting to emergent fault scenarios, additional time is necessary to generate appropriate configurations to repair 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.

2.3.1. Offline Recovery Methods

2.3.1.1. Genetic Algorithm Repair

Genetic Algorithms (GA) are inspired by evolutionary behavior of biological systems to produce solutions to computational problems [Mitchell 1998]. 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 repair them upon a fault [Keymuelen et al., 2000]. Vigander [2001] proposes the use of GAs to repair faulty FPGA circuits. As a proof of concept, Vigander implements extrinsic evolution, utilizing a simulated feed-forward model of the FPGA device with genetic chromosomes representing logic and interconnect configurations.

The evolution process begins with initializing a population of candidate solutions. These initial solutions contain different physical implementations of the same functional circuit. In the midst of a fault, the performance of each configuration is evaluated, revealing which configurations 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. Those

previous configurations having a higher performance rating are more likely to be selected and combine with other configurations by the *Crossover* genetic operator. Additionally, the *Mutation* genetic operator injects random variations in the newly created candidate solutions. Vigander also makes use of a *Cell Swap* operator that allows the functionality and connectivity of a faulty cell to swap with a spare cell. The GA evaluates the newly created solutions and replaces poorer performers in the old population with better performers in the current population to create a new generation of candidate solutions. This evolutionary process repeats, stopping when an optimal solution is discovered or after a specific number of generations.

Garvie et al.'s method [22] tolerates permanent faults using jiggling. Jiggling involves repairing a faulty configuration by using an evolutionary algorithm that uses the other two healthy modules and fitness feedback from the TMR voting element. Vigander's, Garvie's and other *n*-plex *spatial voting* approaches [23] deliver real-time fault resolution, but increase power consumption and area requirement *n*-fold during fault-free operation. Previously, these evolutionary approaches have only been simulated using hypothetical device models. They did not attempt application to *Commercial Off The Shelf (COTS)* FPGAs and development tools.

2.3.1.2. Augmented Genetic Algorithm Repair

To decrease the amount of time required to generate a repair, Oreifej et al. [24] augment Vigander's Genetic Algorithm fault handling concept with a Combinatorial Group Testing (CGT) fault isolation technique. Group Testing partitions suspect resources into

groups and coordinates those groups into a minimal number of tests to isolate the faulty resource. If a group manifests a fault within one of these tests, then the group is known to contain the faulty resource and thus the resources within the group are classified as suspect. In a deterministic manner, the suspect resources are partitioned into iteratively smaller groups and tested until the faulty resource is isolated.

A population within a GA contains various configurations, each of which categorizes the FPGA resources into two groups: utilized and unutilized resources. CGT evaluates each configuration for correct functionality. If a configuration manifests a faulty output, then the resources used by that configuration are considered suspect. Since the various configurations within the population form groups that overlap particular resources, CGT tests multiple configurations and accumulates the number of times each resource is considered suspect through a History Matrix. Configurations are rotated through the FPGA and tested until one element becomes the maximum value within the matrix, isolating the fault to one resource. The GA, in turn, uses the fault location information to avoid faulty resources while evolving a repaired configuration.

2.3.1.3. Incremental Rerouting Algorithms

The Node-Covering method discussed in Section 2.2 avoids a fault by rerouting 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 run-

time instead of design-time. Run-time reserved wire segments allow the method to utilize unused resources in addition to the spares allocated during design-time.

Emmert and Bhatia [25] present a similar Incremental Rerouting approach that does not require design-time allocated spare resources. The fault recovery method assumes an FPGA to contain resources not utilized by the application, thus exploiting unused fault-free resources to replace faulty resources. Upon detecting and diagnosing a logic or interconnection fault by some other detection 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.

Since faulty cells may not be adjacent to a spare resource, a string of cells is created logically, starting with the faulty cell and ending with the logic cell adjacent to the spare resource. To avoid the fault, the string of cells 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 cells within the row 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 cells may extend into multiple rows to reach a spare cell.

Re-placing cells requires the wire segments of the moving logic cells to be rerouted. The configuration memory of the FPGA is read to determine which nets are affected by the re-placed logic cells. All faulty nets and those that solely connect the moved logic cells are ripped-up [25] while those that connect other unmoved logic cells 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 expand its scope to encompass wider routing paths for unroutable nets.

2.3.2. Online Recovery Methods

2.3.2.1. TMR with Single-Module Repair

Since Triple Modular Redundancy (TMR) performs the majority vote of three modules, the voted output remains correct even if a single module is defective. Exploiting this concept allows a system to remain online with two viable modules while a defective module undergoes repair. Methods presented by Ross and Hall [26], Shanthi et al. [27], and Garvie and Thompson [22] repair the defective module through genetic algorithms.

At design-time, Ross and Hall [26] produce a population of diverse configurations for implementation. At run-time, three of these configurations are implemented into the circuit and monitored for discrepancies. Agreeing outputs indicate that the modules are functioning correctly whereas discrepancies indicate defective resources utilized by one of the configurations. A mutation genetic operator is applied to defective modules and the fitness of the new individual is evaluated. The process repeats until the fault is occluded.

In addition to the strategy above, Shanthi et al. [27] utilize a deterministic approach in identifying faulty resources. By monitoring the resources within each configuration, resources utilized by viable modules gain confidence whereas resources utilized by faulty modules gain suspicion. This information allows fault handling by implementing configurations not using defective resources. Additionally, differing configurations can be rotated to reveal dormant faults in unused resources.

Instead of selecting from a diverse population, Garvie and Thompson [22] implement three identical modules. The commonality between configurations permits a *Lazy Scrubbing* technique, which 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 provides a minimal genetic algorithm, which produces one genetically modified offspring from one parent and chooses the most fit between the two. To mitigate the possibility for 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.

2.3.2.2. Online Built-in Self Test

Emmert et al. [28] 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 [28], this method partitions the FPGA into an 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 occurs 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 its 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 programmable logic blocks (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 have equal probability to occur within used resources with unused resources, Roving STARs provides 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 run-time. 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. In addition to pausing the system clock, the TREC implements an Adaptable System Clock where the clock speed is adjusted to account for timing delays arising from new configurations that adapt to faults.

Roving STARs supports 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 reroute incrementally 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, however, may continue roving the FPGA searching for faults in its respective routing resources and PLBs. 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.

As previously stated, testing and diagnosis occurs within a STAR. 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.

2.3.2.3. Consensus-based Evaluation of Competing Configurations

Whereas previous Online Genetic Algorithm-based methods utilize an N-MR voting element, the *Competitive Runtime Reconfiguration (CRR)* approach presented here 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 all *Left-Half* configurations to one region and *Right-Half* configurations to the other region. Together, these configurations comprise the population of competing alternatives. The detection method realizes a traditional *Concurrent Error Detection (CED)* arrangement that allocates mutually exclusive resources for each individual, which detects any single resource fault. The comparison can result in either a discrepancy or a match between left-half and right-half configuration outputs, when resource faults are *articulated* by the configurations that utilize the faulty resources. Such discrepancies indicate the presence of FPGA resource faults in either the resources used to constitute the combinational logic module or a pipeline stage consisting of combinational logic.

2.4. Fault Detection and Location using Exhaustive Testing Techniques

Several approaches to GA-based fault handling in FPGAs utilize exhaustive testing for fault isolation and offline regeneration mechanisms. In addition to TMR, Table 2.1 also lists characteristics of fault-handling schemes that consider reconfigurability. TMR, Vigander's, and other n-plex spatial voting approaches deliver real-time fault resolution, but increase power consumption *n*-fold during fault-free operation. STARS [29] is an example of a resource-oriented diagnostic method that performs *Built-in Self-Tests (BISTs)* on sub-sections of the FPGA. STARS extends the concept of using exhaustive testing by exploiting reconfigurability to occlude faults in the circuits. Under this paradigm, the test area roves across all FPGA resources. Portions of the FPGA are continually taken offline in succession for testing while the functionality is moved to a new location within the reprogrammable fabric. The device, however, remains operational and hence online. One limitation is that detection latency can be large since tests must sweep through all intervening resources before a fault is detected. Potential throughput unavailability due to diagnostic reconfigurations when no faults have yet occurred is also a consideration. However, STARS is a successful example of a method that uses exhaustive online testing to realize regeneration. Methods proposed by Lohn [1] and Lach [19] either rely on offline regeneration supported by exhaustive functional testing, or pre-determined spares defined at design-time.

Table 2.1: Characteristics of Related FPGA Fault-Handling Schemes

Approach	Fault Handling Method	Fault Detection		Resource Coverage			Fault Isolation
		Latency	Distinguish Transients	Logic	Inter-connect	Comparator	Granularity
TMR	Spatial voting	Negligible	No	Yes	Yes	No	Voting element
Vigander [30]	Spatial voting & offline evolutionary regeneration	Negligible	No	Yes	No	No	Voting element
Lohn et al. [1]	Offline evolutionary regeneration	Negligible	No	Yes	Yes	No	Unnecessary
Lach et al. [19]	Static-capability tile reconfiguration	Relies on independent fault detection mechanism					
STARs [29]	Online BIST	Up to 8.5M erroneous outputs	Test pattern transients	Yes	Yes	No	LUT function
Keymeulen[18]	Population-based fault insensitive design	Design-time prevention emphasis	No	Yes	Yes	No	Not addressed at runtime
CRR	*Competitive runtime input fitness evaluation and evolutionary regeneration*	*Negligible*	*Transients are attenuated automatically*	*Yes*	*Yes*	*Yes*	*Unnecessary, but can isolate functional components*

Of the methods in Table I, only Keymeulen, Stoica, and Zebulum [18] investigate the possibility of using a population-based approach to desensitize circuits to faults. They develop evolutionary techniques so that a circuit is initially designed to remain functional even in presence of various faults. Their population-based fault tolerant design method evolves diverse circuits and then selects the most fault-insensitive individual. In this paper we propose a system that achieves improved fault tolerance by using a runtime adaptive algorithm that emphasizes the utilization of responses observed during the actual operation of the device. While their population-based fault tolerance approach provides passive runtime tolerance, CRR is dynamic and actively improves the fault tolerance of the system according to environmental demands.

2.5. Forming a Robust Consensus from Diversity

An evolutionary process that uses absolute fitness measures and exhaustive tests may not be able to provide adaptive fault tolerance. Layzell and Thompson [31] dealt with these aspects in terms of *Populational Fault Tolerance (PFT)* as an inherent quality of evolvable hardware. Under PFT, the creation of the best-fit individual proceeds by incrementally incorporating additional elements into partially-correct prototypes to adapt to faults. They speculate that PFT is less likely to occur for online evolution in dynamic environments. Nonetheless, evaluation becomes focused on the precise regions of relevance within the search space during the execution of online processes. This provides a powerful motivation to explore CBE.

Yao and Liu [32] emphasize that in evolutionary systems the population contains more information than any one individual. They demonstrate the utility of information contained within the population using case studies from the domains of artificial neural networks and rule based systems. In both cases, the final collection of individuals outperforms any single individual. The work in [33] further extends this concept by presenting four methods for combining the different individuals in the final population to generate system outputs. They provide similar results for three data sets, namely the Australian credit card assessment problem, the heart disease problem, and the diabetes problem. While the authors devise a method to utilize the information contained in the population to improve the final solution, they fail to use the information in the population to improve the learning and optimization process itself. The proposed CBE approach indicates that refurbishment problems can benefit from population information.

More recently, in [34] the authors describe using fitness sharing and negative correlation to create a diverse population of solutions. A combined solution is then obtained using a gating algorithm that ensures the best response to the observed stimuli. In evolvable hardware, it may not always be possible to combine solutions without additional physical resources that may also be fault-prone. In our approach, all individuals in the population are recognized as possible solutions, with the best emerging candidate being selected based on its runtime performance record. The authors also claim that applying the described techniques to evolvable hardware applications should be straightforward, but do not provide examples. They state the absence of an optimal way of predicting the future performance of evolved circuits in unforeseen environments as an impediment. Chapter 3 details how an adaptive system can keep track of the relative performances of individuals and implicitly build a consensus.

2.6. Improving Reliability using Autonomous Group Testing

In state-of-the-art Xilinx SRAM-based FPGAs, the device configuration can be modified without interrupting the normal operation of the device. For space applications, it is typical to perform such configuration scrubbing periodically to repair any configuration errors due to Single Event Upsets (SEUs) [35]. The Xilinx TMR tool software [36] can be used to not only triplicate the user's design, but also insert logic to repair transient user memory errors and upsets due to SEUs. TMR can be combined with the scrubbing method to have a reliable system while preventing soft errors. However, configuration scrubbing only refreshes a single complete configuration and therefore cannot be used to

address permanent faults [37]. While an *n*-modular redundancy scheme such as TMR ensures validated correct output, the proposed AGT-based technique can minimize the risk of having two faulty modules. The comparators of the Xilinx TMR tools can be used to detect the discrepancy among the redundant modules. Discrepancies reported by the comparators can be used to target all resources used by a faulty module. Once the faulty module is identified, the GT-based algorithm can localize the fault to a logic slice. Autonomous group testing aims to avoid system failure by providing methods to isolate permanent faults and maintain a healthy population of configurations for each redundant module.

CHAPTER 3: COMPETITIVE RUNTIME RECONFIGURATION FAULT HANDLING PARADIGM

While the fault repair capability of *Evolvable Hardware (EH)* approaches have been previously demonstrated, further improvements to fault handling capability can be achieved by exploiting population diversity during all phases of the fault handling process. In existing fault-handling methods for reconfigurable hardware, fault-tolerance is evolved at design time, or achieved at repair-time using evolution after taking a detected failed unit offline. In both cases, GAs provided a population-based optimization algorithm with the objective of producing a single best-fit individual as the final product. They rely on a pre-determined static fitness function that does not consider an individual's utility relative to the rest of the population. The evaluation mechanisms used in previous approaches depend on the application of exhaustive test vectors to determine the individual with the best response to all possible inputs.

However, given that partially complete repairs are often the best attainable [1], [30], other individuals may outperform the best-fit individual over the range of inputs of interest. In particular, there is no guarantee that the individual with the best absolute fitness measure for an exhaustive set of test inputs will correspond to the individual within the population that has the best performance among individuals under the subset of inputs actually applied. Thus, exhaustive evaluation of regenerated alternatives is computationally expensive, yet not necessarily indicative of the optimal performing individual among a

population of partially correct repairs. Hence, two innovations are developed herein for sustainable EH regeneration:

1. Elimination of additional test vectors, and
2. Temporal Assessment based on aging and outlier identification

In CRR, an initial population of functionally identical (same input-output behavior), yet physically distinct (alternative design or place-and-route realization) FPGA configurations is produced at design time. During runtime, these individuals compete for selection based on discrepancy favoring fault-free behavior. Discrepant behavior, where the outputs of two competing individuals do not agree on a bit-by-bit basis, is used as the basis for the performance evaluation process. Any operationally visible fault will decrease the fitness of just those configurations that use it. Over a period of time, as the result of successive comparisons, a consensus emerges from the population regarding the relative fitness of all individuals. This allows the classification of configurations into ranges of relative reliabilities based on their observed performance during online operation.

3.1. Detecting Faults using a Population of Alternatives

In order to provide fault coverage for the voting element, a distributed discrepancy detector circuit may be used, as described in Section 4.3. Each individual in the population has an instance of one of the two complementary halves of the discrepancy detector circuit. When two competing *L* and *R* half-configurations are loaded on the

FPGA, the discrepancy detector circuit is completed. The design of the discrepancy detector accounts for the possibility of error in either, or both of the complementary halves of the detector. Such an error would reflect on the performance of the half-configurations that instantiated the detector hence degrading any preference for selection of those individuals as described below.

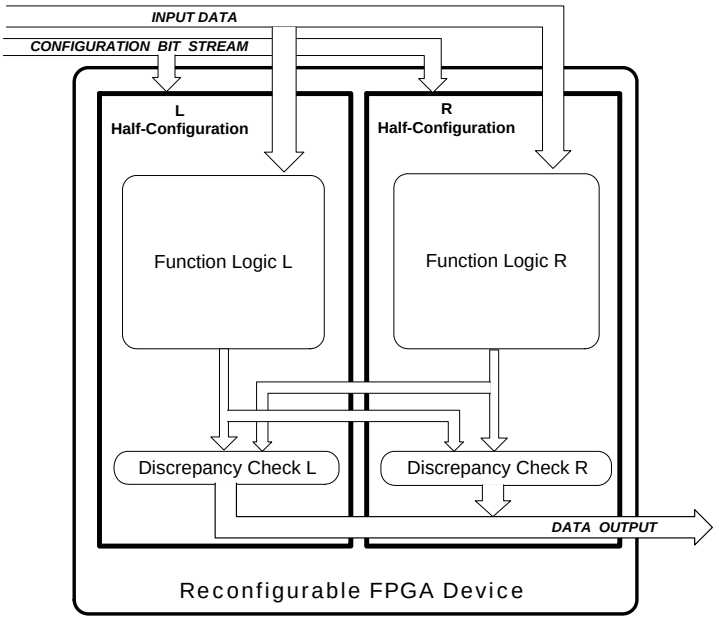


Figure 3.1: Physical Arrangement with Two Competing Configurations

3.2. Assessing Individual Fitness and Managing Fitness States

Instead of using an absolute fitness function with exhaustive testing, outlier identification can be achieved using techniques such as the *hat matrix* [38], $\boldsymbol{H}$, where the diagonal elements $\boldsymbol{H_{ii}}$ are used to identify the threshold to isolate faulty individuals as outliers. The threshold value is determined by an analysis of the diagonal elements $\boldsymbol{H_{ii}}$ of the hat matrix generated from population statistics accumulated over an evaluation window. The

relative reliability of an individual is indicated by its instantaneous fitness state. Through run-time competition, and the concomitant fitness state assignment, a fault becomes occluded from the visibility of subsequent FPGA operations.

Health state transitions are managed by the procedural flow for the CRR algorithm as depicted in Figure 3.2. After *Initialization*, the *Selection* of the *L* and *R* half-configurations occurs. The selected individuals are then loaded onto the FPGA. Next, the *Detection* process is conducted when the normal data processing inputs are applied to the FPGA. The *DV*s of the competing half-configurations are updated based on whether or not their outputs are discrepant. The central *Primary Loop* representing discrepancy-free behavior can repeat without reselection as long as there is no discrepancy. However, even in the absence of any observed discrepancies, one or more of the competing individuals may be replaced to hasten regeneration in the presence of *Under Repair* individuals. As described later, the *Replacement Rate*, R_X, determines the frequency with which such discrepancy-free individuals are replaced to allow rotation of other individuals from the *Dormant* pool. The system availability can be increased by using a low value of R_X.

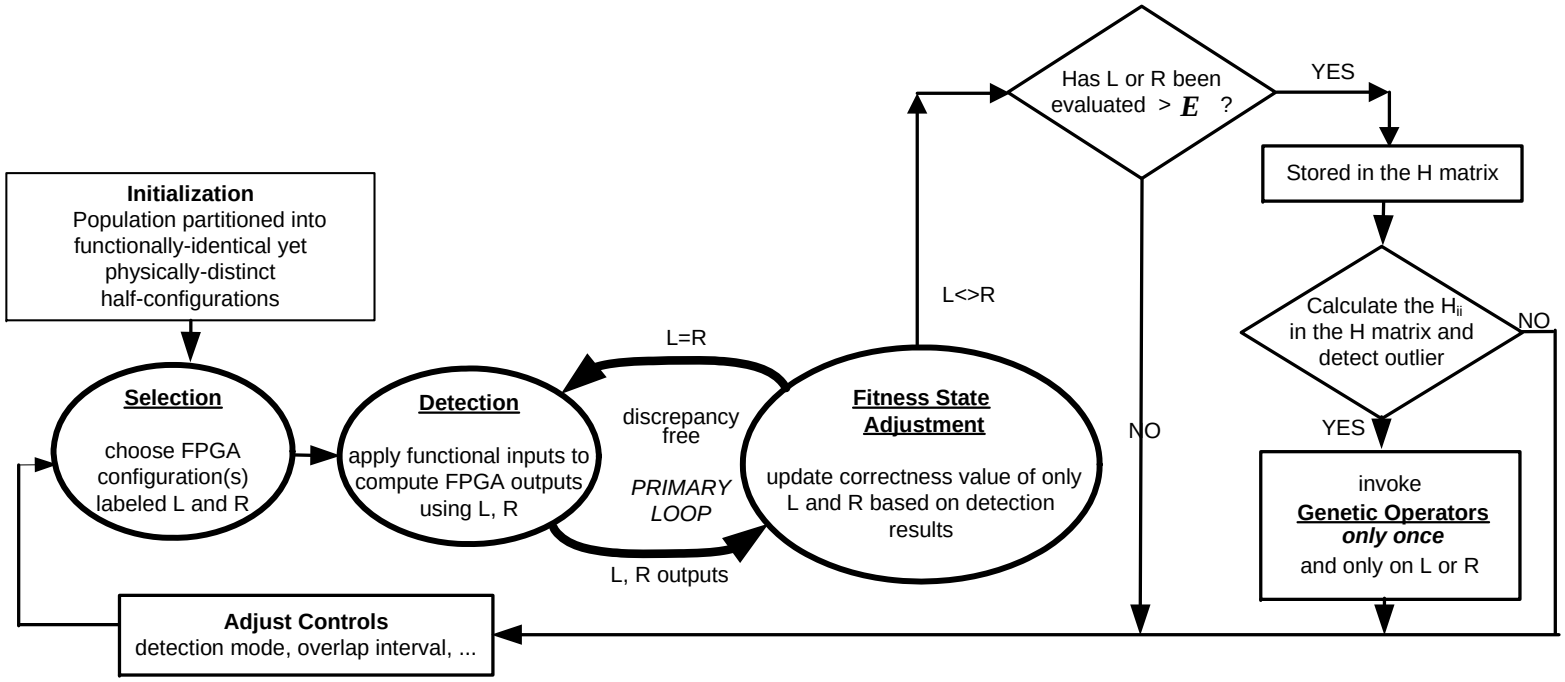


Figure 3.2: Procedural Flow in the CRR Technique

The *Fitness State Adjustment* process will be used to validate and update the state of the individual after E evaluations. Otherwise reselection will occur, without updating the fitness state of the individual being replaced. For Under Repair individuals, if the value of the corresponding $\boldsymbol{H_{ii}}$ element is greater than the threshold value then *Genetic Operators* are invoked only once without attempting to achieve complete refurbishment. The modified configuration is then immediately returned to the pool of competing configurations and the process resumes starting with the *Selection* phase.

3.3. Strategic Prioritization of Individuals for Assessment and Refurbishment

The Selection and Detection processes are shown in Figure 3.3. During the selection process, *Pristine*, *Suspect*, and then *Refurbished* individuals are preferred in that order for one half-configuration. The selection of individuals based on the relative fitness ensures the lowest possible probability of two half-configurations agreeing by producing the same incorrect outputs. The other half-configuration is selected based on a stochastic process determined by the *Re-introduction Rate* (λ_R). In particular, *Under Repair* individuals are selected as one of the competing half-configurations on average at a rate equal to λ_R. Thus, a genetically-modified *Under Repair* configuration is re-introduced at a controlled rate into the operational throughput flow. They act as a new competitor to potentially exhibit fault-free behavior against the larger pool of configurations. An additional innovation is that λ_R can also be adapted to encourage *Mean-Time-To-Repair* (*MTTR) << Mean-Time-Between-Failures (MTBF)* to refurbish the population at a rate

not less than new failures are occurring. Maintaining this inequality realizes sustainable fault-handling under fully autonomous operation.

The Detection process is presented in the lower right corner of Figure 3.3. If a discrepancy is observed as a result of output comparison, the FPGA is reconfigured with a different pair of competing configurations and the output of the device need not be propagated to allow recalculation. The evaluation mechanisms used in previous approaches depend on exhaustive test vectors. They also utilize a pre-determined fitness evaluation scheme to determine the individual with the best response to all possible inputs. Other partially repaired individuals may outperform the best-fit individual for the runtime input vectors. CRR overcomes these issues by using the runtime inputs as the test vector, and the output of the discrepancy detector to detect faults and provide information for the subsequent isolation of outliers as described in Section 4.2. Also, the partially correct outputs generated by competing fault-affected individuals can improve availability as opposed to keeping a device completely offline while a perfect solution is being obtained.

In order to isolate and detect faulty individuals in a timely manner, all the individuals in the population should have an equally likely probability of being selected as the Active individuals with a suitable interval between successive selections. The *Replacement Rate*, R_X, is used to monitor this rotation of individuals onboard the FPGA device, including the individuals not Under Repair.

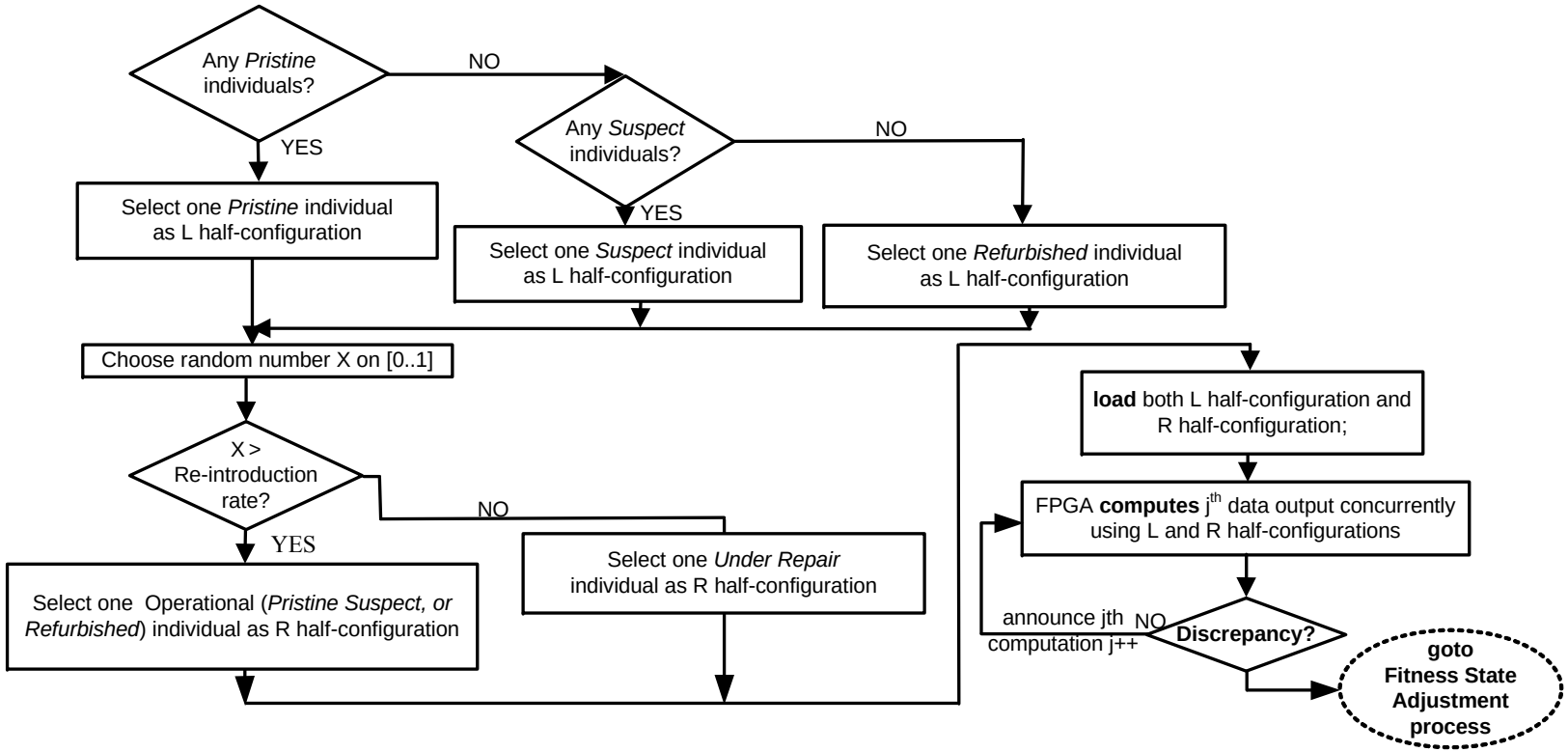


Figure 3.3: Selection and Detection in the CRR Paradigm

3.4. Determination of Evaluation Window

CRR uses runtime-inputs for individual performance evaluation rather than exhaustive testing with a predefined set of test vectors. Nonetheless, pseudo-exhaustive testing on an individual basis provides adequate test coverage. While the range and sequence of the online inputs may not be known at design-time, a probabilistic model is useful to estimate the expected number of evaluations required to encounter a sufficient range of values with high probability. The Evaluation Window, E, is selected accordingly. It regulates the update frequency of each individual's relative fitness based on DV_i values during the interval.

The characteristics of the circuit under repair influence the determination of E as illustrated for an unsigned integer multiplier. Let the circuit input width, W, denote the total number of operand bits to the multiplier. In the case of a 3-bit×3-bit multiplier, W = 6 and the total number of distinct input combinations is $2^W = 64$. Thus in the case of the 3-bit×3-bit multiplier, an exhaustive set of inputs would consist of all 64 possible combinations. The problem of determining the number of random inputs needed to facilitate all possible inputs appearing at least once is similar to the *coupon collector problem* [39]. In the coupon collector problem, the expected number of coupons to be collected before at least one each of D total coupons are collected is given by the simplified expression, $D \times H_D$, where H_D is the D^{th} harmonic sum. However, for the exhaustive test modeling problem at hand, the number of random inputs required to facilitate the appearance of all possible inputs with varying confidence factors needs to be

derived. This problem can be modeled as a game involving selection of balls from a set of 64 differently colored balls. A single ball is selected in each drawing, with replacement. In other words, what is the probability that, after D drawings, at least one ball of each of the 64 colors appeared at least once? Clearly, for $D < 64$, the probability is zero, and for $D = 64$ is 2.54×10^{-116} which is highly improbable.

To solve this problem, consider the case where all balls are of one color. After D drawings, we have $\binom{1}{1}x_1 = 1^D$ where x_1 is the number of feasible sample events, so $x_1 = 1^D$. Now, consider the case when $D \geq 64$. In general, a *K-color experiment* can be described as a sum of experiments involving smaller numbers of colors for any constant value of D:

$$\binom{K}{K}x_K + \binom{K}{k-1}x_{K-1} + \ldots\ldots\ldots + \binom{K}{2}x_2 + \binom{K}{1}x_1 = K^D \tag{4.1}$$

or,

$$\sum_{m=1}^{K}\binom{K}{m}x_m = K^D \tag{4.2}$$

Since the numerical value of K^D in Equation (4.2) can be excessively large, it may not be possible to represent it using an *unsigned long* variable, the widest variable in a 32-bit system, since for example $64^{64} > 2^{32} - 1$. Therefore, an alternate representation can consider x_K as a sample event in which all K colored balls appear at least once with a

probability P_K. D is the number of drawings, and K^D is the total number of possible permutations, yielding:

$$P_K = x_K / K^D \tag{4.3}$$

Now, by dividing Equation (4.1) and Equation (4.2) by K^D, we obtain respectively,

$$\binom{K}{K}P_K + \binom{K}{k-1}P_{K-1} + \ldots\ldots\ldots + \binom{K}{2}P_2 + \binom{K}{1}P_1 = 1 \tag{4.4}$$

and,

$$\sum_{m=1}^{k}\binom{k}{m}P_m = 1 \tag{4.5}$$

So when K=1,

$$\binom{1}{1}P_1 = x_1 / 1^D = 1 \Rightarrow P_1 = 1 \; ; \; x_1 = 1^D \tag{4.6}$$

When K=2,

$$\binom{2}{2}P_2 + \binom{2}{1}P_1 = P_2 + \binom{2}{1}P_1(1^D / 2^D) = 1 \Rightarrow P_2 = 1 - \binom{2}{1}P_1(1/2)^D \tag{4.7}$$

Therefore, in general:

$$P_K + \binom{K}{K-1}P_{K-1}\left(\frac{(K-1)^D}{K^D}\right) + \ldots. + \binom{K}{1}P_1\left(\frac{(K-1)^D}{K^D}\right) = 1 \tag{4.8}$$

$$\Rightarrow P_K = 1 - \binom{K}{K-1}P_{K-1}((K-1)/K)^D - \ldots\ldots - \binom{K}{1}P_1((K-1)/K)^D \tag{4.9}$$

Equation (4.9) yields P_K recursively without the computational burden of calculating K^D as $\left((K-1)/K\right) < 1$ for all K.

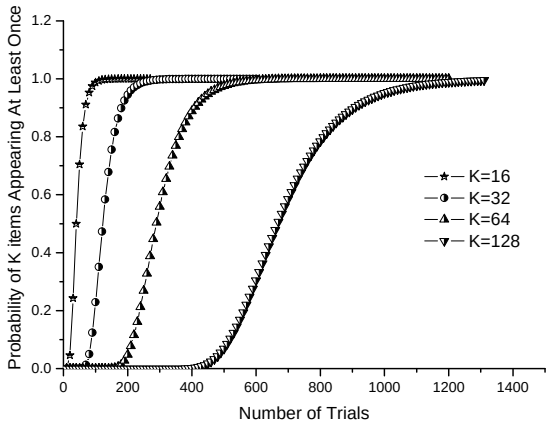


Figure 3.4: Effect of Sample Size on Test Coverage

As shown in Figure 3.4, when K=16 colors and D=100 drawings, the probability P_{16} of all 16 colors appearing is ≈ 100%. Similarly, 250 trials for 32 colors are sufficient given equi-probable inputs. Table 3.1 shows the result for the case when K=64, which applies to the 3-bit×3-bit multiplier. In order to achieve comprehensive coverage with a certainty of 97.59%, approximately 500 evaluations are sufficient. A certainty of 99.50% implies an evaluation window of width E=600 which was adopted for the fault isolation experiments in Section 3.6. Thus, in the case of a 3-bit×3-bit multiplier design, if 1-out-of-64 inputs articulate a fault in a single individual C_i, and all the input combinations are equally likely to appear, then the expected discrepancy value after E = 600 evaluations is:

$$DV_i = \left(\frac{1}{64}\right) \times E = 9.375 \tag{4.10}$$

Table 3.1: Probability of all 64 Inputs Appearing At Least Once given D Evaluations

	D=350	D=400	D=450	D=500	D=550	D=600	D=650
P_{64}	76.96%	88.84%	94.77%	97.59%	99.00%	99.50%	99.77%

3.5. Identifying Outliers using the Sliding Window Technique

From a statistical perspective, the *residuals*, expressed as the difference between the expected fault-free behavior and the observed circuit response, of the faulty individuals are significantly larger than the fault-free individuals when using the *Least Squares (LS)* method [38]. However, the LS method is most effective when exactly one outlying element is expected. In the case of multiple outliers being detected in one Sliding Window, the mean and the standard deviation alone may not aid in detecting the multiple outliers leading to a loss in isolation capacity. Also, to increase the confidence with which outliers are isolated, we increase threshold from one standard deviation from the mean to a value of 2.5σ. Under these circumstances, a simple method such as the LS method is not directly applicable.

Another class of outlier diagnostics is based on the principle of detecting the outlier by the LS projection matrix ***H***. This matrix is well known under the name *hat matrix*, because it is denoted by a hat on the column vector $y = (y_1, \ldots, y_n)^t$ such that $\hat{y}=H*y$ and $\hat{y}$ is the LS prediction for y. The hat matrix ***H*** is defined as follows: consider there are p explanatory variables and one response variable which will have n observations. The n-by-1 vector of responses is denoted by $y=(y_1, \ldots, y_n)^t$. The linear model states that

$y=X\times\theta+e$, where θ is the vector of unknown parameters, e is the error vector and $\boldsymbol{X}$ is the n-by-p matrix:

$$\mathbf{X}=\begin{bmatrix} \mathbf{x_{11}} & \mathbf{x_{12}} & \cdots & \mathbf{x_{1p}} \\ \mathbf{x_{21}} & \mathbf{x_{22}} & \cdots & \mathbf{x_{2p}} \\ \vdots & \vdots & & \vdots \\ \vdots & \vdots & & \vdots \\ \mathbf{x_{n1}} & \mathbf{x_{n2}} & \cdots & \mathbf{x_{np}} \end{bmatrix} \tag{4.11}$$

Then, the H matrix is composed from $\boldsymbol{X}$ as follows:

$$\mathbf{H}=\mathbf{X}(\mathbf{X^t X})^{-1}\mathbf{X^t} \tag{4.12}$$

The diagonal elements of $\boldsymbol{H}$ have a direct interpretation as the effect exerted by the i^{th} observation on the expectation of response variable because they equal $\partial \hat{\mathbf{y}}_\mathbf{i}/\partial \mathbf{y_i}$. The average value of the diagonal element $\boldsymbol{H_{ii}}$ is p/n and it follows that $0 \le H_{ii} \le 1$ for all i.

In the CRR approach, the *DV* of each individual can be viewed as one observation or one explanatory variable, and the Observation Interval can be set as the size of the entire population. Fortunately, since the $\boldsymbol{X}$ matrix consists of only one column in our application, the result of the $\boldsymbol{X^tX}$ product is a single-element vector matrix, and its inverse can be computed using a straightforward one-step computation. In general, the computation complexity of the $\boldsymbol{H}$ matrix approach is $2n^2+1$.

The recommended threshold for the identification of outliers is $\boldsymbol{H_{ii}} > 2p/n$ and a stricter cut-off value 3p/n has been used in previous works [40] [41]. For an analysis of the CRR problem for fault isolation, setting p = 1 and n = 20 corresponds to one faulty individual among a population of 20. For example, a cut-off value of $10\times\frac{p}{n} = \frac{10}{20} = 0.5$ can be used in conjunction with a larger Sliding window width of 15 to ensure consistent outlier identification with 100% certainty.

3.6. Outlier Detection and Fault Isolation Performance with Runtime Inputs

Experimental results regarding the effect of the outlier detection parameters are illustrated in Figure 3.5 through Figure 13. Each has been generated using a simulator written in the C++ programming language which utilizes an equi-probable selection of individuals. In the data reported for experiments, the inputs causing the first discrepancy are applied once after each pair of faulty configurations is replaced to assess damage definitively under the single-fault model.

To further illustrate how the DVs are mapped to the $\boldsymbol{H_{ii}}$ values, Figure 3.5 through Figure 13 are presented in pairs that show results from the same experiment. The first Figure in each pair shows the observed DVs and the subsequent Figure shows the $\boldsymbol{H_{ii}}$ values calculated using this data. Figure 3.5 and Figure 3.6 depict the identification of outlying individuals in the population that has a 10-out-of-64 fault impact caused by a single fault. For example, Figure 3.5 shows the DVs observed over 50 individual evaluations, where each evaluation occurs after the particular individual has completed E=600 computations

as an Active configuration on the simulated FPGA. This corresponds to one individual completing a number of computations equal to E. From Figure 3.5, an outlier is identified when ten individuals have completed E iterations.

A sliding window width of 15 was used in this experiment. Based on analysis of the $\boldsymbol{H_{ii}}$ values, and an outlier cut-off value of 0.5, the outlying individual is identified without statistically significant error. As shown in Figure 3.5, outliers can be identified with a regular periodicity. Figure 3.6 shows the plot of $\boldsymbol{H_{ii}}$ values for a subset of evaluations corresponding to the identification of the first outlier in Figure 3.5. Figure 3.6 also shows that the outlier in the population exhibits $\boldsymbol{H_{ii}} \approx 0.94$ which is over an order of magnitude larger than $\boldsymbol{H_{ii}} \approx 0.02$ of the other competitors. Also, the $\boldsymbol{H_{ii}}$ values of the non-outlying elements conform to a very narrow window of values, clearly demonstrating that the penalty for discrepant observations are amortized among the non-defective members of the population. In Figure 3.5, it can be clearly seen that the first outlier is identified after $11 \times E = 6600$ computations. This period, after which the outlier is detected can be lowered by reducing the sliding window. By choosing a lower value for the sliding window, outlier identification will take place at an increased frequency as shown in subsequent experiments.

In Figure 3.5 and Figure 3.6, individual performance was measured using a simple Winner-Takes-All scheme, where the only information available from the discrepancy detection is bit-wise output equality. A different discrepancy detection mechanism could provide information such as the Hamming distance of the observed output of an individual from the desired output.

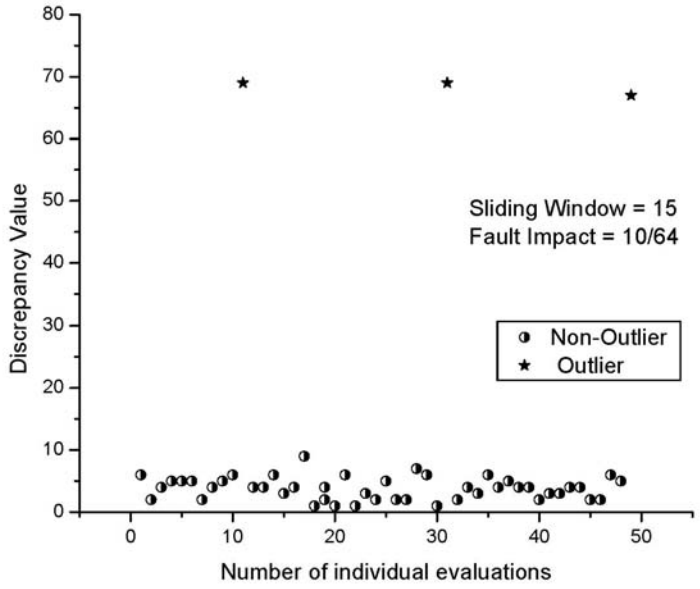


Figure 3.5: Discrepancy Values Observed
when One Individual has a 10-out-of-64 Fault Impact

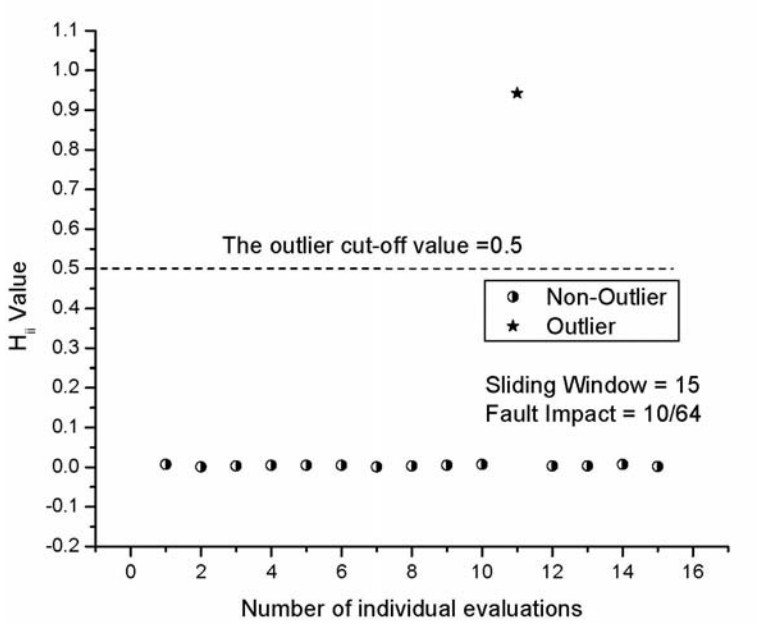


Figure 3.6: Plot of $\boldsymbol{H_{ii}}$ Showing Outlier Identification

The use of Hamming distance information leads to outliers having a higher discrepancy value, as shown in Figure 3.7, when compared to Figure 3.5. As in the previous experiment, a 10-out-of-64 fault impact is considered, with a sliding window width of 15. The higher *DV* of approximately 140 can be accounted for by the fact that the observed Hamming distance between the observed discrepant output and the desired ideal can be greater than 1. This is opposed to the previous case, where the presence of a discrepancy increases the *DV* of the corresponding individuals by one yielding *DV*≈70. The outlier threshold remains the same, nonetheless, since the hat matrix operates on normalized information. Figure 3.7 and Figure 3.8 show plots of the Discrepancy Value and the ***H*** values when the Hamming distance is used to quantify divergence.

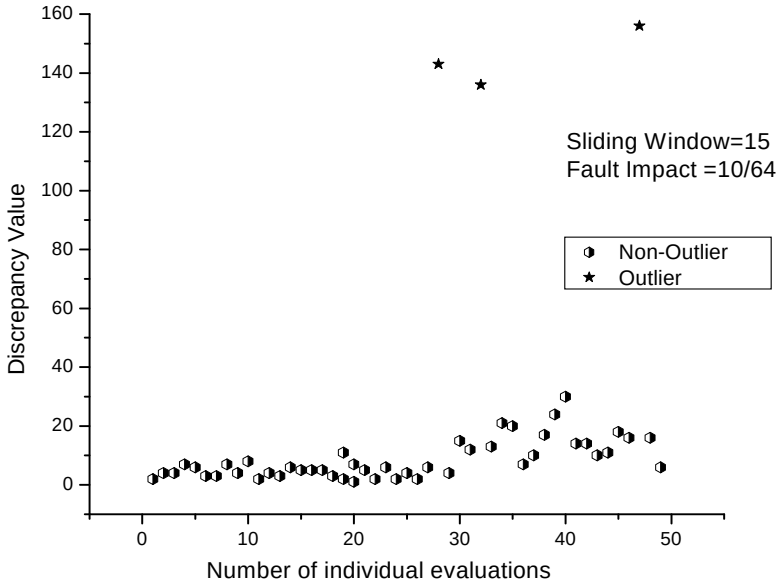


Figure 3.7: Discrepancy Values Observed When Hamming Distance is Used

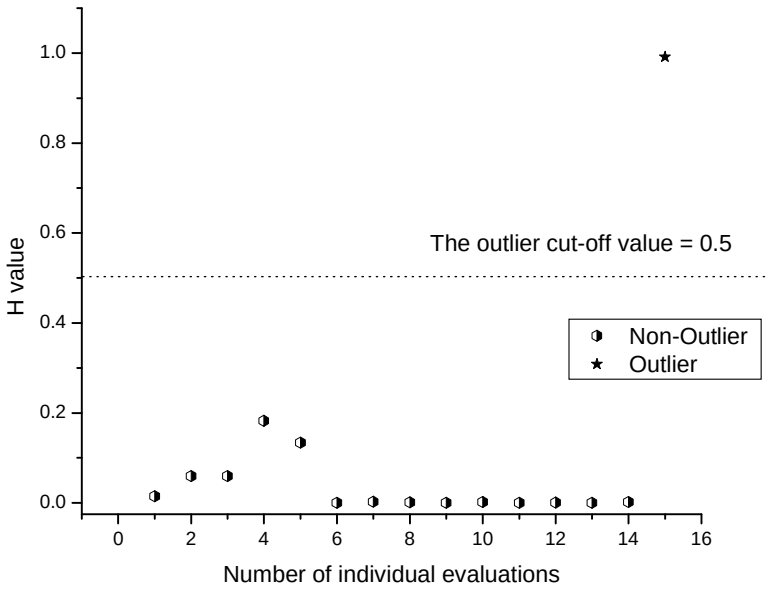


Figure 3.8: Plot of $\boldsymbol{H_{ii}}$ Showing Outlier
Identification When Hamming Distance is Used

In the case when a single faulty *L* individual with a less catastrophic 1-out-of-64 fault impact is analyzed, two outlier points are successfully isolated as shown in the Figure 3.9. Figure 3.10 shows the corresponding plot of the $\boldsymbol{H_{ii}}$ for the same experiment. The detection rate is 100% for this scenario. When compared to the results in Figure 3.5 and Figure 3.6, it can be seen that the identification takes place more frequently with a periodicity of approximately $5 \times E$. This corresponds to the use of a narrower sliding window width as opposed to the $15 \times E$ used in the earlier experiment. In Figure 3.9, the outlier cut-off value is 0.3 as compared to 0.5 in Figure 3.6. Also, the first outlier in Figure 3.10 is closer to the cut-off value which can be expected with a narrow sliding window. A wider sliding window width helps reinforce identification, yet too large a value can delay identification without improving the discrimination among faulty and viable competitors.

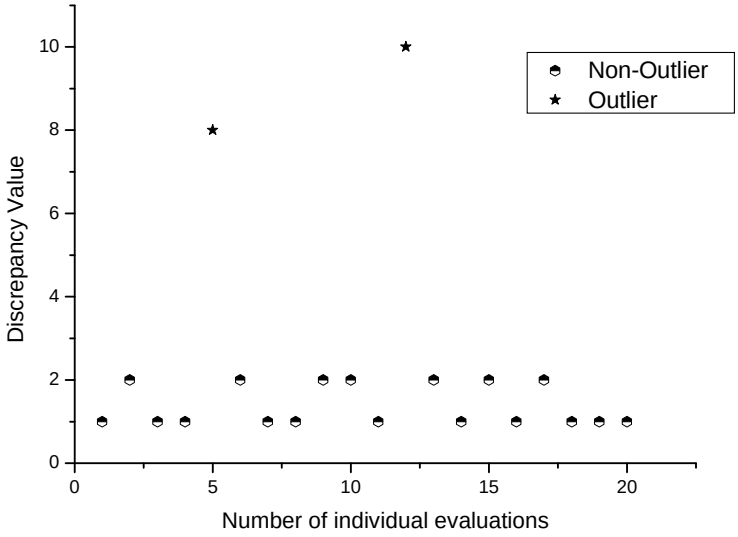


Figure 3.9: *DV* of a Single Faulty *L* Individual With a 1-out-of-64 Fault Impact

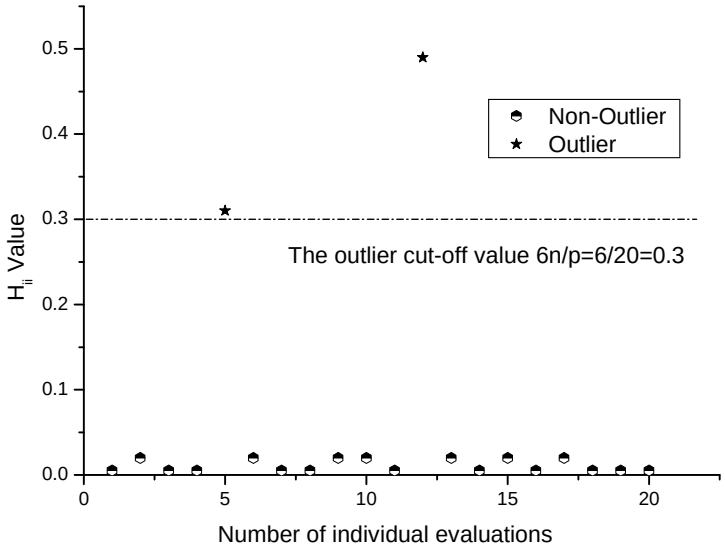


Figure 3.10: Isolation of a Single Faulty *L* Individual With a 1-out-of-64 Fault Impact

For a greater fault impact, the isolation will be more challenging and time-consuming as shown in Figure 3.11 and Figure 3.12. Both Figures depict the isolation characteristics for a single faulty *L* individual with a 32-out-of-64 fault impact. A greater number of

observations are required than the 1-out-of-64 scenario and the divergence of the outlier is also greater. Individuals that are eventually identified as outliers are replaced more often, since the computations involving these individuals produce discrepant outputs. Under the default replacement strategy for discrepancy-free behavior depicted in Figure 3.3, fault-free individuals reside on the FPGA indefinitely. However, in this experiment, they are replaced in accordance with the Replacement Rate R_X=0.16, which corresponds to a guaranteed evaluation period of 100 contiguous iterations out of the E=600 window. Individuals that do not produce discrepant outputs are replaced with other individuals less frequently than ones that do. Thus, individuals that are not fault-affected complete the required E number of iterations to complete evaluation much sooner than the fault-affected individuals. This is because discrepancies trigger immediate reconfiguration as a means of maintaining throughput and improving system availability.

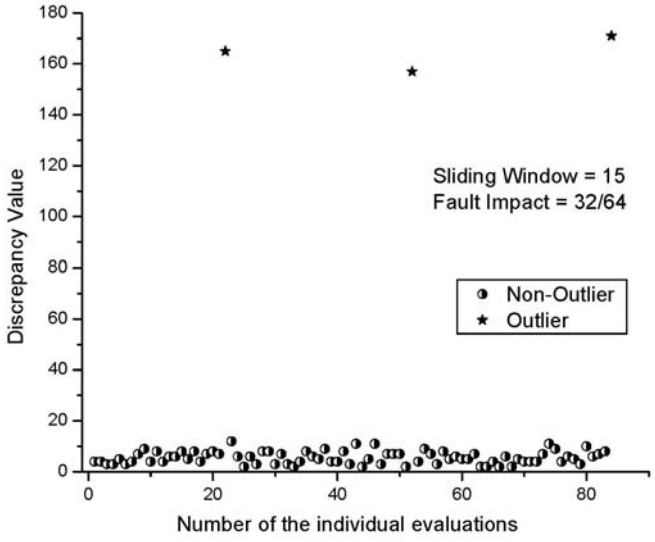


Figure 3.11: *DV*s Observed When a Single Faulty Individual has a 32-out-of-64 Fault Impact

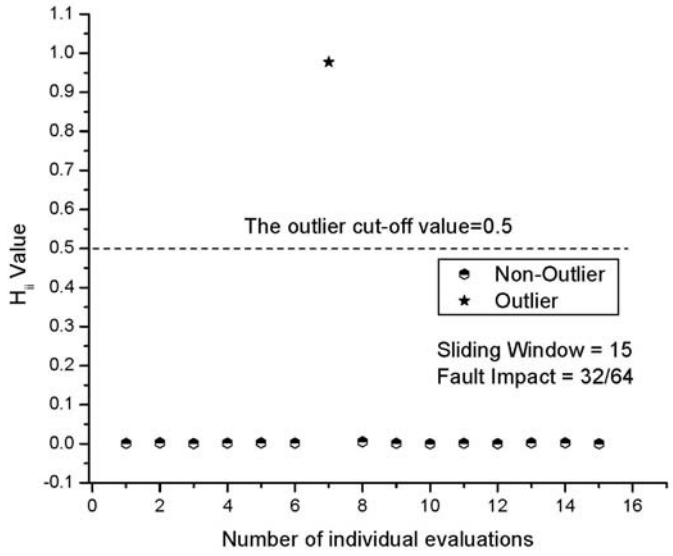


Figure 3.12: Isolation of a Single Faulty *L* Individual with a 32-out-of-64 fault Impact

3.7. Feed-Forward FPGA Circuit Representation Model

The FPGA structure used in the following experiments is similar to that used by Miller and Thompson for GA-based arithmetic circuit design [42]. The feed-forward combinational logic circuit uses a rectangular array of nodes with two inputs and one output. Each node represents a *Look-up Table* (LUT) in the FPGA device, and a *Configurable Logic Block* (CLB) is composed of four LUTs. In the array, each CLB will be a row of the array and two LUTs are represented as four columns of the array. There are five dyadic functions – OR, AND, XOR, NOR, NAND – and one unary-function NOT, each of which can be assigned to an LUT. The LUTs in the CLB array are indexed linearly from 1 to *n*. Array routing is defined by the internal connectivity and the inputs/outputs of the array. Internal connectivity is specified by the connections between the array cells. The inputs of the cells can only be the outputs of cells with lower row

numbers. Thus, the linear labeling and connection restrictions impose a feed-forward structure on the combinational circuit.

As an example of the circuit representation, the 3-bit×3-bit multiplier can be implemented using the above FPGA structure, as shown in Figure 3.13. The entire configuration utilizes 21 CLBs. XOR gates are excluded from the initial designs to force usage of a higher number of the gates than conventional multiplier designs to increase the design space. XOR gates simplify the process of calculating partial binary sums, and thus reduce the number of gates required to build half-adders and full-adders.

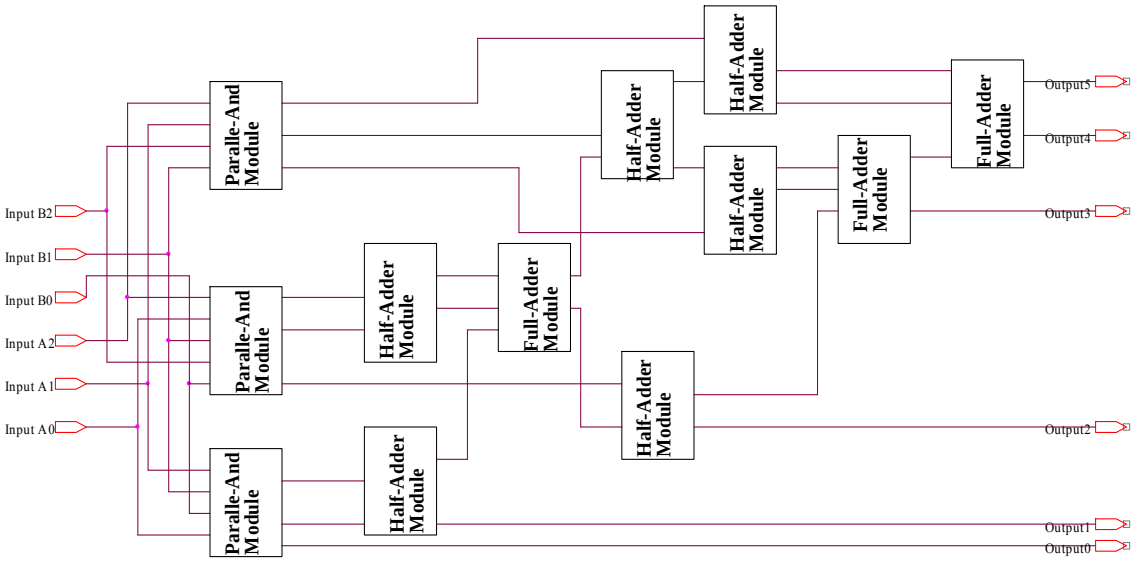


Figure 3.13: Example of a 3-bit×3-bit Multiplier Design

A library of user-defined modules can be defined to instantiate a population of diverse yet functionally-equivalent circuits. In this case study, 20 distinct individuals are created at design-time using a set of 10 or more variations of three fundamental sub-circuits. These consist of *parallel-AND*, *half-adder*, and *full-adder* primitives. For example, 24 different

full-adder designs and 18 different half-adder designs were created for use in building the individual 3-bit×3-bit multiplier designs. Thus, each multiplier is a distinct combination of building blocks, where each building block itself is chosen from among alternate designs in the library. Figure 3.13 illustrates an individual with 3 parallel-AND, 3 full-adder, and 6 half-adder modules.

The population of competing alternatives is then divided into two groups, L and R, where each group uses an exclusive set of physical resources. For crossover to occur such that offspring are guaranteed to utilize only mutually-exclusive physical resources with other resident half-configurations, a two-point crossover operation is carried out with another randomly selected Pristine, Suspect or Refurbished individual belonging to the same group. By enforcing speciation, breeding occurs exclusively within *L* or *R*, and non-interfering resource use is maintained. The crossover points are chosen along the boundary of CLBs so that intra-CLB crossover is precluded. The mutation operator randomly changes the LUT's functionality or reconnects one input of the LUT to a new, randomly selected output inside the CLB.

3.8. Refurbishment of a Unique Failed Configuration – 3-bit×3-bit Multiplier Case Study

In this experiment, GA-based recovery operators are applied to regenerate the functionality in the affected individuals. In order to simulate a hardware fault in the FPGA, a single stuck-at fault is inserted at a randomly-chosen LUT input pin. This fault will affect the *L* individuals in the population. Similar faults are later introduced into the

R individuals. Upon observing the first discrepancy, the same inputs are applied once to the reloaded configurations as a definitive means of damage assessment under a single-fault model. Over 25 experimental runs, an average of 2,171 iterations were required to dependably demote the fitness state of the affected individual from *Pristine* to *Under Repair*. During regeneration, the genetic algorithm performs inter-module crossover and intra-module mutation operator called the *input permutation operator*. Unlike traditional mutation, the input permutation operator alters a specific LUT's functionality, choosing from among AND, OR, XOR, NOR and NAND gates, as also changing the connections to the input pins. Such mutation in conjunction with the crossover operator enables full exploration of a wide range of designs.

Table 3.2 lists the evolutionary regeneration characteristics of CRR for stuck-at-0 and stuck-at-1 faults. The faults were injected at randomly chosen locations in the designs. For the experiment, DV_R DV_O, the repair and operational thresholds, were 2.5σ and 1σ respectively. The use of multiples of standard deviation as the threshold ensures that the system adapts in the case of catastrophic fault conditions, as well as the condition where very few discrepancies are observed. The parameters which control the rate at which individuals are rotated on the FPGA, λ_R and R_X were set at 0.2 and 0.16, respectively. The reintroduction rate of 0.2 implies that 20% of the computations were carried out using a pair of individuals, one of which was *Under Repair*. In spite of this, the effective throughput remains high and above 97.5% on an average. This shows that individuals undergoing repair produce useful output approximately $0.975-(1-\lambda_R)/\lambda_R \times 100\% = 87.5\%$ of the time.

Table 3.2: Regeneration Characteristics for a Single Fault under CBE

Exp. Number	Fault Location	Failure Type	Correctness after Fault	Total Iterations	Discrepant Iterations	Repair Iterations	Final Correctness	Throughput (%)
1	CLB3,LUT0,Input1	Stuck-at-1	52 / 64	1.7×10^7	4.2×10^5	1194	64 / 64	97.7
2	CLB6,LUT0,Input1	Stuck-at-0	33 / 64	8.0×10^5	1.7×10^4	47	64 / 64	97.9
3	CLB5,LUT2,Input0	Stuck-at-1	22 / 64	3.1×10^6	6.8×10^4	193	64 / 64	97.8
4	CLB7,LUT2,Input0	Stuck-at-0	38 / 64	8.1×10^6	1.8×10^5	513	64 / 64	97.7
5	CLB9,LUT0,Input1	Stuck-at-0	40 / 64	2.3×10^6	7.1×10^4	219	64 / 64	96.9
		Average	32.6 / 64	6.4×10^6	1.5×10^5	433	64 / 64	97.6

Using a higher value for λ_R will lead to faster regeneration at an incremental cost to repair throughput. This provides a great deal of adaptability and fine-grained control over system performance measured in terms of availability and regeneration latency. Unlike other circuit design and regeneration approaches, CRR can be optimized to reduce downtime, increase availability, or to speed up the fault identification and regeneration process. The results listed in Table 3.2 indicate that the evolutionary algorithm is capable of regeneration for the tested fault locations. The correctness of the affected configurations is raised from as low as 22-out-of-64 correctness to complete operational suitability. The effective throughput is maintained throughout at above 97.6%. It can also be seen that CRR-based regeneration can be more computational tractable without exhaustive evaluation, as is listed in the Repair Iterations column.

In Vigander's experiment with using a voting system in conjunction with TMR [30], the target circuit is a 4-bit×4-bit multiplier. With a population size of 50, and a crossover rate of 70%, most of the 44 runs developed a set of three modules which vote to provide

fully-fit output for the exhaustive set of 256 unique input combinations. However, it is not always able to identify a single fully repaired individual. Vigander's experiment has a population size of 50, which is 500% greater than the population in the repair experiments attempted herein. Most significantly, it relies on exhaustive serial testing against the set of all possible inputs. CRR, however, achieves refurbishment with runtime inputs, continually providing some validated outputs that maintains useful throughput above 85%. Compared to *Jiggling* [22], which is a similar evolutionary-algorithm based approach to repairing permanent faults, CRR has lower latency by virtue of not relying on exhaustive tracking of the repair candidates. Additionally, the (1+1) Evolutionary System described therein relies on rollbacks to preserve best-fit mutants. CRR, by virtue of depending on a population of higher-fit alternatives that are evaluated temporally over many iterations, precludes the need for rollback of configurations and ensures higher populational fault tolerance capability. Significantly, as opposed to the work of Keymeulen in populational fault tolerance [18], CRR achieves device refurbishment at runtime, while ensuring sustainable levels of throughput with graceful degradation. As compared to the Roving STARs approach [29], CRR minimizes detection latency, as faults are evident immediately upon a discrepancy at the outputs. Also, unlike STARs, by virtue of the runtime-input based performance evaluation, CRR leverages partially-fit configurations to provide some functional throughput. This effectively improves the granularity of spare usage to include those affected by stuck-at faults, as the GA may evolve solutions that use fault-affected resources in generating repair configurations.

In Summary of the Repair experiments, evolutionary regeneration addresses a problem domain that is distinct from evolutionary design. Namely, regeneration can benefit from a population of partially working designs which provide diverse, relevant alternates. This also allows departure from conventional fitness evaluation with a rigid individual-centric fitness measure defined at design-time. CRR uses instead, a self-adapting, population-centric assessment method at runtime. CRR relies on the consensus observed among a group of individuals to evolve and adapt fitness criteria for individual members, thus providing graceful degradation. By utilizing outlier detection techniques that work temporally without the need for exhaustive testing, CRR provides a fault tolerance technique that maximizes device throughput during the fault detection process.

While the pre-existing methods focus on creating a single fully-fit configuration, CRR extends this to maintain a population of solutions that have a higher average fitness. This ensures the adaptability of the population of viable alternatives to a variety of unanticipated faults. An additional benefit of maintaining a population of diverse partially-fit individuals is that when the inputs to the system are localized to a subset of the set of all possible inputs, even partially-fit individuals can assist in generating expected outputs, thereby improving the rate of viable throughput during recovery.

Population-centric assessment methods such as CRR can provide an additional degree of adaptability and autonomy to fault-handling in reconfigurable devices. The demonstrated potential of such population-centric methods can be further enhanced as follows, and as further explored in the subsequent chapters. After discrepancy detection, a CGT method which tracks utilization of resource sets among individuals in the population, is used to

identify the stage containing the faulty resource. This is readily incorporated within the configuration selection step of CRR. The genetic operators are then applied only to that isolated stage to attempt recovery, thus providing an approach to extend the CRR method to larger circuits while remaining computationally tractable.

In order to accelerate the fault recovery process, a fault detection and isolation method that functions on the run-time inputs is required. Further, the method has to operate on the basis of comparisons between two functional configurations' performance. In the next chapter, a discrepancy-enabled dueling scheme is presented that enable fast fault detection.

CHAPTER 4: FAULT ISOLATION USING GROUP TESTING

A fault detection and isolation method for stuck-at logic faults in FPGAs is developed starting from a simple reconfigurable device model. A discrepancy detector is realized and implemented in CMOS to demonstrate the viability of the approach. Starting from a fully-articulating fault model, a general outline for discrepancy-enabled group testing is generated and expanded to the a partially-articulating fault model. Finally, examples of adapting group testing techniques to improve the performance of GAs and also for exhaustive BIST-based techniques are presented.

4.1. Motivating Example and Problem Definition

In order to better understand the group testing problem at hand, consider an analogy termed the *Treasurer's Problem* which is related to the Counterfeit Coin Problem [43]. The Counterfeit Coin Problem is extended here by analogy to support arbitrary groupings of logic cells within FPGAs. In this Treasurer's Problem, legitimate coins are made of gold, with the face value of the coins being proportional to their weight. However, some counterfeit coins have other metals mixed in with the gold, and these counterfeit coins are to be identified and removed. The weight of an impure coin is different from the weight of pure coins of the same denomination. The treasurer must inspect large quantities of coins for authenticity. Most significantly, since the number of counterfeit instances is small relative to the total number of coins present, the treasurer does not weigh the coins individually. Instead the coins are in a vat, and the treasurer retreives coins from the vat

to fill bags containing exactly 100 monetary units worth of coins. The number of coins in each bag may vary because of their multiple denominations, yet due to the property that their mass is proportional to their denomination then only two equally-valued legitimate bags will display equal weight.

Using a pan balance, the treasurer compares the weight of two bags at a time to determine whether they are equal weight or not. The coins from the bags may be returned to the vat after weighing, so that they can be filled in other bags later after shuffling. Given these pre-conditions, a number of questions arise about how the treasurer will identify any faulty coinage such as: How many weighings will the treasurer need to identify bags containing the impure coins? Can the impure coin be identified, if there was only one?

These questions are analagous to the problems addressed in this paper for identification of a faulty physical resource used by a functional arrangement of FPGA configurations. FPGA devices are composed of an array of logic resources such as LUTs that are utilized by functional configurations just as the coins are grouped into a bag for weighing. A digital design can be mapped onto the resources on an FPGA in several ways, just like a bag worth 100 monetary units can be filled with coins of different denominations in several different ways. When one of the resources used by a configuration is faulty, the output of the configuration in response to an input may be faulty. Identifying the faulty resource from among many fault free resources, without testing the resources individually is a challenging task. Exhaustive testing of the individual resources is time consuming which takes the device offline and reduces its availability. By analogy, if the coins were weighed and checked individually, the time required would be phenomenal to locate a

single faulty coin out of thousands of coins. Instead, we re-cast the problem of identifying the faulty resource into one of making choices for group comparison from among the given FPGA configurations.

4.2. Fault Isolation by Discrepancy-Enabled Repetitive pairing

Robust fault detection is central to the problem of enhancing the fault-handling capabilities of digital circuits. A common limitation facing many fault detection schemes is that the failure detector itself may fail. A fault involving the checker may be undetectable or result in the corruption of otherwise valid outputs. Traditional approaches to fault-detection typically rely on coding-based schemes or redundancy using a single voter, comparator, or error detector. Those fault checkers possess a single point-of-failure exposure involving the detector elements, or must rely upon special test-vectors or data encodings to isolate them. Detector components in the reliability path have been referred to as golden elements [22] because the fault-handling strategy relies on them to be fault-free. The following sections develop an alternative approach to self-checking fault detection based on random pairings and temporal voting to reduce such exposures.

Table 4.1: Comparison of Fault-Detection Techniques

	Special Encoding	**Number of Elements**	**Transient Fault Detection**	**Detection Latency**	**Special Test Inputs**	**Fault Isolation Resolution**	**Interrupt-Free Testing**
Temporal CED Schemes	Yes	1	Optional	Algorithm Dependent	Yes	Sub-circuit Under Test	No
Spatial CED Schemes	Yes	≥ 2	No	Algorithm Dependent	Yes	Sub-Circuit Under Test	No
TMR	No	3	No	Minimal	No	Voter Elements	No
Discrepancy Mirrors	*No*	*2*	*Yes*	*Minimal*	*No*	*Individual Logic Resources*	*Yes*

Table 4.1 lists characteristics of selected fault-handling strategies. Specialized encoding schemes are often required by CED approaches as opposed to TMR and the Discrepancy Mirror methods which do not. The number of functional logic elements required by TMR is greater than that of the other schemes. Discrepancy Mirrors provide inherent transient fault coverage with minimal detection latency. They also support fine-grained resolution of the fault location, without interruption to the data throughput flow when a fault occurs. Thus, Discrepancy Mirrors offer improved detection of permanent and transient faults, with reduced time and space overheads. Section 4.3 provides the design of the discrepancy mirror approach. Results of simulations and fault location experiments conducted in the case study are given in Section 4.4.

4.3. Designing a Discrepancy Mirror – Case Study

The Discrepancy Mirror approach is a duplex redundancy technique that utilizes alternate physical configurations from a population of candidate designs that are functionally equivalent. As shown in Figure 4.1, the technique is composed of three phases, namely Selection, Detection, and Preference Adjustment. The Selection phase selects two candidates, each of which utilize mutually exclusive subsets of the resources under test. The Detection process uses the Discrepancy Mirror logic shown in Figure 4.2 to check for bit-wise equivalence between outputs of the candidates as described below. The Preference Adjustment phase utilizes the results of successive comparisons to update the pairing strategy during subsequent selections. These steps will be explained below in the context of a FPGA-based realization whereby two configurations of the functional logic are loaded in tandem.

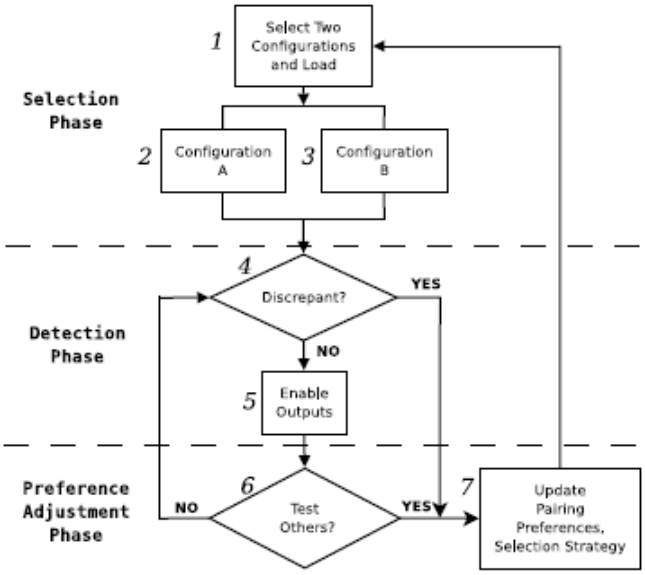


Figure 4.1: Discrepancy Mirror-based Scheme

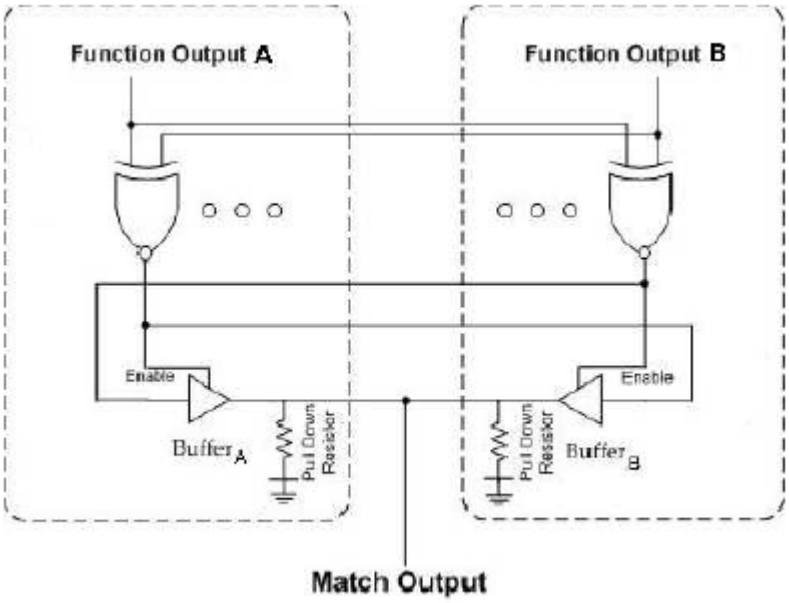


Figure 4.2: Discrepancy Detection Circuit

4.3.1. Selection Phase

Candidate designs are selected from a population developed at design time, either manually or via a CAD tool. Random pairings or an adaptive scheme based on the results of Preference Adjustment can be employed. This process is identified as Step 1 in Figure 4.1. The selected designs are then loaded as the active configurations during Step 2 and Step 3. Identical input operands are applied in parallel to each configuration and the outputs are redundantly computed for comparison in the next phase.

4.3.2. Detection Phase

As shown in Step *4* in Figure 4.1, the discrepancy mirror circuit is used to identify whether the outputs of the two configurations under test agree. A complete instance of

the discrepancy mirror is obtained whenever two configurations are loaded, since the discrepancy detector consists of two identical sections as shown in Figure 4.2. Assertion of MATCH output from the discrepancy mirror indicates the absence of a single-fault in the configurations under test, as well as the logic in the discrepancy mirror. The data output is enabled if and only if no faults are detected as shown in Step *5* in Figure 4.1.

The inputs to the Discrepancy Mirror shown in Figure 4.2 as "Function Output A" and "Function Output B" are generated independently. If there is a fault in a resource utilized by either of these configurations, then a discrepancy is observed at the output. The truth table shown in Table 2 describes the operation of the circuit shown in Figure 4.2. Outputs from the function configurations A and B are applied as inputs to both the XNOR gates. The output from each XNOR gate acts as the ENABLE signal for the tri-state buffer in the same half, as well as the input to the tri-state buffer in the other half of the discrepancy mirror. The tri-state buffer outputs are tied together to form a Wired-OR connection which provides the MATCH output signal. The pull-down transistors hold the signal to a logic '0' level when the tri-state buffer output is in a high-impedance state. In an active-high non-inverting tri-state buffer, the input is buffered at the output only when the ENABLE signal is high. When the ENABLE signal is low, the output of the buffer is in a high-impedance state.

A CMOS model of the discrepancy detector was created using PSpice. The circuit was constructed using 44 p- and n-channel MOS transistors with a 1.5 micron minimum width, and a 600nm length. The width of the p-mos transistors was set to thrice the width of the n-mos transistors. Figure 4.3 below shows the PSpice schematic and Figure 4.4

shows the transient response of the circuit demonstrating that the circuit conforms to specifications enabling the correct identification of discrepancies. Subsequently, the circuit was also simulated on the Xilinx Virtex-II Pro FPGA using the ModelSim-II simulator.

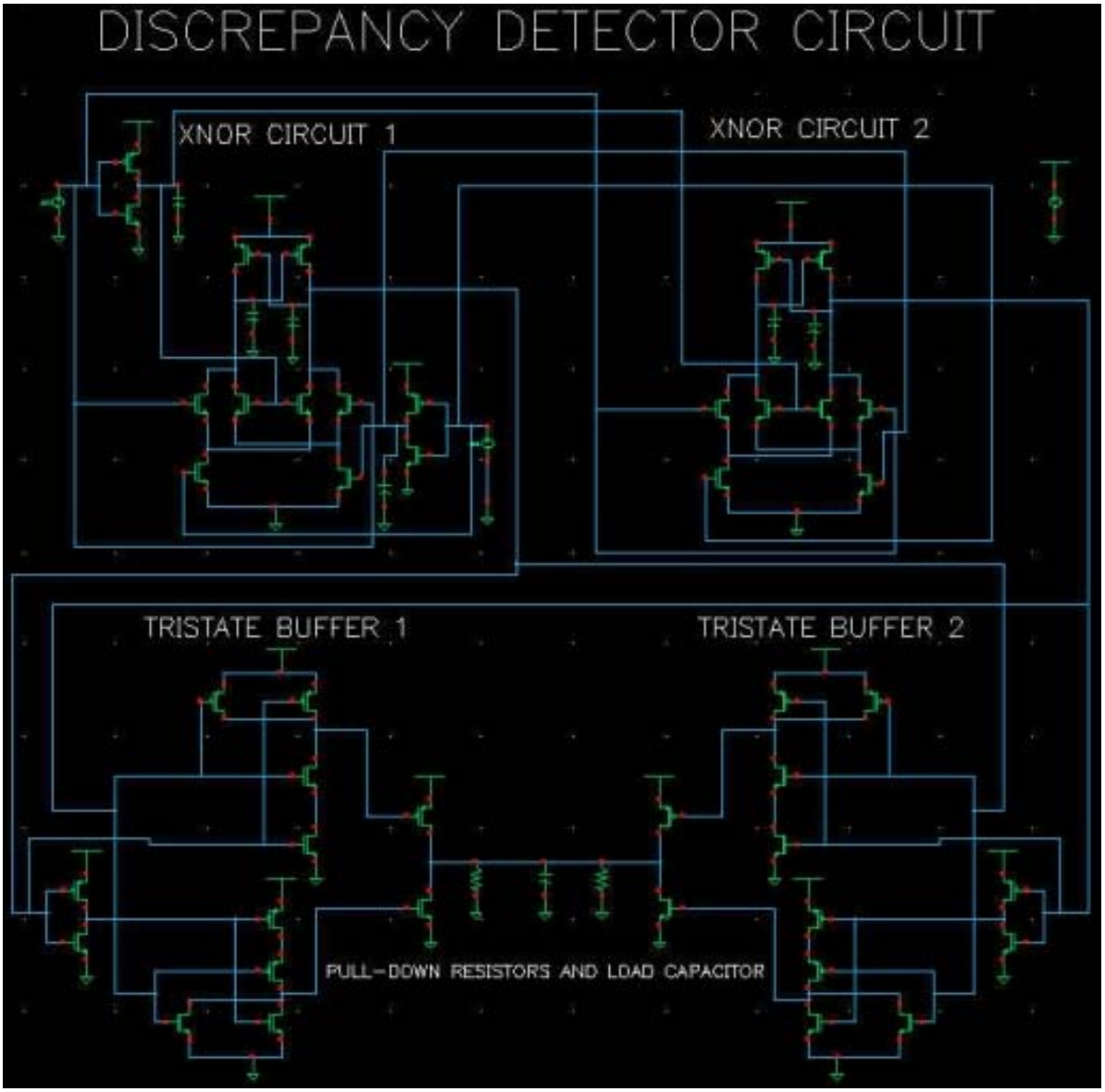


Figure 4.3 Discrepancy Detector Circuit Schematic Layout

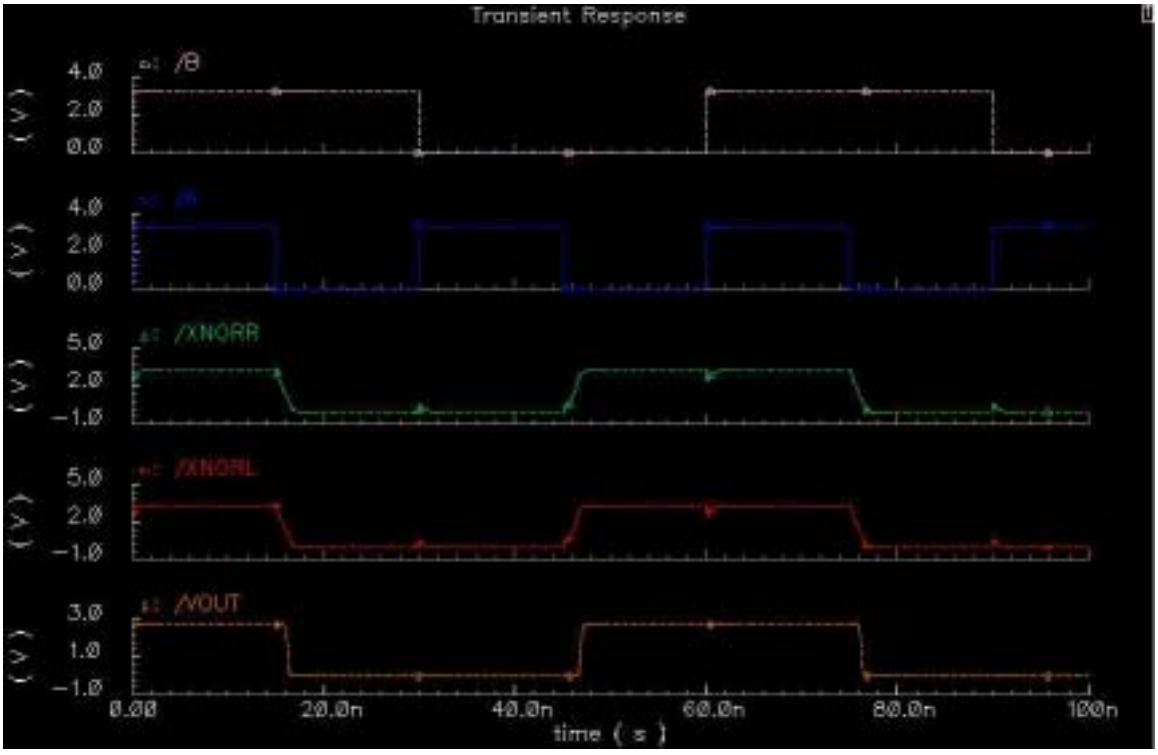


Figure 4.4 Transient Response of the CMOS Discrepancy Detector Circuit

As listed in Table 4.2, the response of the circuit is robust to several possible fault scenarios. If either of the XNOR gates fail, then one of the two tri-state buffers will be disabled and the other will have an input of zero, thus MATCH will be a '0', signifying discrepancy. If the tri-state buffers fail, producing a high impedance output, the pull down resistors in the circuit will hold the signal to '0'. The wired-OR connection reduces single points of failure to a stuck-at fault exposure for the MATCH output.

Table 4.2: Discrepancy Mirror Truth Table

A	B	$XNOR_A$	$XNOR_B$	ENB_A	ENB_B	TRI_A	TRI_B	**Match**
0	0	1	1	1	1	1	1	1
0	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1

4.3.3. The Preference Adjustment Process

Step 6 and Step 7 comprise the Preference Adjustment process. When the Discrepancy Mirror returns a MATCH output, alternate configurations can be loaded for testing or the resident fault-free configurations can be used. The output from the discrepancy mirror over a period of time indicates the relative fitness of the different configurations. This information can isolate the fault location and aid in regeneration of lost functionality through the identification of alternate resources. The cumulative discrepancy information from diverse pairings over time can be used in Step 7 to modify the selection preferences for the configurations in the population.

4.4. Analysis of Fault Isolation with a Simplified Articulation Model

The operation of the discrepancy mirror circuit was verified on a Xilinx Virtex-II Pro FPGA platform using ModelSim-II. The pull-down resistors were emulated using digital components as shown in the Xilinx data sheet [44]. The waveform for the MATCH output was asserted whenever the inputs to the discrepancy mirror were in agreement. The simulation waveform showed a LOW signal whenever the MATCH output was a ‘0’.

In the Xilinx Virtex-II Pro FPGA, when pull-down resistors are emulated, a LOW signal is the equivalent of a logic-0 output. The circuit was also simulated using Cadence SPICE. The entire circuit was also realized using a total of 44 p- and n-channel MOS transistors using a 1.5 micron minimum width technology with a length of 600 nm. A total of 44 CMOS transistors were utilized to realize the circuit. The widths of the pMOS transistors in the XNOR circuit were selected to be thrice the widths of the nMOS transistors to shape the waveform rise and fall times, to develop the required timing characteristics. The simulation results and waveforms obtained indicated behavior conforming to Table 4.2 and Table 4.3.

Table 4.3: Discrepancy Mirror Fault Coverage and Response

Component	**Fault Scenarios**				**Fault-Free**
Function Output A	*Fault*	Correct	Correct	Correct	Correct
Function Output B	Correct	*Fault*	Correct	Correct	Correct
$XNOR_A$	Disagree(0)	Disagree(0)	*Fault :Disagree(0)*	Agree (1)	Agree (1)
$XNOR_B$	Disagree(0)	Disagree(0)	Agree (1)	*Fault: Disagree(0)*	Agree (1)
$Buffer_A$	0	0	High-Z	0	1
$Buffer_B$	0	0	0	High-Z	1
Match Output	*0*	*0*	*0*	*0*	*1*

Two sets of experiments were performed to analyze the fault isolation latency. Both experiments sought identify the number of iterations required to identify the faulty resource in the case of single fault. A simulator was constructed using a C-language program for simulating the Selection, Detection, and Preference Adjustment phases. The inputs to the simulated mirror were obtained using random number generators. Random input values were applied to two configurations chosen at random from the pool of competing configurations. More formally, let U denote the set consisting of all the logic resources in the FPGA, S denote the pool of resources suspected of being faulty, and

$C_i \subset U$ denote the set of resources used by the i^{th} configuration. Initially, $|S| = |U|$. A process of m successive intersections among the subsets $C_j \cap C_{k=j}$ ($i \leq j$, $k \leq m$) are performed. Each successive intersection reduces $|S|$ until after the m^{th} intersection at time $t = m$ eventually $|S| = 1$, completing the fault-location process. Each experiment was conducted with $|U|$ = 1,000, 10,000, and 100,000. The expected number of iterations to isolate the fault are reported for the mean values observed over 100 trials of the simulator. An individual logic resource is the equivalent of a CLB in an FPGA so the range of resource pool sizes reflect a realistic device scenario.

In the first set of experiments, the inputs applied consistently articulate any fault in the logic resources used by the configurations under test. Thus, a match output indicates that the logic resources used by the configurations being compared are completely fault-free. A discrepancy between the configurations' functional outputs indicates the presence of at least one resource fault. Assertion of the MATCH output exonerates all logic resources currently being used, and a de-assertion of the MATCH output implicates the subset of logic resources currently being used as suspect. The faulty resource is isolated after m pairings through a process of successive intersection. Figure 4.3 shows the faulty resource can be identified using an expected value of 11.1 pairings when $|U|$ = 1,000 and half of the resources are utilized by each configuration. When $|U|$ = 100,000, the mean number of pairings required to locate the fault increases by much less than a factor of ten to a value of 17.6. Under more demanding parameters, when $|U|$ =100,000, and when only 5% of the resources are being used by each configuration, a mean value of 63.7 pairings are required to isolate the faulty resource.

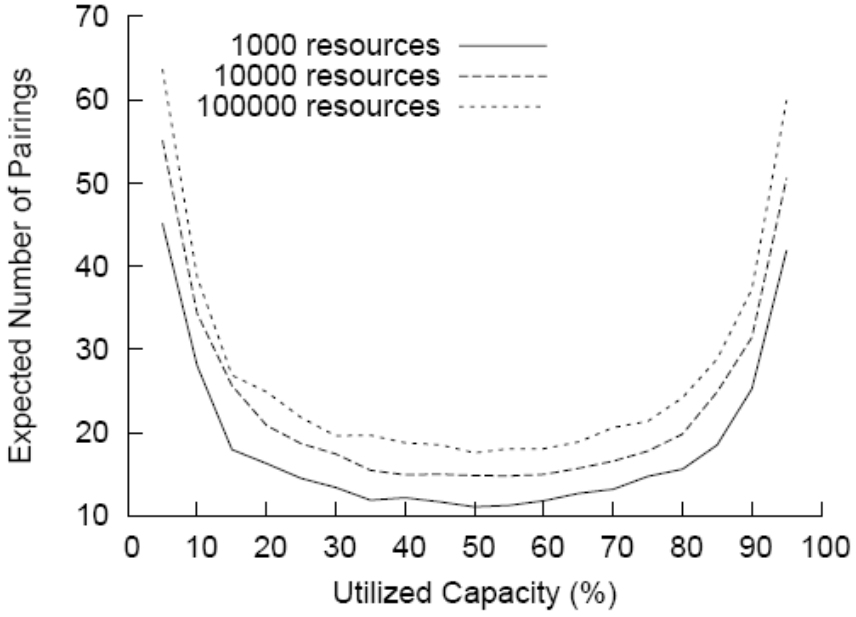


Figure 4.5: Fault Isolation with Perpetually Articulating Inputs

Depending on the inputs applied, the fault in the functional logic under test may remain dormant and thus some inputs would not articulate a visible discrepancy. In this case, a match output from the discrepancy mirror cannot evaluate whether all the resources under test are fault-free. A discrepant output is a definitive indicator of the existence of a single-fault. With such *Intermittently Articulating Inputs*, the discrepancy mirror based scheme requires additional random pairings to isolate the single-fault. As shown in Figure 4.4, when $|U| = 1,000$, with resource utilization at 45%, an expected 42 random pairings are required to uniquely identify the faulty resource. When $|U| = 100,000$, the best performance is observed for a utilization of near 50%, where the expected value of random pairings is 64.1.

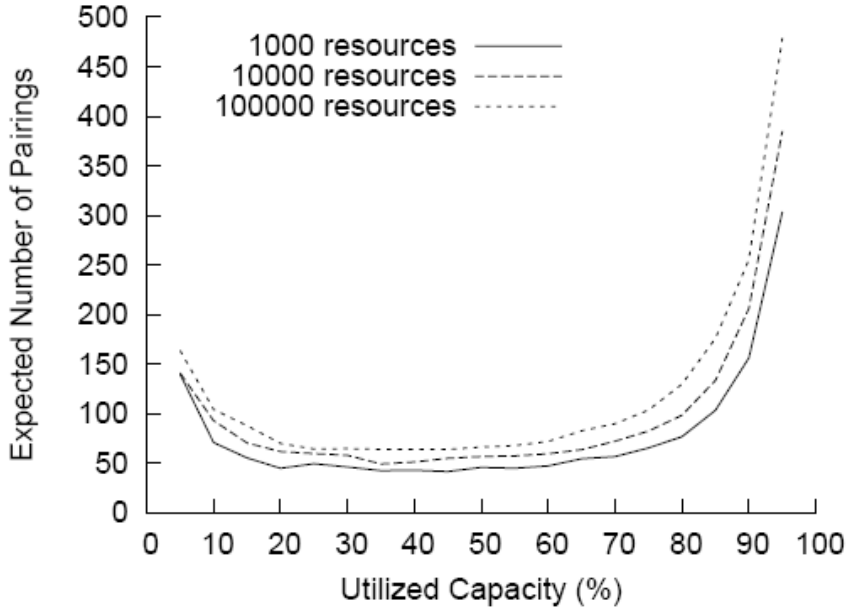


Figure 4.6: Fault Isolation with Intermittently Articulating Inputs

The discrepancy mirror is capable of handling faults in either the functional logic or the detector. If there is a failure in either, then the output of the mirror remains de-asserted indicating the presence of at least one resource fault. It is able to isolate the faulty resources with a expected number of random pairings that is sub-linear in the number of resources under test. It does not depend upon a specific coding scheme or a pre-defined set of inputs. Random pairings of configurations perform successive intersection of the resources under test to isolate the faulty resource. Figure 4.3 and Figure 4.4 show that more pairings are required to identify the faulty resource when the utilization of available resources is below 20% or above 80%. In these situations, each successive pairing implicates (or exonerates) a smaller sub-set of resources than when half of the resources are utilized. Finally, using a discrepancy mirror based approach, the number of pairings required for fault location increases sub-linearly with an increase in $|U|$. For example, at

50% utilization, the expected number of pairings to locate a fault within pools of 1,000, 10,000, and 100,000 resources are 11.1, 14.9, and 17.6, respectively, demonstrating the viability of the technique. Though the model is abstract, and of minimal complexity, the case-study demonstrates the viability of the discrepancy detector, and provides the basis for investigating group testing-based approaches to FPGA fault isolation.

There are certain cases where the simple fault isolation scheme described above may fail to converge on a single faulty resource. A trivial case is when all the resources available on the FPGA are used by each configuration. If the application demands that all the resources be used, then isolation cannot occur through the process of successive intersection. Also, in cases where a very low number of resources are used by individual configurations, it is possible that none of the individuals utilize the faulty resource, leading to the state where no discrepancies will be observed. The most challenging case is when multiple individuals utilize the faulty resource. In this situation, the history matrix elements corresponding to the intersection of the subset of resources used by these individuals will have no relative differences, and will all have the highest value. Successive intersections between the resource subsets will not lead to any further fault isolation. For example, with a resource utilization of 40% in a device with 40,000 unit resources, isolation proceeds as shown in Figure 4.5. The isolation cannot be completed, and after about 23 iterations, the number of suspected faulty elements stays a constant at 36. Any further isolation cannot occur since there is none of the intersections that may follow provide any additional isolation information. This necessitates an algorithm based on group testing.

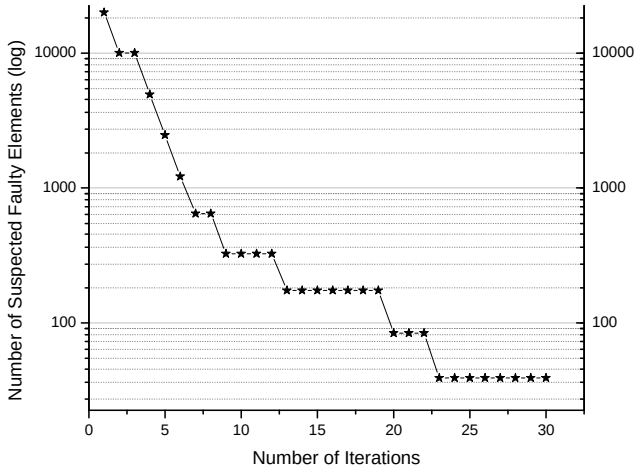


Figure 4.7: Successive Isolation as Input Iterations Increase

4.5. Fault Isolation using Halving and Column-Swapping

To avoid the problem of not being able to proceed with isolation in certain cases where successive iterations do not provide isolation information, a dueling algorithm is proposed which tries to emulate halving. *Halving* is the process of successively reducing the size of the subgroup under test by half until, finally a test of a single element is required to identify the faulty element.

The algorithm works by swapping columns in the configurations of individual elements. When the fault isolation process approaches a state of stasis, some of the columns in the individuals are swapped. The number of columns to be swapped is determined by considering the number of resources currently suspected of being faulty. A number of columns equal to half of the remaining number of suspect elements are swapped with other columns in the same individual. This will introduce new information, as some of the suspected faulty elements used by the individual earlier will no longer be used, for

example. Swapping is restricted only to the columns to facilitate future implementation in FPGA hardware. As shown in Figure 4.6, isolation proceeds till a single faulty element is isolated under the same conditions under which the results shown in Figure 4.5, for dueling without swapping were obtained.

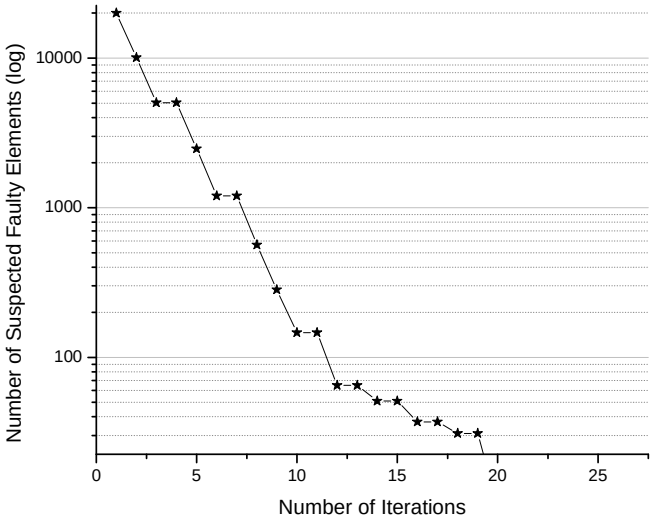


Figure 4.8: Isolation Progress when Halving is used

In order to analyze the behavior of the dueling algorithm with modified halving, further experiments were conducted to see the implications of various factors on the isolation process. In each of the following experiments, the population size specifies the number of competing individual configurations in the population. Resource utilization, expressed as a percentage signifies the amount of available resources used by an application implemented on the FPGA. The FPGA device is simulated by using a square matrix of order n where n denotes the number of rows and columns in the device. Each of the experiments that follow list average values observed over 100 experimental trials.

The effect of the size of the isolation problem was evaluated by applying the proposed technique to simulated FPGAs of various array sizes. As shown in Figure 4.7, for an isolation problem where there are 100 rows and columns, or 10000 elements, only an average of 14.3 iterations are required to isolate a single fault. As the size of the array containing the fault increases, the increase in the required number of iterations is minimal. For example, for the difficult case where there is a single fault in 1 million resources, the algorithm requires only an average of 27.4 iterations to isolate the fault, showing that the algorithm scales well with the size of problem.

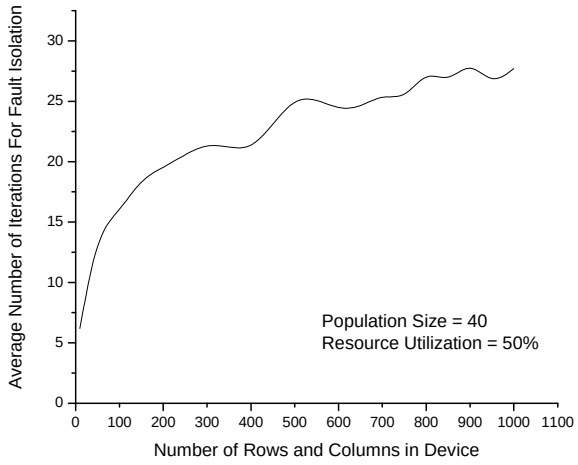


Figure 4.9: Isolation Performance as a Function of the Total Number of Elements

As the population size increases, fault isolation is expected to become faster, since more information will be available to the algorithm from the increased population size. However, a very high population size may lead to more individuals being affected by the same fault. As shown in Figure 4.8, the number of iterations required for isolation, with 40000 elements, and 50% resource utilization shows a tendency to decrease with an increase in the population size. For a population of size 60, only an average of 17.2

iterations are required for isolation. Practically, however, a very high population size will imply the need for a higher number of alternative individual configurations. A population size of 30 seems to be an ideal tradeoff between ease of isolation, and the difficulty of generating increased number of individuals.

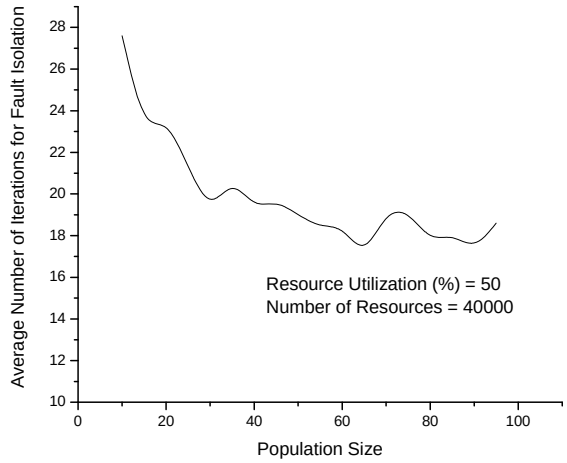


Figure 4.10: Isolation Performance as a Function of the Population Size

4.6. Isolating Embedded Cores using Group Testing

Although group testing-based methods are primarily presented as a tool to improve upon existing run-time fault isolation techniques, they are also amenable to post-manufacturing testing of FPGAs. In this chapter a specific example of using group testing techniques to accelerate the isolation of faulty embedded cores in FPGAs is presented.

The current generation of 65 nm FPGAs by Xilinx, such as the Virtex-5 platform FPGAs introduce space-efficient hard IP cores implemented using the column-based *Application Specific Modular Block (ASMBL)* architecture. The Virtex-5 platform provides anywhere from 32 to 640 embedded DSP48E cores across a range of devices [45]. These cores are

designed, placed, and routed into the fabric of the FPGA, and have been characterized and verified to optimize performance. Unlike soft IP cores, these enable designers to utilize the *Configurable Logic Blocks (CLBs)* as general-purpose logic resources and minimize the space and power required to implement DSP applications on FPGAs. The embedded IP cores are characterized by their predictable timing and are optimized to work efficiently in a manner independent of the rest of the design. These cores are highly customizable based on the designers requirements and provide a range of in-built structures for efficient arithmetic calculation and signal processing requirements. All these characteristics lend to more efficient implementation of an entire system on an FPGA known commonly as a *System On Programmable Chip (SOPC)*. The development of FPGAs with an increasing number of embedded hard IP cores drives the need for faster testing methods for failures in the cores.

The embedded cores are distributed throughout the FPGA fabric and as an integral part of the computational resources, these require extensive post-manufacturing testing and verification. It is therefore important to develop testing methods to identify hardware faults with minimal latency and resource overheads.

4.6.1. BIST-based Testing of Embedded FPGA Cores

Advances in FPGA production technologies have improved capabilities to the point where FPGAs have dedicated embedded cores, in addition to multiplexers and Block RAMs. The most widely accepted approach to detect faults at the chip level in VLSI is to apply BIST on the components [46-48]. The built-in nature of BIST also allows testing

the chip in a variety of working environments. In BIST both the *Test Pattern Generation (TPG)* and *Output Response Analyzer (ORA)* are incorporated inside the chip. Assuming that all levels of the hierarchy use BIST, each element can test itself and transmits the result to the succeeding level in the hierarchy. BIST also increases controllability and observability by providing access to the internal nodes since tester logic is located on the chip. BIST allows tests to be run at system speed and eliminates this gap.

BIST has been the choice of convention for testing Embedded Memory [46, 47]. Conventional ASIC BIST techniques typically accrue between 10% to 30% area overhead and delay penalties [48]. Therefore, it is essential that the FPGA core test method leverages the reprogrammability inherent in FPGAs. An additional advantage of utilizing the programmable feature of an FPGA to test itself is that BIST logic can be removed when the circuit is reconfigured for another use and testability is achieved without permanent area overhead or performance degradation.

There has been considerable research on developing and applying BIST techniques for programmable logic resources in an FPGA including CLBs [49, 50] and interconnect matrix of routing resource [16, 51]. Abramovici and Stroud [49] presented BIST architecture to test CLBs in an FPGA. In their scheme, a column or (a row) of CLB is configured to generate pseudo-exhaustive test patterns to alternating columns of identically configured CLBs under test. They use two identical TPGs to detect any fault in the CLBs used to construct TPGs. Comparator-based ORAs monitor the output of the BUTs and latch mismatches due to faults. The BUTs are tested and configured for

different modes of operation. Each complete test (session) covers only half of the CLBs and another session is required to test the other half.

The diagnostic procedure called MULTICELLO (Multiple faulty Cell Locator) developed by Abramovici et al., identifies faulty BUTs based on the failing BIST results. Stroud and Garimella [52] targeted multiple regular structure cores including memories and multipliers and developed a diagnostic procedure based on the extension of the MULTICELLO algorithm. The diagnostic procedure is performed in five steps. They presented a BIST approach in which neighboring blocks are compared by a set of ORAs. Thus, each core is observed by two sets of ORAs and is compared to two different cores. Circular comparison of the first and last block covers the corner block. Following and applying the MULTICELLO algorithm, Garimella and Stroud [53] presented development of an automated BIST generation for embedded Block RAMs in an FPGA, based on parameterized VHDL model. The MULTICELLO algorithm provides a good diagnostic resolution and is able to locate the faulty blocks (unless all blocks have equivalent faults). However, it is not applicable when testing a set of two blocks in cascade mode. For example, in many applications and operations it is required that two DSP blocks cascaded together to produce the final outputs. In this case, they produce different outputs and therefore it is not possible to compare the outputs of neighboring blocks.

Renovell et al. [54] present a method to test the LUT/RAM modules of FPGAs using a minimal number of test configuration by proposing a model architecture with *N* inputs and *2N* memory cells. With a unique test configuration, they test a single module by

extending conventional algorithms for testing SRAM modules such as the *March tests* [55]. They also propose a unique test configuration called *pseudo shift register*. In this method, the circuit operates as a shift register and the MATS++ algorithm is adapted to test the FPGA RAM modules. However this method is limited to the SRAM modules on the FPGA, or the LUTS operating in the SRAM mode. Current state-of-the-art FPGAs such as the Spartan-3a DSP FPGAs from Xilinx offer embedded SoPC DSP modules that include dedicated 18×18 multipliers along with 18-bit pre-adder and 48-bit post-adder/accumulations and dedicated DSP circuitry consisting of DSP48A slices [56].

Earlier Sarvi et al [57] developed a diagnostic method to detect and locate faulty embedded cores in FPGAs using BIST was developed. However, the technique configures the device twice in order to complete fault isolation. The method partitions the cores on an FPGA into two groups and conducts BIST on each of these groups. Fault isolation is achieved by comparing the results of the two tests. Under this scheme the two configurations are constructed to enable isolation by comparison. In post-processing, defectives are identified by analyzing the results of comparisons among blocks enclosed within the same group. However, this method fails to isolate faulty blocks when there is a defective block in each of the compared pairs.

Improvement over previous approaches is attained using an automated diagnostic methodology that is applicable to different cores, including DSP cores, that takes into account the different modes of operation such as *cascade* and *direct*. The group-testing enhanced method is scalable to different FPGA families including the Xilinx XtremeDSP products and the Virtex-5 family of FPGAs. Further, these techniques can be easily

extended to provide testing coverage for new families of embedded cores on FPGAs since the method is core-independent. A significant improvement is the one-shot testing of all embedded cores of a specified type using a single test pattern. Group testing techniques are utilized to generate a non-adaptive testing regimen that involves a single group of tests executed concurrently. The test provides complete coverage for all cores of a type on the chip by dividing the cores-under-test into subsets with a cardinality of four. By generating, comparing and encoding the outputs produced by the cores in response to the test pattern, complete fault resolution is achieved in a single test.

4.6.2. Enhancing Embedded Core BIST using Group Testing Techniques

The embedded IP cores in the Xilinx Virtex-5 family of devices are distributed evenly throughout the fabric ensuring optimal timing. The BIST technique proposed in this article utilizes the CLBs adjacent to the embedded cores to realize the TPG and the ORA. Each embedded core comprises a *BUT*. The current generation Virtex-5 FPGAs from Xilinx include embedded cores in the form of 36-Kbit Block dual-port Block RAMs and Advanced DSP48E slices. The DSP48E slices provide a range of functionality such as two's complement, multiplication, and optional adder, subtracter, and accumulator. These also provide pipelining and dedicated cascade connections. The number of DSP48E slices in the Virtex-5 FPGAs varies from 32 in the XC5VLX30 device to 640 in the XC5VSX95T device. In the experiments described here, the DSP48E slices are the blocks under test.

Under the proposed group testing-based technique, the *m* embedded cores on the device are divided into *m/4 = n groups* of BUTs. Tests are conducted on these groups to provide fault isolation in a *single-stage*, *non-adaptive* group testing regimen. Comparators k_n generate a *PASS/FAIL* result based on discrepancies between the outputs of two of the BUTs. For a group of 4 BUTs, a total of six comparators are required to compare each BUT's output with that of all the other BUTs in the same group. For purposes of simplicity, Figure 4.10 shows the replicable BIST model in its smallest scale, considering one such group of 4 BUTs, numbered $\boldsymbol{B_0}$ through $\boldsymbol{B_3}$. It is assumed that the CLBs and routing resources have been tested for correct functionality.

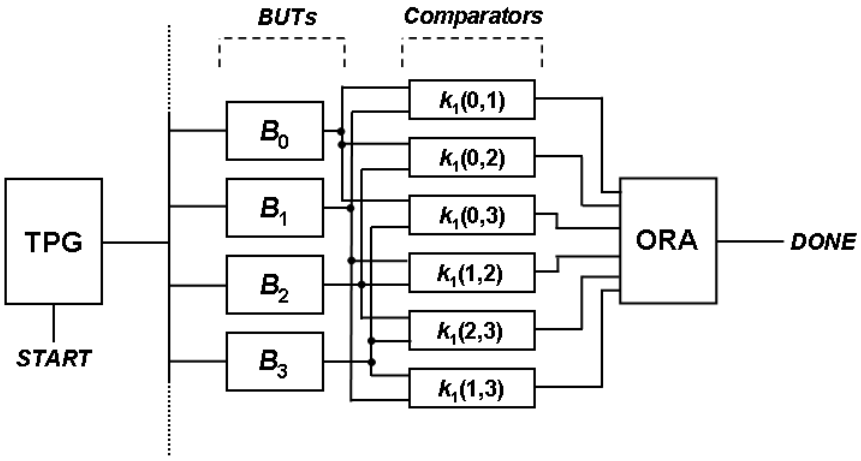


Figure 4.11: BIST Structure for Testing a Group of Four Blocks Under Test

The TPG is realized using an FSM to realize the states required for testing the embedded cores. In order to test the DSP48E cores, the FSM generates 400 states and 14-bit wide control signals for each state. The control signal bits are comprised of a 7-bit *opmode* signal, a 3-bit *carryin_sel* signal, and a 4-bit wide *alumode* signal. These serve as control

inputs to each of the DSP48E embedded cores. For each of the 400 states, the FSM generates valid combinations of these 14 control signals which define the function implemented on the DSP48E at any given clock signal. The FSM is optimized via XST into one 512x17 ROM and a 14-bit registered output. This ROM is realized as one of the embedded BRAM cores which is pre-defined through initialization. State-transitions are performed via a 9-bit adder, whose output is registered using a 9-bit register. The three data operands for the DSP48E cores are generated using one 18-bit *Linear Feedback Shift Register (LFSR)*, one 48-bit LFSR and one 30-bit LFSR.

Each pair of BUTs requires a 48-bit comparator and 4 1-bit comparators for their outputs to be compared for discrepancies. In addition to these, for each pair of BUTs, a 2×1 multiplexer is used to serialize the results of the comparators. Thus for every group of BUTs, a total of six 2×1 multiplexers are required. This circuitry is further optimized as described in the following section. Figure 2 shows these six comparators $\boldsymbol{k_1(i,j)}$ for comparing the outputs of the 4 BUTs in group $\boldsymbol{n} = 1$. technique uses a test controller in addition to the TPG and the ORA, to activate the test routine by asserting the START signal. Termination of the test is achieved when the DONE signal is asserted, followed by the propagation of the test results.

4.6.3. Embedded Core Fault Isolation Experiments on Virtex-5 FPGAs

As a particular example of the BIST technique, experiments were conducted on the Virtex-5 family of Xilinx FPGAs. The testing of an XC5VLX30 device provides the following case study which further elaborates the procedure. The XC5VLX30 device

consists of 36 DSP48E embedded cores, with 4800 slices that provide 19200 LUTs. The ***m*** = 36 embedded cores on the XC5VLX30 device are divided into ***n*** = 8 groups. Since six 2-to-1 multiplexers are required for each group, a total of 48 such multiplexers are required. However, the synthesized design optimally uses six 8-to-1 multiplexers.

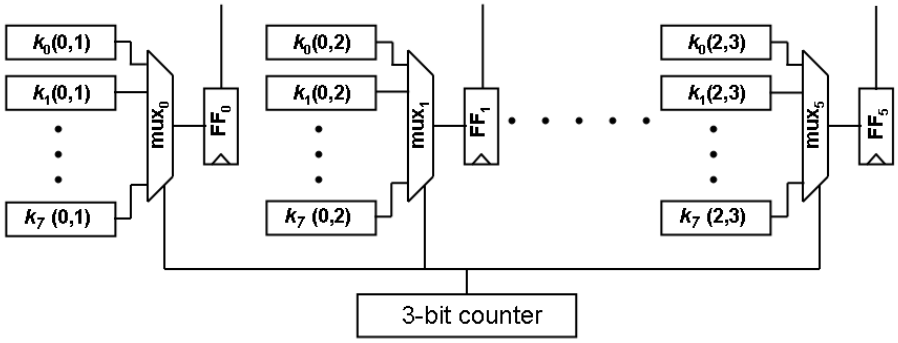


Figure 4.12: BIST Structure used for Testing the XC5VLX30 Device

A block diagram of the scheme is shown in Figure 4.11. As shown in the figure, a total of six multiplexers and flip flops, numbered $\boldsymbol{mux_o}$ through $\boldsymbol{mux_5}$ and $\boldsymbol{FF_o}$ through $\boldsymbol{FF_5}$ are utilized. There are six columns of comparators, with each column consisting of eight comparators, $\boldsymbol{k_0}$ through $\boldsymbol{k_7}$. Comparators $k_n(i,j)$, $0 \leq i,j \leq 3, \forall\ i \neq j$ complete the test for a group of four BUTs as shown in Figure 2. The results for comparisons among one group of BUTs, for example, the results from $k_0(0,1)$, $k_0(0,2)$, $k_0(0,3)$, $k_0(1,2)$, $k_0(1,3)$ and $k_0(2,3)$ are registered in the flip flops FF_0 through FF_5. This is then repeated for the other groups, using the 3-bit counter to enable the succeeding inputs of each multiplexer. Thus, at the end of each test, when the inputs from the TPG have been applied, the counter goes through all the multiplexer inputs and sending the output of the six flip flops

simultaneously to 6 1-bit outputs. The fault diagnosis script then processes the results of each set of 6 outputs to resolve the location of the defective BUTs. This can lead to isolation of faults in any two of the four BUTs in each group, irrespective of the location of the faulty BUTs within each group.

Table 4.4: Resource Utilization Results from Experiments Conducted on the Xilinx Virtex-5 Family of FPGAs

Device	DSP48E	Available Slices	Available LUTs	Available FFs	Resource Utilization under Test (Percentage)	
					LUTs	Flip flops
XC5VLX30	32	4800	19200	19200	1,418 (7%)	384 (2%)
XC5VLX50	48	7200	28800	28800	1862 (6%)	408 (1%)
XC5VLX85	48	12960	51840	51840	1862 (6%)	408 (1%)
XC5VLX110	64	17280	69120	69120	2300 (3%)	432 (1%)
XC5VLX155	128	24320	97280	97280	4058 (4%)	528 (1%)
XC5VLX220	128	34560	138240	138240	4058 (2%)	528 (1%)
XC5VLX330	192	51840	207360	207360	5822 (2%)	624 (1%)
XC5VSX35T	192	5440	21760	21760	5822 (26%)	624 (2%)
XC5VSX50T	288	8160	32640	32640	8462 (25%)	768 (2%)
XC5VSX95T	640	14720	58880	58880	18139 (30%)	1296 (2%)

The solution was implemented on various devices of the Virtex-5 family. Table 4.4 summarizes the resource usage for each of these devices. As listed in the Table, for the XC5VSX95T device, which contains 640 DSP48E embedded cores, the device utilization during testing is approximately 30%. In Table 4.4, all Utilization Percentage figures less than 1% have been rounded up to 1%. Also, each Slice in the Virtex-5 family of FPGAs contains four LUTs and four flip flops.

Embedded cores within FPGAs provide improved performance by optimizing area and power consumption. With improvements in the process technology, the smaller geometries will drive the inclusion of an increasing number of diverse hard IP blocks in FPGAs. As shown in this article, the XC5VSX95T device in the Virtex-5 family contains 640 DSP cores and 488 Block RAM cores. This shows the need for efficient fault isolation techniques to diagnose these devices to improve yields and facilitate faster debugging. The demonstrated technique achieves the goal of fast detection and isolation of faults by leveraging a group testing technique that isolates faulty embedded cores in a single-step procedure that precludes the need for device reconfiguration. The approach is scalable at the cost of area overhead. However, no permanent area cost or performance overheads are incurred as a result of testing. This technique can be used in conjunction with other existing methods for isolating faults in interconnect and CLBs to provide complete post-manufacturing testing for FPGAs with embedded cores.

4.7. Improving GA Performance Using CGT

The fault isolation provided by *Combinatorial Group Testing (CGT)* can be utilized to accelerate the design and repair process in a genetic algorithm. To demonstrate the benefit using an example, A *CGT-pruned GA* was developed [58] to evaluate the performance benefit obtained by using the halving testing scheme. As shown in Figure 4.13, the simulator for the CGT-Pruned GA optionally uses a seed configuration and uses the resource information provided by the CGT technique to effect refurbishment in faulty configurations using the GA. The simulator is a C++ based console application that

consists of two main components: the CGT procedure and the GA. The CGT algorithm uses the *Gnu Scientific Library* (GSL) and simulates the fault location method. The GA is implemented using an object oriented architecture that contains classes which model the FPGA resources with flexible geometries such as the *Configurable Logic Block* (CLB) and *Look-Up Table* (LUT) classes, and others that model the GA such as *Individual* and *Generation* classes. When this simulator is run in the *CGT-pruned GA mode*, the CGT component simulates the desired FPGA chip and obtains resource performance information which is an input to the GA. The GA then performs evolutionary design or reads the *Seed Configuration* file and performs evolutionary repair according to the active mode of operation. In the *Conventional GA mode*, the CGT component is not invoked and no resource performance information is available to the GA.

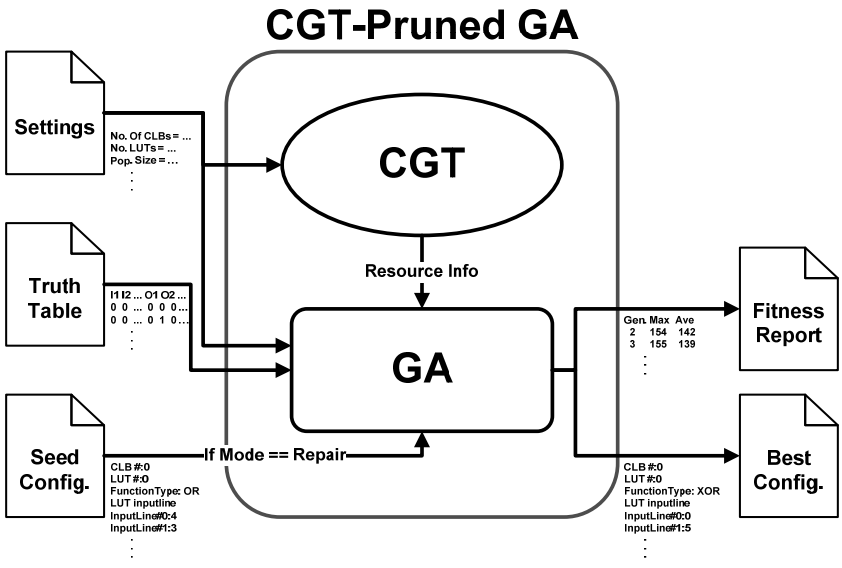


Figure 4.13: CGT-Pruned GA Simulator

Table 4.5: CGT-Pruned GA - Repair Performance

Experiment Type	**Conventional Repair**	**CGT-pruned Repair**
Circuit	3-bit x 2-bit Multiplier	3-bit x 2-bit Multiplier
Number of Experiments	30	30
Arithmetic Mean (Generations)	17150	10700
Standard Deviation	15650	12550
Standard Error of the Mean	2850	2300
68% Confidence Interval	[14300 → 20000]	[8400 → 13000]

In the experiments, a 3-bit × 2-bit multiplier is circuit evolved from seed configurations, and in the repair experiments, functional circuit representations with a simulated fault are repaired. The optimized GA parameters used were a mutation rate of 0.05, a crossover rate of 0.4, and a population size of 25. Further, elitism was imposed where the two best-fit configurations from a generation were propagated to the next generation. The simulated FPGA architecture consisted of 15 CLBs configured with a strict feed-forward topology. As listed in Table 4.5, with a single stuck-at fault, the CGT-pruned GA outperformed a GA unassisted by the results of group testing in the experiment concerning the repair of individuals affected by the fault. Over 30 trials, the CGT-pruned GA required an average of 10700 generations to realize a repair as opposed to 17150 generations for the non CGT-pruned GA. Further the result ranges do not overlap at the 68% confidence interval, which makes the result more statistically significant.

CHAPTER 5: LOGIC ELEMENT ISOLATION USING AUTONOMOUS GROUP TESTING

The logic resources on a Xilinx FPGA device are organized as a two-dimensional array of CLBs [59]. Each CLB consists of 4 *slices*, which in turn contain two 4-input LUTs. In the AGT-based fault isolation method described, a *logic resource* refers to a slice in the FPGA. As shown in Figure 5.1, the FPGA is seen as a two-dimensional array of resources, each resource being a slice. The fault model accounts for stuck-at faults at the inputs of one of the two LUTs in a slice specified by its (x,y) coordinate pair.

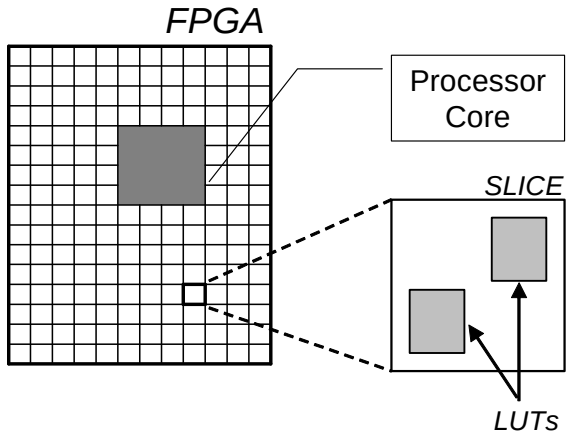


Figure 5.1: FPGA Resources as Seen by the Group Testing Algorithm

5.1. Terminology and Nomenclature for Analysis of Autonomous Group Testing Techniques

Let **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, **C**, consisting 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 *population preset* value p_{preset} determines the maximum number of individuals in a generation so that $p_{stage} \leq p_{preset}$ as testing progresses. At each *stage* of the adaptive testing algorithm, the configurations in the population are replaced by new designs, creating a new *generation* of individuals.

$\mathbf{T}$ denotes the set of binary input vectors applied and $t_i \in \mathbf{T}$ are the individual input vectors. These inputs to the implemented combinatorial logic are also the test vectors for the isolation procedure. Let the function implemented on the FPGA be denoted by $\boldsymbol{F}(\mathbf{T}, \mathbf{c_i})$. If any of the resources in $\mathbf{c_i}$ used to realize $\boldsymbol{F}(\mathbf{T}, \mathbf{c_i})$ are faulty, then the response will deviate from the correct realization, for some subset $\mathbf{T'} \subset \mathbf{T}$ which articulate the fault as follows:

Definition 5.1. The *syndrome* **T'** of a configuration $\mathbf{c_i}$ is the set of positive tests for the configuration.

Definition 5.2. 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, **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.

Definition 5.3. The *articulation rate* $a(\mathbf{c_i})$ for a configuration $\mathbf{c_i}$ is the ratio of the number of incorrect outputs to the cardinality of the entire output space:

$$\text{Articulation rate, } a(\mathbf{c_i}) = \frac{|T'|}{|T|}. \tag{5.1}$$

Since the articulation rate cannot controlled by the designer, it introduces randomness into the rate of progress of fault isolation as discussed in section 6.2. Fault isolation proceeds by reducing the cardinality of the set of *suspects*, **S. S** is defined as the intersection of resources $r_i(x,y) \in \mathbf{c_i}$ used by all $\mathbf{c_i}$ $\forall$ D(**T'**, $\mathbf{c_i}$) $\neq \{\}$. The set of all viable resources tenable to creating fault-free configurations is denoted by $\overline{\mathbf{S}}$, such that $\mathbf{S} \cup \overline{\mathbf{S}} = \mathbf{R}$.

Definition 5.4. *Forward Progress* is made, if, as fault isolation proceeds, |**S**| decreases and $|\overline{\mathbf{S}}|$ decreases, until finally |**S**| = *d,* the number of known defectives.

As fault isolation progresses |**S**| decreases and $|\overline{\mathbf{S}}|$ increases, until finally |**S**| = *d,* the number of known defectives.

Definition 5.5. The *defect scouring ratio, d(stage)* defines the ratio of number of known suspects |**S**| to $|\overline{\mathbf{S}}|$, given the number of test stages that have been completed:

$$d(stage) = \frac{|\mathbf{S}|}{|\overline{\mathbf{S}}|} \tag{5.2}$$

5.2. Autonomous Group Testing Algorithm Overview

As shown in Figure 5.2, the AGT algorithm comprises of three phases of fault isolation which occur after the fault has been detected. First, in the initialization phase, all elements of the History Matrix, H, described in Section 5.3, are initialized to zero. In addition, since the isolation procedure is yet to begin, the set of suspect resources, **S** is equal to the set of resources under test, **R**. After initialization, the p_{stage} configurations that comprise the first testing stage are created, which forms the second phase of the algorithm. The third phase consists of performing tests on the configuration thus created. Phases 2 and 3 are repeated until the defective resource is isolated.

Before the configurations for a stage are created in phase 2, the *equal share factor,* n_{share}, and the population size, p_{stage}, are determined as described in Sections 5.4 and 5.5, respectively. Once n_{share} is known, the p_{stage} individuals that comprise the first test stage are created using the *Fault Injection and Analysis Toolkit (FIAT)* described in Section 5.9. During the fault isolation phase shown in Figure 5.2, isolation proceeds by applying random test vectors which emulate the input data stream to randomly selected configurations that comprise the first test stage. This process continues until *stasis* is attained, as described in Section 5.6. After the system attains stasis, a new testing stage is created, and fault isolation is pursued until the defective resource is identified.

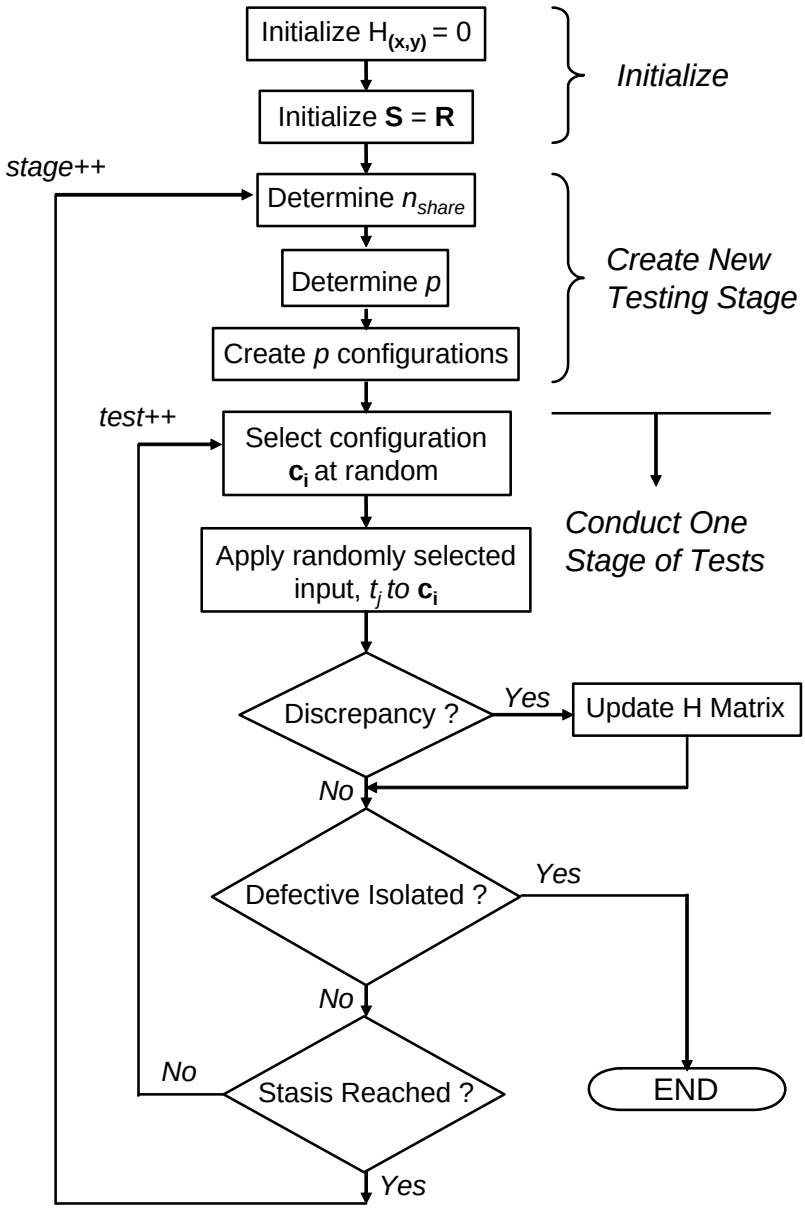


Figure 5.2: AGT Process Flow

5.3. Tracking Defectives Using the History Matrix

The history matrix, H, keeps track of the discrepancy counts of the resources. As shown in Figure 5.2, all elements in the H matrix are initialized to zero. As a stage of tests

proceeds, for each test t_i for which $\mathbf{D}(t_i, \mathbf{c_j}) \neq \{\}$, all H matrix entries $H_{(x,y)}$ are incremented by one where *(x,y)* are the coordinates of all $r_i(x,y) \in \mathbf{c_j}$. 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 defective resource will be identified by the coordinates of the maximal element in H.

5.4. The Equal Sharing Test Group Formation Strategy

Initially, **S = R**, since no information is available regarding the fitness of any of the resources. The algorithm proceeds in stages, with a new generation of individuals being created in each stage. In each stage, the members of **S** are equally shared among the configurations $\mathbf{c_i}$, $0 < i < p_{stage}-1$ in the generation.

The remaining n_{reqd} resources required to realize the design are randomly selected from the set $\overline{\mathbf{S}}$ which has a cardinality $|\mathbf{R}| - |\mathbf{S}|$. Thus each individual $\mathbf{c_i}$ will be allocated $|\mathbf{R}| - |\mathbf{S}| + |\mathbf{S}|/p_{stage}$ resources. Hence if the number of suspects $|\mathbf{S}|$ is small enough such that $|\mathbf{R}| - |\mathbf{S}| + |\mathbf{S}|/ p_{stage} > n_{reqd}$, then the configurations in that group will have mutually exclusive shares of the suspect resources, with each individual configuration $\mathbf{c_i}$ being allocated exclusive resources $r_j(x,y) \in \mathbf{c_i}$, $r_j(x,y) \notin \mathbf{c_k}$, where $0 < k < p_{stage}-1$. Otherwise, some suspect resources need to be shared among the configurations to meet the application resource demand n_{reqd}. The maximum cardinality of $|\mathbf{S}|$ such that mutually exclusive shares of suspect resources are possible, denoted by $|\mathbf{S_{max}}|$ can be obtained by evaluating the following expression:

$$| R | - | S_{\max} | + \frac{| S_{\max} |}{p_{preset}} = n_{reqd}\,, \tag{5.3}$$

which yields:

$$| S_{\max} | = \frac{p_{preset}}{(1 - p_{preset})} \times (n_{reqd} - | R |) \tag{5.4}$$

If $|\mathbf{S}| > |\mathbf{S_{max}}|$ then the equal share factor, n_{share}, is derived by rearranging Equation(5.5) to yield Equation(6.6):

$$n_{reqd} = | R | - | S | + n_{share} \tag{5.5}$$

$$n_{share} = n_{reqd} - | R | + | S | \tag{5.6}$$

Figure 5.3 shows an example of how $|\mathbf{S}| = 30$ suspect resources from among $|\mathbf{R}| = 100$ resources are shared among $p_{stage} = 2$ configurations. In case 1, $n_{reqd} = 85$, yielding $|\mathbf{S_{max}}| = 30$ using Equation(6.4). Since $|\mathbf{S}| = |\mathbf{S_{max}}|$ in this scenario, configurations $\mathbf{c_0}$ and $\mathbf{c_1}$ use mutually exclusive subsets of **S**, and they both use all $r_i(x,y) \in \overline{\mathbf{S}}$ to satisfy the application resource demand. In scenario 2, however, $n_{reqd} = 91$, and thus, $|\mathbf{S_{max}}| = 18$. Since $|\mathbf{S}| > |\mathbf{S_{max}}|$, the equal share factor is evaluated using Equation(6.6) to be $n_{share} = 21$. As shown for case 2 in Figure 5.3, $\mathbf{c_0}$ and $\mathbf{c_1}$ share $|\mathbf{S}| - 2 \times n_{share} = 12$ suspect resources.

Alternative resource allocation strategies can be adopted to replace the equal share strategy. For instance, in [60], an *Interleaved Allocation* strategy is proposed that ensures that each LUT in the *Suspect* pool is used by more than one individual in every new stage. This will reduce the probability that a faulty LUT does not articulate the fault for the observed test vectors. The strategy uses a *Coverage Factor* to determine the number of different individuals that utilize any suspect resource. The interleaved allocation scheme adopts a low-risk approach by covering each suspected resource with two or more configurations, making it less probable that a group of testing yields no improvement.

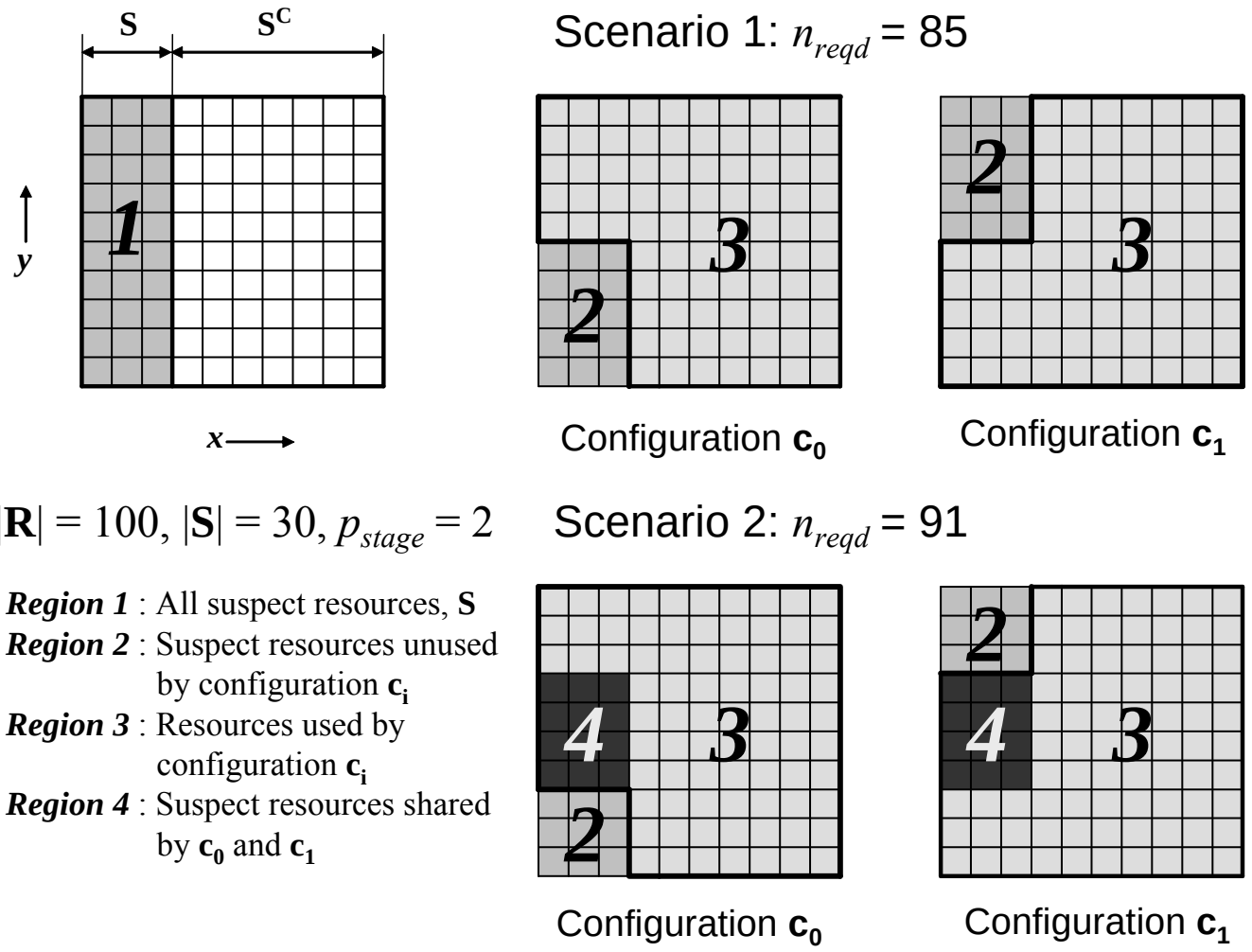


Figure 5.3: Sharing the Suspect Resources Equally – Two Different Scenarios

5.5. Adapting the Population Size for Optimal Resource Coverage

In order to reduce the number of individuals under test, the population size is adapted. For example, if in the final stage of testing, $|\mathbf{S}| = 3$ even though the p_{preset} may be greater than 3, only 3 individuals, each using one of the suspect resources is required for isolation to complete. Such a situation occurs frequently in the beginning of the isolation process. For example, with a resource redundancy ratio, $rr = 0.5$, in the first stage, only two individuals are required to cover the entire resource space. Additional individuals will only form tests for resources that are already covered by these two, and will thus be redundant. The number of individuals required in any stage of testing is given by:

$$p_{stage} = \left\lceil \frac{|S|}{n_{\text{share}}} \right\rceil \tag{5.7}$$

Reducing the number of individuals in a test stage provides two benefits. First, it significantly reduces the time required for the fault isolation process. Secondly, it reduces the number of redundant test groups – making the algorithm more reasonable. In particular, a *reasonable* group testing procedure is one that contains no test whose outcome can be predicted from outcomes of other tests conducted previously [12].

Once n_{share} and p_{stage} are known, the individuals for a given generation are created, and then tested. As shown in Figure 5.2, testing comprises the third phase of the isolation process. The tests are conducted by randomly selecting an individual $\mathbf{c_i}$ for instantiation on the FPGA. A test vector t_j is then applied to the individual. If $\mathbf{D}(t_i, \mathbf{c_j}) \neq \{\}$, all H

matrix entries $H_{(x,y)}$ are incremented by one where (x, y) are the coordinates of all $r_i(x,y) \in \mathbf{c_j}$. Regardless of whether there is a discrepancy, this configuration is then replaced by another, and the testing continues. When a configuration containing the defective resource is tested, the probability of the fault being expressed as a discrepant output is governed by the articulation rate, $a(c_i)$, of that configuration. Once a fault is articulated, the set of suspects will be reduced to the intersection of the resources utilized by $\mathbf{c_j}$ and the resources $H_{(x,y)} = H_{max}$. Thus:

$$S_{new} = \mathbf{c_j} \cap \mathbf{H} \tag{5.8}$$

where h_{max} is the maximal element in the history matrix $H_{(x,y)}$

5.6. Overcoming Stasis During Isolation

A state of *Stasis* is encountered in a stage of the isolation procedure if further tests on configurations comprising the stage are expected to lead to no significant reduction in the number of suspect resources. By Definition 6.4, stasis occurs when forward progress stalls. Defining a method to overcome stasis is essential to ensure fast fault isolation.

Since the suspect resources were equally shared among the individuals in the population, the maximum possible reduction in |S| is given by:

$$\frac{|S|}{|S_{new}|} = n_{share} \tag{5.9}$$

Once $|S_{new}|$ is obtained, the system is defined to have entered a state of stasis, when further improvements to the defect scouring ratio, *d(stage)*, have stalled. Further reduction in $|\mathbf{S}|$, beyond those described in Equation(6.9) is only possible if there exists another individual in the same generation that also utilizes the defective resource. Since such an individual is not guaranteed to exist and to articulate the fault, stasis is declared after the suspect pool is reduced by the factor shown in Equation(6.9). Stasis can also occur when the individual utilizing the defective resource does not articulate the fault, or does so with a very low articulation rate. Thus, stasis occurs when no discrepant outputs are observed after a fixed number of inputs are applied.

5.7. Walkthrough of Isolation Process

As an example of the isolation process, consider a situation where there is one defective resource with the coordinates (1,8) in a set of $\mathbf{R}$ = 100 resources, where $r_i(x,y) \in \mathbf{R}$, $0 < x,y < 9$. For simplicity, let us assume that a configuration that utilizes the defective resource always articulates the fault at the output. The number of resources required to implement the application is $n_{reqd} = 35$. Thus, for the first stage, by Equation(6.6):

$$n_{share} = 35 - 100 + 100 = 35 \tag{5.10}$$

For $n_{share} = 35$, and population preset, $p_{preset} = 5$, by Equation(6.7), we have:

$$\text{Population size for the first stage , } p_1 = \left\lceil \frac{100}{35} \right\rceil = 3 \tag{5.11}$$

Thus, in the first stage, there are three configurations, the first, $\mathbf{c_0}$ uses resources $r_i(x,y)$ where $0 < x < 4$, $0 < y < 5$, i.e., 35 resources with coordinates (0,0) through (3,4) inclusive; $\mathbf{c_1}$ uses resources with coordinates (3,5) through (6,9) and $\mathbf{c_2}$ uses resources (7,0) through (9,9) and (0,0) through (0,4) inclusive. Over a period, each of these three configurations are chosen at random and inputs are applied, until a discrepancy is observed. Since the defective resource (1,8) is used by $\mathbf{c_0}$, this configuration will articulate the fault. When this occurs, the H matrix entries corresponding to the resources with coordinates (0,0) through (3,4) used by $\mathbf{c_0}$ will be incremented by one. The set of suspect resources **S** now has a cardinality of 35, and contains the resources used by $\mathbf{c_0}$. After this first discrepant output, the cardinality of $\overline{\mathbf{S}}$ exceeds the critical cardinality of 35. Also, the prime realization for this experiment is 1, since $\mathbf{c_1}$ is known fault-free after $\mathbf{c_0}$ is identified as the discrepant configuration.

As the set of suspects has diminished by a factor equal to n_{share}, the next stage of configurations is formed. The number of suspects can be divided equally among the members of this new stage, thus, each new configuration will contain 35/5 = 7 suspect resources. The rest of the resources to create the 5 configurations are chosen at random from the 100-35 = 65 members of $\overline{\mathbf{S}}$. Thus, in the second stage of testing, $\mathbf{c_0}$ will use the suspect resources with coordinates (0,0) through (0,6) and the other configurations will use 7 suspect resources each, in order. The defective resource with coordinates (1,8) will be utilized by configuration $\mathbf{c_2}$. In the tests performed in the second stage, $\mathbf{c_2}$ will articulate a fault, and H matrix entries corresponding to resources with coordinates (1,4) through (2,0) inclusive will be incremented by one.

In the next stage, only four configurations need be created, with the first three configurations utilizing two of the seven suspect resources. This stage will further reduce **S** to two suspect resources. Finally in the last stage of testing, only two configurations will be created, with the first using the resource with coordinates (1,8) and the second utilizing the resource with coordinates (1,9). Tests on these two configurations will finally yield (1,8) as the defective resource. Thus, in four stages, the defective resource will be uniquely identified.

5.8. The Fault Isolation and Analysis Toolkit for Xilinx FPGAs

The UCF *Fault Injection and Analysis Toolkit (FIAT)* is a set of Python APIs that aid the analysis of fault-testing algorithms for Xilinx FPGAs. Faults are injected in the implemented designs by editing the design file. This precludes the need to edit the configuration bitstream directly. The Xilinx ISE design tools are used in the process flow to place and route the edited designs. FIAT can be used to model and evaluate various testing regimens that seek to identify and isolate faults in FPGAs. The toolkit enables easy injection of faults without directly modifying the bitstream. The principle of interfering minimally with the functions of the Xilinx ISE is adopted to reduce accidental bitstream errors that may invalidate the design or even damage the FPGA. The generation of post-place-and-route simulation executables offers a fast and reliable way of analyzing test routines without the additional expense of downloading the designs and reconfiguring FPGAs.

FIAT provides the following functions to enable the modeling and evaluation of group testing regimen:

get_list_slices_used(*proj_path, xdl_fn*): This method takes the project path(*proj_path*) and the xdl filename(*xdl_fn*) as inputs and returns a list of slices used by the design specified by the xdl file.

get_slice_count(*proj_path, xdl_fn*): Returns an integer representing the number of slices used by the specified xdl file. Inputs are the xdl filename(*xdl_fn*) and the project path(*proj_path*).

is_slice_used(*proj_path, xdl_fn, x, y*): Returns a Boolean value corresponding to whether or not a slice specified by its *x* and *y* coordinates is utilized by the design specified by the xdl filename(*xdl_fn*).

disclists(*list1, list2*): This function returns an integer representing the number of discrepancies observed in comparing the two lists *list1* and *list2*.

modify_xdl(*proj_path, xdl_fn, slice_coords, g_or_f, faulty_pin*): The *modify_xdl f*unction inserts stuck-at faults in a specified design. The slice where the stuck-at-fault is to be inserted is specified by a coordinate pair(*slice_coords*). The G or F LUT in the slice can be chosen using the *g_or_f* parameter. *faulty_pin* specifies the pin in the LUT where the stuck-at-fault needs to be injected.

create_ucf(*proj_path, ucf_fn, occ_area_xy, f_max_x, f_max_y, occ_area, req_resources, pop*): This method creates UCF files and placed-and-routed designs according to the specified parameters to aid the physical placement of the design on the FPGA. The *proj_path* and *ucf_fn* parameters define the project path and the UCF filename. *occ_area_xy* specifies the area that the design has to be placed in as a coordinate pair. All resources outside the square area defined by *occ_area_xy* are prohibited from being used in the design. *f_max_x* and *f_max_y* specify the maximum value of the x and y coordinates for resources that can be utilized by the design. *occ_area* defines the number of resources available for use by the design and *req_resources* defines the number of resources that are essential for instantiating the design. This figure can be ascertained by assessing the minimum number of slices required by the design. The *pop* parameter defines the population size, or the number of unique designs that need to be produced. The *create_ucf* function creates *pop* number of unique ucf files where the resources used are chosen at random. The amount of slices available for implementing the design can be varied from (*occ_area – req_resources)* to *occ_area* depending on the needs of the test routine by using custom functions to determine the selection of resources. After creating the UCF files, the create_ucf function proceeds to create the NCD files for the designs and converts the NCD files to XDL files which can then be modified according to need using the modify_xdl function.

simulate_ppr(*proj_path, sim_fn*): The *simulate_ppr* function accepts the project path and the name of the simulation executable as inputs. It invokes the Xilinx commands to compile the HDL design, testbench files, and simulation libraries to create the simulation

executables. Finally it runs the simulation. This is useful in conjunction with testbench files that save the output of the simulation in the form of text files.

5.9. Creating and Modifying Alternatives with FIAT

FIAT provides a high level of control over the physical location of the logical units used in the design. In particular FIAT provides methods for modifying and parsing the *User Constraint File (UCF)* and the *Xilinx Design Language (XDL)* file. The XDL file is a plain text file that can be created from the NCD file using the *xdl* command line tool provided by Xilinx. Throughout the design flow, the Xilinx ISE tools are used all processes except for those that edit and parse the UCF and XDL files. The tools provided by FIAT can be used for determining the physical placement of the logical units by editing the UCF file. Stuck-at faults are injected into the design by converting the NCD file to the XDL format and then using the FIAT APIs to insert the fault at the chosen LUT. The presence of a stuck-at fault ties the signal at the input of the chosen LUT in a slice to zero or one. After fault injection, the XDL file is converted back to an NCD file. Placement and routing is completed automatically using the Xilinx ISE. The post-place-and-route simulation executable is created using the provide testbench and the simulation libraries.

Figure 5.4 shows the processes that constitute the FIAT design flow. The input files for the process are the HDL files specifying the combinatorial design to be instantiated on the FPGA. These files are synthesized to build a netlist, which FIAT then builds, maps, places and routes using commands provided by the Xilinx ISE 9.1i tools. In the last step

a post-place-and-route simulation executable is created using the user-provided testbench and the simulation libraries. The same *Native Circuit Description (NCD)* file used to create the simulation executable can also yield the configuration bitstream for a hardware implementation of the design. The generation of post-place-and-route simulation executables offers a flexible and accurate way of analyzing test routines. In addition to providing methods to implement designs using the Xilinx commands, FIAT provides automated methods to edit physical constraints and to inject faults into configuration bitstreams.

FIAT provides a high-level of control over the physical location of the slices used to create a configuration by providing APIs to modify the *User Constraint File (UCF)*. This enables editing configurations before they are placed and routed. Given a set of suspect resources to be used by each configuration, FIAT creates the UCF files to ensure the use of the suspect resources. It then invokes the Xilinx place-and-route tool provided in the ISE 9.1i platform to realize the designs required by the AGT.

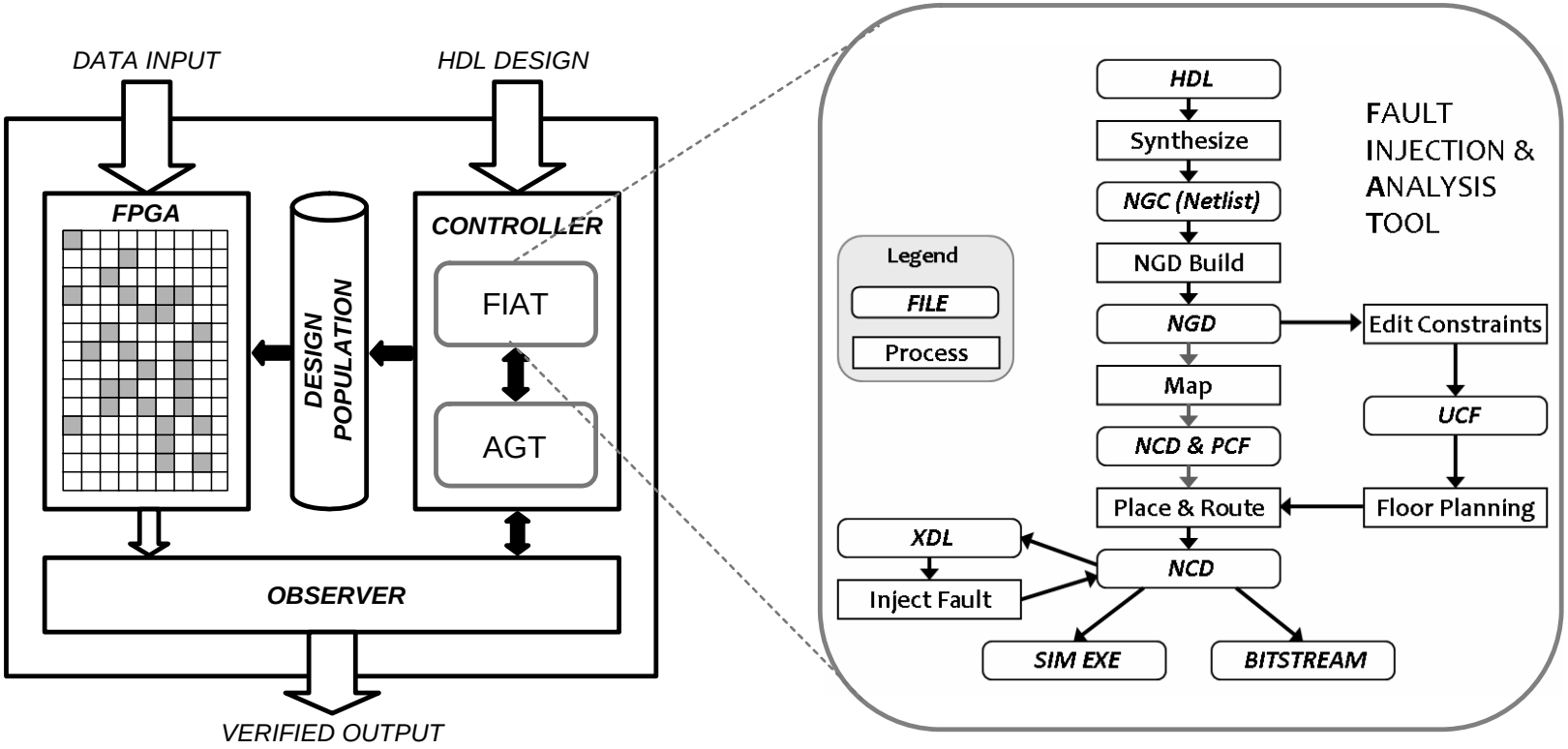


Figure 5.4: Fault Isolation Using FIAT – An Overview

Since it is not viable to destructively modify the FPGA hardware resources, stuck-at faults need to be simulated in the configurations to enable analysis of the AGT algorithm. Stuck-at faults are simulated in the experiments by editing all configurations to exhibit behavior consistent with the presence of a stuck-at fault at one of the input pins of a specified LUT. To inject the fault, FIAT converts the NCD file, which describes the placed-and-routed design, to the *Xilinx Description Language (XDL)* format using the *xdl* command line tool provided by Xilinx. This text file is then edited to modify the logic function instantiated on the target fault-affected LUT. The presence of a stuck-at fault ties the signal at the input of the fault-affected LUT to zero or one. After fault injection, the XDL file is converted back to an NCD file. Placement and routing is then completed automatically using the Xilinx tools included in the ISE 9.1i suite.

FIAT precludes the need to edit the configuration bitstream directly. Throughout the design flow, the Xilinx 9.1i ISE tools are used for all processes except for those that parse and edit the UCF and XDL files. The Xilinx design tools, such as *netgen*, *par*, *ngdbuild,* and *fuse* are invoked by FIAT in the design flow to place and route the edited designs. This principle of interfering minimally with the functions of the Xilinx ISE reduces accidental bitstream errors that may invalidate the design or irrecoverably damage the FPGA.

CHAPTER 6: CHARACTERISTICS, CAPABILITIES, AND METRICS FOR SUSTAINABILITY

Experiments on the AGT algorithms were conducted using post-place-and-route designs created for the Xilinx Virtex-II Pro FPGA. A 56-bit *Data Encryption Standard (DES-56)* encryption/decryption implementation was used in generating the data. Sections 6.1, 6.2, and 6.3 present results from these experiments with regards to the efficacy and the impact of system parameters on the algorithm.

6.1. Experimental Configuration for the Xilinx Virtex II Pro FPGA

The AGT, together with FIAT, implements the controller for autonomous fault handling. As shown is Figure 5.4, this controller receives observed feedback and updates the design population across stages. FIAT has been constructed as part of the work presented using the Python programming language to provide APIs to edit resource constraints, introduce stuck-at-faults, and generate post-place-and-route designs, as described previously in Section 5.9.

Experiments were conducted on a Virtex-II Pro FPGA xc2vp4-7ff672 model using the Xilinx ISE 9.1i design platform. The 7ff672 package provides 3008 slices and 348 *Input-Output Blocks (IOBs)*.

To analyze performance of the algorithm, the following characteristics are defined by the functionality of the application implemented on the FPGA:

Definition 6.1. The *application resource demand,* n_{reqd} is the minimal cardinality of any design configuration $|\mathbf{c_i}|$, required to implement the application on the FPGA.

Definition 6.2. The *resource redundancy* ratio, *rr* is defined as the ratio of the application resource demand to the cardinality of the set of all resources $|\mathbf{R}|$

$$rr = \frac{n_{reqd}}{|R|} \tag{6.1}$$

Definition 6.3. The *critical cardinality* is the cardinality of $|\mathbf{S}^C|$ such that $|\mathbf{S}^C| = n_{reqd}$**.**

Definition 6.4. The *prime realization* is the index *i,* of the first identified subset $\mathbf{c_i}$, which satisfies the two conditions: $\mathbf{c_i} \subset \overline{\mathbf{S}}$ and $|\mathbf{c_i}| \geq n_{reqd}$.

Let:

p be the population size

R be the total number of resources

T be the total number of tests to exhaustively test the configurations

A be the mean articulation rate of the population,

and

ρ be the *fault articulation rate* for a configuration, defined as follows:

$$\rho = rr \,.A \tag{6.2}$$

Additionally, the probability that a given configuration is affected by a single random fault in any of the R resources is given by the resource redundancy ratio ρ.

Since the tests are independent of each other, and the results of the random tests follow a binomial distribution, the probability that exactly n faults are observed in S tests is given by:

$$\mathrm{P(n)} = \binom{S}{n} \rho^n (1-\rho)^{S-n} \tag{6.3}$$

Let the outcome x be defined as the number of the *successes* identified in S tests. A *success* is when a fault is observed. The *cumulative distribution function (cdf)* denoted as F(X) describes the probability that the outcome $x \leq \mathrm{X}$.

The cdf for x successes in S tests, where the probability of success is ρ, is given by:

$$F(x; S, \rho) = P(X \leq x) = \sum_{x_i \leq x} P(x_i) \tag{6.4}$$

Various methods to approximate bounds for the cdf exist, notably when $x < S\rho$, the *Hoeffding's inequality* yields the upper bound:

$$F(x;S,\rho) \le \exp\left(-\frac{1}{2\rho}\frac{(S\rho - x)^2}{S}\right) \tag{6.5}$$

Then, from Equation (7.2), the probability that a certain configuration is observed as being faulty at least once over S tests on the population is given by the *complementary cumulative distribution function (ccdf)*, given by

$$\mathrm{P}(X \ge x) = 1 - F(x;S,\rho) \tag{6.6}$$

Of particular interest is the probability that a particular configuration is observed as being faulty at least once after S tests. This probability can be calculated by noting that the probability that a certain configuration is selected for testing is $(1/p)$. This modifies the probability for success to (ρ/p) as compared to ρ earlier.

The probability that a particular configuration is observed as being faulty at least once after S tests is therefore given by:

$$\mathrm{P}(X \ge 1) = 1 - F(1;S,(\rho/p)) \tag{6.7}$$

and the upper bound is approximated as:

$$= F(1;S,\frac{\rho}{p}) \le \exp\left(-\frac{1}{2\frac{\rho}{p}}\frac{(S\frac{\rho}{p}-1)^2}{S}\right) \tag{6.8}$$

Let μ_n be the mean number of tests for n different configurations to be identified as faulty.

The mean number of tests before one configuration is identified as faulty is the mean of the binomial distribution, defined as:

$$\mu_1 = S\rho \tag{6.9}$$

The mean of the number of tests required to identify another configuration as being faulty is the sum of the mean time taken for one configuration to be identified as faulty, and the mean of the number of tests where another configuration is paired with the faulty configuration, or, itself articulates the fault, therefore:

$$\mu_2 = \mu_1 + S\left(\rho + \frac{\rho(p-1)}{\binom{p}{2}}\right) \tag{6.10}$$

Further, since comparing a configuration to a faulty configuration will result in the configuration being marked as *Suspect,* the probability that all the configurations are marked as faulty is given by the probability that a faulty configuration is chosen, and all the other configurations are chosen in turn to be paired with the faulty configuration.

The DES-56 implementation utilizes 304 slices and 191 bonded IOBs. Thus, for the fault isolation experiments, the application resource demand, n_{reqd} = 304. The total gate-equivalent count for the design is 5266. The area under test on the FPGA can be varied

by controlling the total resources, **R**, available for placing and routing the design. This enables varying *rr* for the experiments. Initially, the DES-56 core was synthesized, mapped, placed, and routed on the FPGA. This model was later modified using FIAT according to the requirements of the AGT to form configurations and test stages. For each of these configurations, a simulation executable was created using a testbench. The inputs for the DES-56 circuits were obtained from the National Bureau of Standards publication 500-20 [61]. These inputs comprehensively test the functionality of hardware implementations of DES-56. Sixty of these inputs, representing a cross-section of the NBS test suite were used to create the test bench.

6.2. Isolation Progress Across Test Stages in AGT

Figure 6.1 shows the progress of defect isolation across various stages for p_{preset}= 5 for three different experimental runs. The best performance is seen in experiment 1, where fault isolation is completed using 5 stages of tests. A total of 21 different configurations were created to identify the single defective resource. In the first test stage, three individuals were created, one of which utilized the fault-affected resource and articulated the fault. Thus, at the end of stage 1, the number of suspect resources drops from 625 to 304. The two individuals in stage 1 that do not utilize the defective slice are the prime realizations of the circuit which can provide fault-free implementations on demand. Also, by the end of stage 1, $|\overline{\mathbf{S}}| = 625–304 = 314 > n_{reqd}$, and thus, critical cardinality is met. In stages 2, 3, and 4, five configurations each are created, as $p_{stage} = p_{preset} = 5$. Since the equal sharing method is used to create the configurations in each group, the

number of suspect resources decreases by a factor of $\left\lceil \frac{|S|}{p_{stage}} \right\rceil$ in each stage. In the final stage, since |**S**| = 3, only three configurations are created. The number of discrepant outputs in all the tests is equal to the number of test stages since at the occurrence of the first discrepant output, the creation of a new group of configurations is initiated.

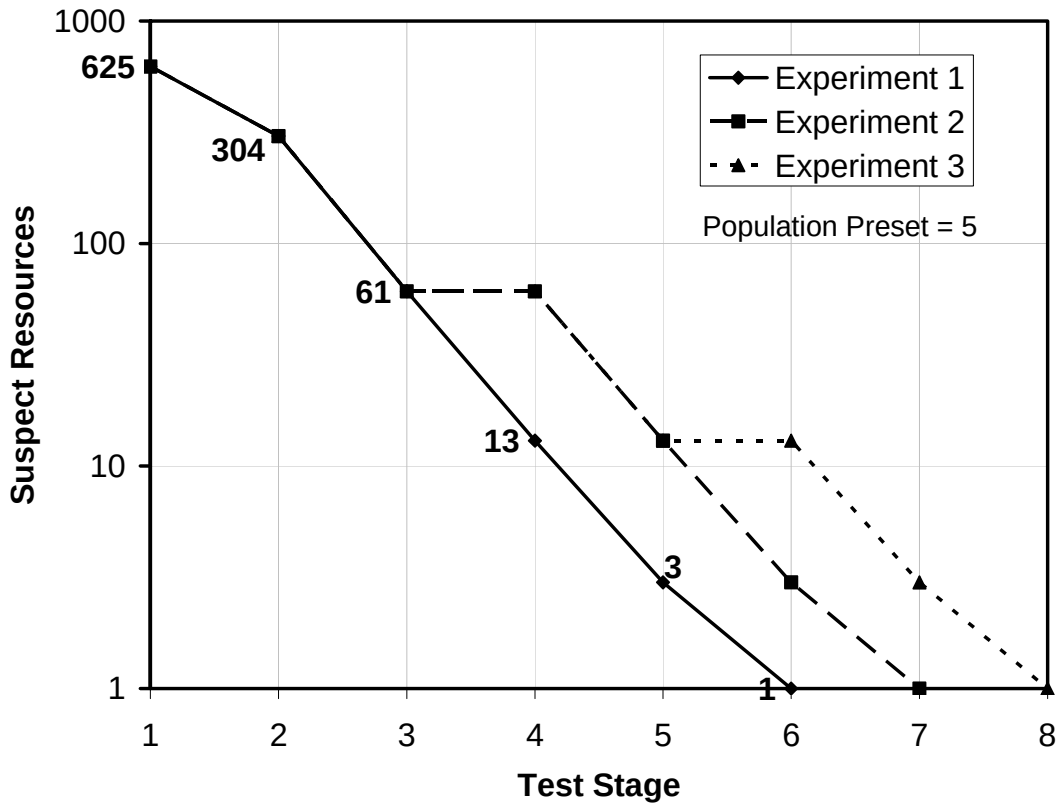


Figure 6.1: Fault Isolation Progress Across Stages for p_{preset} = 5

As shown in Figure 6.1, in the Experiment 2, no progress is made in the third stage of testing, where the number of suspect resources remains at 61. This is due to the fact that the individual utilizing the fault-affected resource does not articulate the fault, leading to a stasis in the system. In stage 4, five new individuals replace the configurations in the

population. In this stage, the configuration with the faulty resource articulates the fault, leading to a decrease in the number of suspects. Similarly, in the third experiment, stasis occurs in the fifth stage. This increases the number of stages to isolate the fault and the total number of configurations created.

In the best performing experiment, five stages were required, and five tests with discrepant outputs are observed before the defect is isolated. Even in the worst case, with a test stage containing configurations that do not articulate the fault, only five discrepancies are observed. Non-articulating individuals that use the faulty resource increase the time taken to scour the defects, but do not affect the observed goodput. In addition, in all these case, since $rr < 0.5$, the prime realization, as well as a non-suspect set of resources with a cardinality greater than the critical cardinality are obtained after the first discrepant test output.

6.3. Effect of Population Preset on Defect Scouring Rate

The scouring rate is directly proportional to the population preset, p_{preset}. Table 6.1 lists the observed defect scouring performance for varying values of p_{preset}. A total of 15 experiments were conducted for each value of p_{preset}. The physical logical resource overhead for the AGT-based technique can be varied by adjusting the resource redundancy ratio, *rr*. In all these experiments, initially, $|\mathbf{R}| = 625$. This value was chosen as $25^2 = 625$ yields a redundancy ratio $rr = 304/625 = 0.49 \approx 0.5$. As the column labeled M2 in Table II indicates, throughout the experiments, a subset of non-suspect resources, with cardinality $\overline{\mathbf{S}} > n_{reqd}$, is identified after the first stage of testing. Similarly, from the

results for metric M3 in Table 6.1, it is shown that the prime realization, which provides a fault-free replacement configuration, is consistently identified from within the first group of configurations. The number of discrepant outputs, or positive tests required to isolate the fault is the same as the number of stages, since the articulation of a fault will immediately improve the scouring rate and trigger formation of the next stage of tests.

Table 6.1: Results from Experiments With Varying Population Preset Values

p_{preset}	**Fault Resolution Metrics***			**Number of Stages**			**Number of Configurations**		
	M1	M2	M3	Best	Worst	Mean	Best	Worst	Mean
5	5	1	1	5	7	5.53	21	31	23.67
10	4	1	1	4	5	4.27	27	37	29.67
15	3	1	1	3	4	3.20	35	38	35.47
20	3	1	1	3	4	3.13	39	59	42.73
25	3	1	1	3	4	3.13	41	66	44.27

* Fault Resolution Metrics:
M1: Number of observed discrepant outputs before the defective resource is isolated.
M2: Number of stages required to surpass critical cardinality for $\overline{\mathbf{s}}$.
M3: Number of stages required to identify the prime realization.

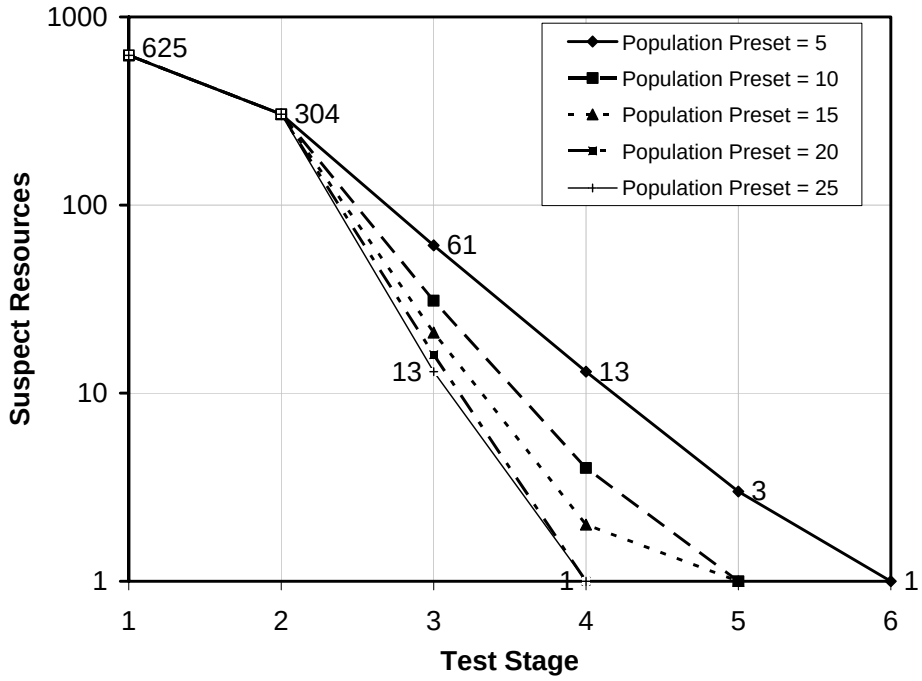


Figure 6.2: Effect of Population Preset on the Scouring Rate

Figure 6.2 shows the best defect scouring performance of AGT for increasing values of p_{preset}. Each curve depicts the size of the suspect pool, |**S**|, at the beginning of the test stage depicted on the x-axis. For all values of p_{preset} , population size, $p_{stage} = 3$ in the first stage of testing, by Equation(6.7). In all other stages except the last stage, $p_{stage} = p_{preset}$. In the last stage, p_{stage} is equal to the number of remaining suspect resources. The slope of the curve is proportional to the defect scouring ratio, and it increases proportionately with $p_{preset.}$ Except in the initial and last stages, defect scouring proceeds at a logarithmic rate, when the articulation rate for the configuration utilizing the defective resource is non-zero. Most significantly, across all values of p_{preset} the defective is isolated with 5 or fewer positive tests. Assuming that the time taken to reconfigure the device is insignificant when compared to the mean time between defects, the AGT-based method

can tolerate faults with minimal loss of goodput, with $p_{preset} = 5$, which will require the minimal number of reconfigurations.

The total number of configurations created in each of the five best performing experiments are shown in Figure 6.3. As p_{preset} increases, the total number of configurations increases. However, there is only two extra configurations are required for $p_{preset} = 25$ as opposed to $p_{preset} = 20$. Figure 6.3 also shows the number of test stages as a function of p_{preset}. With increasing p_{preset}, each stage reduces the number of suspects by a factor proportional to the population size. Thus, with increasing p_{preset}, though a decreased number of stages are required, the total number of configurations required is increased.

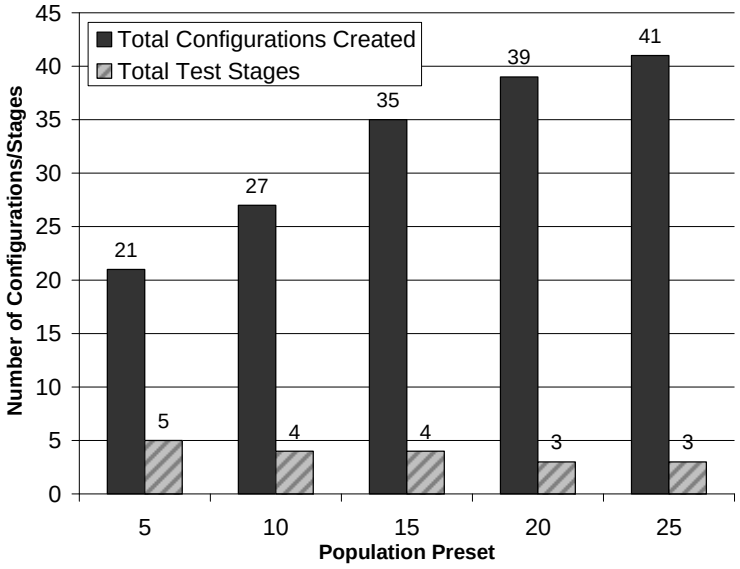


Figure 6.3: Total Test Stages and Configurations Created for Varying Population Presets

6.4. Maintaining System Throughput During Fault Isolation

System goodput, defined as the percentage of useful outputs, can be maintained at a pre-defined level throughout the fault isolation process using a feedback mechanism and an observer-controller model. The system goodput decreases each time there is a discrepant output – fault isolation will proceed faster with more frequent discrepancies. Thus, the tradeoff involved in maintaining goodput is that fault isolation will proceed at a slower rate.

Figure 6.4 shows the observed goodput as a function of the number of tests completed for three different values of required goodput throughout the fault isolation process. In all three experiments, the value used for the population present, $p_{preset} = 5$. In the first experiment, the system-level goal is to maintain a goodput of 0.99. A discrepant output is observed in the first ten tests, leading to a goodput of 0.90. Since this is lower than the performance goal, the system responds by utilizing the fault-free configuration until the goodput is restored to 0.99 by the hundredth test. Afterwards, the next stage of testing proceeds. When the fault-affected configuration in the second stage articulates the fault, the goodput drops to 0.982 by the 110^{th} test. Again, the system waits for the goodput to return to 0.99 before proceeding with conducting the third stage of tests. After 500 tests, after five positive tests, fault isolation is complete. The observed goodput will then continue to rise past 99%. In the second and third experiments, the goodput requirement is 0.95, and 0.90 respectively. As seen in Figure 6.4, for experiment 3, the system goodput never falls below 90% throughout the isolation process. After 10 tests, the goodput falls to 0.90 and rises subsequently until the fault is isolated in 320 tests.

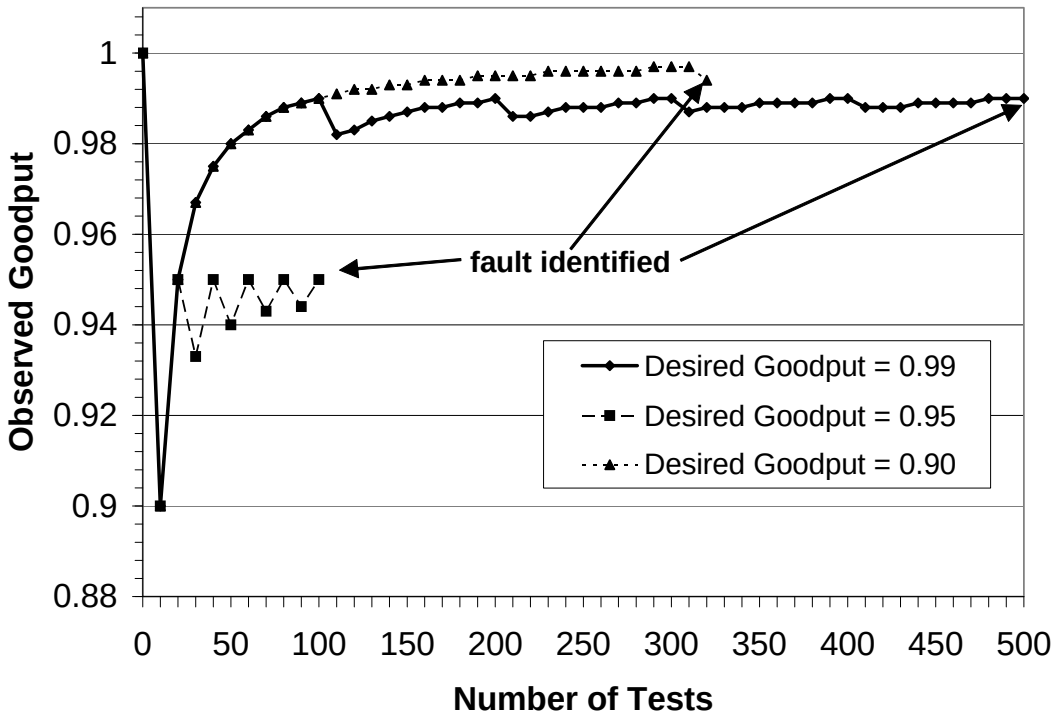


Figure 6.4: System Goodput Vs. Total Number of Tests

The time taken to create the configurations and reconfigure the FPGA is not reflected in the system goodput measurement. The goodput measured here is that of only the AGT-controlled system. Since AGT verifies correct functional behavior using output response analysis, it is essential to have an identical fault-free implementation of the same functionality, which would provide the correct outputs to which the outputs of the AGT-monitored configurations can be compared. Under a single-fault assumption, when the portion of the FPGA monitored by the observer is being reconfigured, the system outputs are provided by the other fault-free configuration.

Overall, the AGT-based autonomous method can isolate the single defective with a minimal number of positive tests, as low as 3, as listed in Table 6.1. This result is made

even more significant by the fact that this method avoids the use of exhaustive serial test procedures. Of all the previous approaches in Table 2.1, the roving STARS approach is the only comprehensive fault tolerance solution that isolates defects at a granularity lower than 1% of the total resources on an FPGA. Compared to this approach, the AGT-based technique has a minimal fault detection latency, and thus a higher expected goodput. In addition, as shown by the experiments where the goodput is maintained at a pre-defined value, the AGT algorithm can be used to build an autonomous fault tolerant solution that accomplishes system-level goals.

CHAPTER 7: CONCLUSION

This dissertation demonstrated the feasibility of an integrated approach to fault handling in FPGAs. A population of alternatives, when combined with a competitive evolutionary strategy, provides a framework that refurbishes fault-affected configurations. Group testing-based fault isolation methods are presented. Based on a straightforward FPGA model, an autonomous group-testing algorithm for runtime fault isolation that removes the need for exhaustive test inputs and the need for the system to be taken offline is developed. To this end, a discrepancy detector is designed for fault detection. In order to demonstrate the flexibility of group testing techniques, a group testing-based technique for identifying faulty FPGA embedded cores is also presented that highlights the utility of group testing for exhaustive functional testing. FIAT, a fault analysis toolkit, is developed to enable fault isolation experiments on FPGAs. Finally, an autonomous group testing technique is demonstrated that maintains the system goodput at pre-defined levels throughout the fault isolation process.

7.1. Graceful Degradation of Performance

In applications where the FPGA on which the application is deployed cannot be retrieved for repair or replacement, graceful degradation of service is a highly desirable quality. Deep-space deployment of FPGAs provides an example of such a scenario. In deep-space, the probability of failures also increases due to the absence of a protective atmosphere. In this high-ionizing radiation environment, multiple hardware faults

induced by high-energy particles demand a fault tolerance implementation that can ensure that the system remains available even in the presence of faults. While fast recovery from faults is essential, certain applications might demand that the FPGA continue to provide service, at reduced availability, as opposed to not providing any service at all during the recovery process. A system that degrades gracefully as faults appear should be able to handle faults while continuing to provide acceptable levels of service. Through the elimination of additional test vectors and by using a temporal assessment process based on aging and outlier identification, CRR provides a self-regulating repair mechanism with reduced downtime which is also capable of such graceful degradation.

With a limited pool of resources on an FPGA, sustainable fault handling is achieved only when the available resources are recycled. Such resource recycling needs to leverage residual functionality provided by defective resources. A LUT which has a stuck-at fault at one of the input pins might still provide residual functionality. Section 3.8 shows that such functionality can be leveraged by a system that measures performance by evaluating the outputs to actual runtime inputs – as opposed to a system where the resources are exhaustively tested using additional test vectors. As an example, an evolutionary algorithm that relies on a fitness function-based evaluation of a configuration's performance might tolerate a LUT with a stuck-at fault at one input pin, if the faulty input pin is not used by the configuration. A design in which the faulty four-input LUT is only required to receive three-bit inputs will be identified as being fully-fit by an FPGA, but will be precluded from use if the resources were to be tested exhaustively.

7.2. Improving Evolutionary Repair using a Population of Alternatives

Two major improvements over a more conventional GA-based repair scheme are observed. First, this dissertation provides evidence for a significant improvement in fault handling capability by exploiting population diversity during all phases of the fault handling process. By relying on the inherent information contained in a population of alternatives, the approach improves on previous techniques for evolutionary fault handling that have the objective of creating a single best-fit individual. In CRR, the population of alternatives is classified into separate pools of relative operability, and all individuals are refurbished over time with no one individual being preferred over others. As opposed to previous approaches, the goal of CRR is to maintain a healthy population, as opposed to creating one single individual that acts as the responder in the case of faults. Secondly, GAs asymptotically approach the perfect configuration. With CRR, these partially fit configurations provide an increased benefit. CRR's competitive focus automatically chooses the best performing configurations for a given input space.

A significant observation made during fault refurbishment experiments is that a system that functions in a fully-fit manner can be realized using configurations that are not themselves fully-fit. Individuals that perform best for the subset of inputs that are observed provide high goodput even when they may not demonstrate ideal behavior for the entire input space. As the subset of observed inputs change over time, alternate partially-fit configurations may be identified that provide high quality service for the new inputs. Such redundancy can occur at minimal physical resource overhead and is limited by the storage space requirement and reconfiguration time. Interestingly, the dual-

competition system presented in CRR can easily be extended to three competing modules to provide a more traditional TMR system that can provide even higher quality of service at the cost of the physical resources needed to implement an extra module.

7.3. Fast Fault Response using Group Testing

While the evolutionary algorithm excels at recycling resources and finding solutions that may seem counter-intuitive, this comes at the cost of the time required to identify the solution. This is where group testing-based isolation provides a direct benefit by fast identification of the fault-affected resource. More importantly, by tracking the resource allocation across configurations, this also provides alternative configurations to respond to faults with minimal latency. The group testing-based fault isolation method presented in this work demonstrates the capability for the fast isolation of logic faults, and, more importantly, the ability to maintain the system's availability and goodput throughout the fault isolation process. For example, Section 6.6 shows how the AGT system maintains the system goodput at 90%. This does not have to delay the speed with which a functioning configuration is identified to respond to the fault. The experiments in Section 6.3 show that with as few as three discrepant outputs, the system identifies the faulty resources for a DES implementation. Due to the use of multiple alternative configurations that are designed in a way that minimizes the probability for all configurations to be affected by the same hardware resource fault, a handy replacement for guaranteed service is immediately available in case of a single fault.

The versatility of group testing-based isolation is clearly demonstrated by the case study where fault embedded cores in FPGAs were identified using BIST techniques. Group testing techniques are also shown to suitable for exhaustive offline testing, and can provide a significant improvement in fault isolation time over a more conventional BIST approach as demonstrated. A 640 DSP core FPGA device is tested exhaustively with a 30% testing resource overhead in a single stage of tests that are designed using group testing principles. An adaptive multi-stage group testing algorithm can provide fault isolation for online FPGAs. This dissertation demonstrates viability, and the methods presented here can be further enhanced and improved based on the specific system in which they are implemented. For example, a group testing regimen can be developed for TMR systems, and improvements to many other exhaustive testing are possible using various group testing techniques that have already been analyzed and researched.

7.4. Future Work

While CRR is show to be capable of achieving refurbishment in combinational logic circuits in Section 3.8, it remains to be seen if it can be extended to sequential logic circuits. 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. For any sequential circuit of substantial size, the number of states of the circuit, and transitions between the states make fitness evaluation challenging. A general strategy to enable evolutionary repair of sequential circuits remains to be addressed. Also, CRR provides coverage for only the logic resources. Though there are several approaches for

tolerating faults in the interconnect resources, the choices are severely limited when it comes to online isolation of such faults. Thus, the integration of interconnect-fault and logic-fault handling strategies for online fault-handling remains a major challenge.

Partial reconfiguration in COTS FPGAs is currently hindered by severe limitations, and support for partial reconfiguration is subjective at best. Currently, the time taken for partial reconfiguration is a significant bottleneck in effecting repairs. The lack of well-tested and supported APIs to reconfigure only a portion of the FPGA while keeping the rest of the FPGA operational is also a major roadblock [62]. To realize fast online fault handling, there is a need for more open standards and improved support for partial reconfiguration. In commercial SRAM FPGAs there is a very high level of dependency on the design tools provided by the manufacturer. With an open bitstream structure, and more portable design tools, it may be possible in the future to instantiate evolutionary algorithms within the design loop. Currently, due to the closed nature of the configuration bitstream's structure, one has to rely on the Xilinx tools to produce the configuration bitstream, and it is almost impossible to produce and modify the bitstream in a guaranteed fashion to achieve desired functional changes.

Finally, further enhancements can be made to FIAT, and FIAT can be used to analyze the performance of alternative group testing strategies. Since it provides a set of tools for the injection of faults, and to manage and track resource allocation across configurations, it should serve as a useful tool for further experiments in FPGA fault tolerance. While this dissertation provides a new paradigm for a hardware-in-the-loop online fault tolerance

strategy, several alternative target technologies, such as software reliability tools, or future nano-scale mechanisms can benefit from the same principles.

REFERENCES

[1] J. Lohn, G. Larchev, and R. DeMara, "Evolutionary fault recovery in a Virtex FPGA using a representation that incorporates routing," in *Parallel and Distributed Processing Symposium*, 22-26 April 2003.

[2] J. Lohn, G. Larchev, and R. DeMara, "A Genetic Representation for Evolutionary Fault Recovery in Virtex FPGAs," *Evolvable Systems: From Biology to Hardware, 5th Intl. Conf.(ICES 2003),* pp. 47–56, 2003.

[3] D. Keymeulen, R. S. Zebulum, Y. Jin, and A. Stoica, "Fault-tolerant evolvable hardware using field-programmable transistor arrays," *Reliability, IEEE Transactions on,* vol. 49, pp. 305-316, 2000.

[4] J. M. Perotti, A. R. Lucena, P. J. Medelius, C. T. Mata, B. M. Burns, and A. J. Eckhoff, "Advanced Data Acquisition Systems " *KSC Research & Technology 2002 Annual Report,* 2002.

[5] D. P. Siewiorek and R. S. Swarz, *The theory and practice of reliable system design*: Digital Press, 1982.

[6] A. Matrosova, V. Ostrovsky, I. Levin, and K. Nikitin, "Designing FPGA based self-testing checkers for m-out-of-n codes," *On-Line Testing Symposium, 2003. IOLTS 2003. 9th IEEE,* pp. 49-53, 2003.

[7] E. J. McCluskey, "Design Techniques for Testable Embedded Error Checkers," *Computer,* vol. 23, pp. 84-88, 1990.

[8] J. C. Laprie, "Dependable Computing: Concepts, Limits, Challenges," *Special Issue of the 25th International Symposium On Fault-Tolerant Computing,* pp. 42-54.

[9] F. MacWilliams and N. Sloan, "The Theory of Error Correcting Codes." vol. 16: North-Holland, New York, 1977.

[10] M. Garvie and A. Thompson, "Scrubbing away transients and jiggling around the permanent: long survival of FPGA systems through evolutionary self-repair," *On-Line Testing Symposium, 2004. IOLTS 2004. Proceedings. 10th IEEE International,* pp. 155-160, 2004.

[11] R. Dorfman, "The Detection of Defective Members of Large Populations," *The Annals of Mathematical Statistics,* vol. 14, pp. 436-440, 1943.

[12] D. Du and F. Hwang, *Combinatorial group testing and its applications*: World Scientific River Edge, NJ, 1993.

[13] H. Q. Ngo and D. Z. Du, "A survey on combinatorial group testing algorithms with applications to DNA library screening," *Discrete Mathematical Problems with Medical Applications,* pp. 171–182, 2000.

[14] A. B. Kahng and S. Reda, "Combinatorial Group Testing Methods for the BIST Diagnosis Problem," *Proceedings of Asia and South Pacific Design Automation Conference,* January 2004.

[15] J. A. Cheatham, J. M. Emmert, and S. Baumgart, "A survey of fault tolerant methodologies for FPGAs," *ACM Transactions on Design Automation of Electronic Systems (TODAES),* vol. 11, pp. 501-533, 2006.

[16] A. Doumar and H. Ito, "Detecting, diagnosing, and tolerating faults in SRAM-based field programmable gate arrays: a survey," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* vol. 11, pp. 386 - 405, June 2003.

[17] W.-J. Huang and E. J. McCluskey, "Column-Based Precompiled Configuration Techniques for FPGA," in *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*, 2001, pp. 137-146.

[18] D. Keymeulen, R. S. Zebulum, Y. Jin, and A. Stoica, "Fault-Tolerant Evolvable Hardware Using Field-Programmable Transistor Arrays," *IEEE Transactions On Reliability,* vol. 49, September 2000.

[19] J. Lach, W. H. Mangione-Smith, and M. Potkonjak, "Low overhead fault-tolerant FPGA systems," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on,* vol. 6, pp. 212-221, 1998.

[20] S. Dutt, V. Shanmugavel, and S. Trimberger, "Efficient incremental rerouting for fault reconfiguration in field programmable gate arrays," *International Conference on Computer Aided Design: Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design,* vol. 7, pp. 173-177, 1999.

[21] V. Lakamraju and R. Tessier, "Tolerating operational faults in cluster-based FPGAs," *Proceedings of the 2000 ACM/SIGDA eighth international symposium on Field programmable gate arrays,* pp. 187-194, 2000.

[22] M. Garvie and A. Thompson, "Scrubbing away transients and Jiggling around the permanent: Long survival of FPGA Systems through evolutionary self-repair," in *10th IEEE International On-Line Testing Symposium*, Funchal, Madeira Island, Portugal, July 12-14, 2004.

[23] C. J. Milliord, C. A. Sharma, and R. F. DeMara, "Dynamic Voting Schemes to Enhance Evolutionary Repair in Reconfigurable Logic Devices," *Proceedings of the 2005 International Conference on Reconfigurable Computing and FPGAs (ReConFig'05) on Reconfigurable Computing and FPGAs,* 2005.

[24] R. S. Oreifej, C. A. Sharma, and R. F. DeMara, "Expediting GA-Based Evolution Using Group Testing Techniques for Reconfigurable Hardware," *proc. International Conference on Reconfigurable Computing and FPGAs (Reconfig'06), San Luis Potosi, Mexico,* pp. 106-113, 2006.

[25] J. M. Emmert and D. K. Bhatia, "A Fault Tolerant Technique for FPGAs," *Journal of Electronic Testing,* vol. 16, pp. 591-606, 2000.

[26] R. Ross and R. Hall, "A FPGA Simulation Using Asexual Genetic Algorithms for Integrated Self-Repair," *Proceedings of the first NASA/ESA conference on Adaptive Hardware and Systems,* pp. 301-304, 2006.

[27] A. P. Shanthi and R. Parthasarathi, "Exploring FPGA structures for evolving fault tolerant hardware," in *2003 NASA/DoD Conference on Evolvable Hardware*, Chicago, Illinois, 9-11 July 2003, pp. 174 - 181.

[28] J. M. Emmert, C. E. Stroud, and M. Abramovici, "Online Fault Tolerance for FPGA Logic Blocks," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on,* vol. 15, pp. 216-226, 2007.

[29] M. Abramovici, J. M. Emmert, and C. E. Stroud, "Roving STARs: An Integrated Approach to On-Line Testing, Diagnosis, and Fault Tolerance for FPGAs in Adaptive Computing Systems," *Proc. Third NASA/DoD Workshop on Evolvable Hardware,* pp. 73-92, 2001.

[30] S. Vigander, "Evolutionary Fault Repair in Space Applications," in *Dep. of Computer & Information Science*. vol. Masters Thesis Trondheim: Norwegian University of Science and Technology (NTNU), 2001.

[31] P. Layzell and A. Thompson, "Understanding Inherent Qualities of Evolved Circuits: Evolutionary History as a Predictor of Fault Tolerance," *Proceedings of Third Int. Conf. on Evolvable System (ICES2000,* vol. 1801, pp. 133-142.

[32] X. Yao, Y. Liu, and P. Darwen, "How to make best use of evolutionary learning," *Complex Systems: From Local Interactions to Global Phenomena,* pp. 229–242, 1996.

[33] X. Yao and Y. Liu, "Making use of population information in evolutionary artificialneural networks," *Systems, Man and Cybernetics, Part B, IEEE Transactions on,* vol. 28, pp. 417-425, 1998.

[34] X. Yao and Y. Liu, "Getting most out of evolutionary approaches," *Evolvable Hardware, 2002. Proceedings. NASA/DoD Conference on,* pp. 8-14, 2002.

[35] B. Bridgford, C. Carmichael, and C. W. Tseng, "Correcting Single-Event Upsets in Virtex-II Platform FPGA Configuration Memory," *Xilinx Application Note XAPP197,* 2007.

[36] C. H. Carmichael and P. E. Brinkley Jr, "Techniques for mitigating, detecting, and correcting single event upset effects in systems using SRAM-based field programmable gate arrays," Google Patents, 2007.

[37] C. Carmichael, M. Caffrey, and A. Salazar, "Correcting Single-Event Upsets Through Virtex Partial Configuration," *Xilinx Application Notes,* vol. 216, 2000.

[38] P. J. Rousseuw and A. M. Leroy, "Robust Regression and Outlier Detection," *New York: Jon Wiley & Sons Inc,* 1987.

[39] P. Flajolet, D. Gardy, and L. Thimonier, "Birthday paradox, coupon collectors, caching algorithms and self-organizing search," *Discrete Applied Mathematics,* vol. 39, pp. 207-229, 1992.

[40] D. C. Hoaglin and R. E. Welsch, "The Hat Matrix in Regression and ANOVA," *The American Statistician,* vol. 32, pp. 17-22, 1978.

[41] H. V. Henderson and P. F. Velleman, "Building Multiple Regression Models Interactively," *Biometrics,* vol. 37, pp. 391-411, 1981.

[42] J. F. Miller, P. Thomson, and T. Fogarty., "Designing Electronic Circuits Using Evolutionary Algorithms. Arithmetic Circuits: A Case Study," in *Algorithms and Evolution Strategy in Engineering and Computer Science*, D. Quagliarella, J. Periaux, C. Poloni, and G. Winter, Eds. Chichester, England, 1998, pp. 105-131.

[43] D. Du and F. Hwang, *Combinatorial Group Testing and Its Applications*: World Scientific, 2000.

[44] Xilinx, "Virtex-II Platform FPGAs: Complete Data Sheet," *DS031, v3,* vol. 4.

[45] S. Douglass, "Introducing the Virtex-5 FPGA Family," *Xcell Journal, Xilinx,* pp. 8-11, 2006.

[46] S. K. Jain and C. E. Stroud, "Built-in Self Testing of Embedded Memories," *IEEE Design & Test of Computers,* vol. 3, pp. 27-37, 1986.

[47] P. Camurati, P. Prinetto, M. S. Reorda, S. Barbagallo, A. Burri, and D. Medina, "Industrial BIST of embedded RAMs," *Design & Test of Computers, IEEE,* vol. 12, 1995.

[48] C. Stroud, S. Konala, P. Chen, and M. Abramovici, "Built-In Self-Test of Logic Blocks in FPGAs," in *VLSI Test Symposium* Princeton, NJ, pp. 387-392.

[49] M. Abramovici and C. E. Stroud, "BIST-based test and diagnosis of FPGA logic blocks," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on,* vol. 9, pp. 159-172, 2001.

[50] E. Atoofian and Z. Navabi, "A BIST Architecture for FPGA Look-Up Table Testing Reduces Reconfigurations," *Proceedings of the 12th Asian Test Symposium,* pp. 84-89, 2003.

[51] J. Liu and S. Simmons, "BIST-diagnosis of interconnect fault locations in FPGA's," *Electrical and Computer Engineering, 2003. IEEE CCECE 2003. Canadian Conference on,* vol. 1, 2003.

[52] C. Stroud and S. Garimella, "Built-In Self-Test AND Diagnosis OF Multiple Embedded Cores IN So Cs," *Proceedings of The 2005 International Conference on Embedded Systems and Applications,* pp. 130-136.

[53] S. Garimella and C. Stroud, "A system for automated built-in self-test of embedded memory cores in system-on-chip," *System Theory, 2005. SSST'05. Proceedings of the Thirty-Seventh Southeastern Symposium on,* pp. 50-54, 2005.

[54] M. Renovell, J. M. Portal, J. Figueras, and Y. Zorian, "SRAM-Based FPGAs: Testing the Embedded RAM Modules," *Journal of Electronic Testing,* vol. 14, pp. 159-167, 1999.

[55] A. J. Van De Goor, "Using march tests to test SRAMs," *IEEE Design & Test of Computers,* vol. 10, pp. 8-14, 1993.

[56] Xilinx, "Spartan-3A DSP FPGA Family: Complete Data Sheet," DS610, http://www. datasheetcatalog. com, 2007.

[57] A. Sarvi and J. Fan, "Automated BIST-based diagnostic solution for SOPC," *Design and Test of Integrated Systems in Nanoscale Technology, 2006. DTIS 2006. International Conference on,* pp. 263-267, 2006.

[58] R. S. Oreifej, C. A. Sharma, and R. F. DeMara, "Expediting GA-Based Evolution Using Group Testing Techniques for Reconfigurable Hardware," *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (Reconfig'06), San Luis Potosi, Mexico,* pp. 106-113, 2006.

[59] Xilinx, "Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet, Version 4.6," *Publication Number DS083,* 2006.

[60] R. N. Al-Haddad, C. A. Sharma, and R. F. DeMara, "Performance Evaluation of Two Allocation Schemes for Combinatorial Group Testing Fault Isolation Method," in *IEEE International Conference on Reconfigurable Computing and FPGAs (ReConfig '06)* San Luis Potosi, Mexico, pp. 106-113.

[61] J. Gait, "Validating the Correctness of Hardware Implementations of the NBS Data Encryption Standard," *NBS Special Publication 500-20,* November 1977.

[62] H. Tan and R. F. DeMara, "A Multi-layer Framework Supporting Autonomous Runtime Partial Reconfiguration."