OPTIMIZING DYNAMIC LOGIC REALIZATIONS FOR PARTIAL RECONFIGURATION OF FIELD PROGRAMMABLE GATE ARRAYS

by

MATTHEW G. PARRIS
B.S. University of Louisville, 2005

A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science
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

Spring Term
2009

©2008 Matthew G. Parris

ABSTRACT

Many digital logic applications can take advantage of the reconfiguration capability of Field Programmable Gate Arrays (FPGAs) to dynamically patch design flaws, recover from faults, or time-multiplex between functions. Partial reconfiguration is the process by which a user modifies one or more modules residing on the FPGA device independently of the others. Partial Reconfiguration reduces the granularity of reconfiguration to be a set of columns or rectangular region of the device. Decreasing the granularity of reconfiguration results in reduced configuration filesizes and, thus, reduced configuration times. When compared to one bitstream of a non-partial reconfiguration implementation, smaller modules resulting in smaller bitstream filesizes allow an FPGA to implement many more hardware configurations with greater speed under similar storage requirements.

To realize the benefits of partial reconfiguration in a wider range of applications, this thesis begins with a survey of FPGA fault-handling methods, which are compared using performance-based metrics. Performance analysis of the Genetic Algorithm (GA) Offline Recovery method is investigated and candidate solutions provided by the GA are partitioned by age to improve its efficiency. Parameters of this aging technique are optimized to increase the occurrence rate of complete repairs. Continuing the discussion of partial reconfiguration, the thesis develops a case-study application that implements one partial reconfiguration module to demonstrate the functionality and benefits of time multiplexing and reveal the improved efficiencies of the latest large-capacity FPGA architectures. The number of active partial reconfiguration modules implemented on a single FPGA device is increased from one to eight to implement a dynamic video-processing architecture for Discrete Cosine Transform and Motion Estimation functions to demonstrate a 55-fold reduction in bitstream storage requirements thus improving partial reconfiguration capability.

To my wife, Kathryn

ACKNOWLEDGMENTS

The author thanks the following for their significant contribution to this work:

Kathryn Parris for her enduring love and patience.

Beckie Lewis, Patrick Breen, Arnold Postell, Pat Simpkins, and Bill Parsons of NASA Kennedy Space Center for supporting this endeavor through the Kennedy Graduate Fellowship Program.

Carthik Sharma, Kening Zhang, Rashad Oreifej, Rawad Al-Haddad, Raul Dookhoo, and Heng Tan for their camaraderie and expertise.

Dr. Lee and Dr. Wu for their encouragement as professors and dedication as Committee Members.

Dr. DeMara for his impressive ability as a mentor in truly caring for his students and inspiring them to greatness.

God for bestowing the gift of intelligence to know Him and benefit humanity.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS/ABBREVIATIONS

AFTB	Atomic Fault Tolerant Block
AG	Area Group
ALPS	Age-Layered Population Structure
ASIC	Application Specific Integrated Circuit
BIST	Built-in Self Test
BLE	Basic Logic Element
BUT	Block Under Test
CED	Concurrent Error Detection
CGT	Combinatorial Group Testing
CLB	Configurable Logic Block
CPLD	Complex Programmable Logic Device
CRR	Competitive Runtime Reconfiguration
DCT	Discrete Cosine Transform
DSP	Digital Signal Processor
EAPR	Early Access Partial Reconfiguration
EPROM	Erasable Programmable Read-Only Memory
FABRIC	Fault-Bypassing Roving Configuration
FIFO	First In, First Out
FPGA	Field Programmable Gate Array
GA	Genetic Algorithm
GND	Electrical Ground

HDL	Hardware Description Language
IOB	Input Output Buffer
JPEG	Joint Photographic Experts Group
I/O	Input/Output
LOC	Location
LPSIF	Lander Pyro Switching Interface
LUT	Look-up Table
MD5	Message-Digest Algorithm 5
ME	Motion Estimation
MER	Mars Exploration Rover
MGM	Minimax Grid Matching
MUX	Multiplexer
NASA	National Aeronautics and Space Administration
NCD	Native Circuit Description
ORA	Output Response Analyzer
OTP	One-time Programmable
PAR	Place and Route
PE	Processing Element
PLB	Programmable Logic Block
PS/2	Personal System/2
RAM	Random Access Memory
TBUF	Tri-state Buffer
SEL	Single Event Latchup

SEU	Single Event Upset
SHA-1	Secure Hash Algorithm 1
SRAM	Synchronous Random Access Memory
STAR	Self-Testing Area
TMR	Triple Modular Redundancy
TMRSB	TMR with Standby
TPG	Test-Pattern Generator
TREC	Test and Reconfiguration Controller
VCC	Electrical Positive Supply Voltage
VGA	Video Graphics Array
WUT	Wires Under Test

CHAPTER 1
INTRODUCTION

1.1 Field Programmable Gate Arrays

After the advent of the transistor, Application Specific Integrated Circuits (ASIC) emerged to combine different types and numbers of transistors, allowing engineers to create complex digital designs on a single silicon substrate. To avoid expensive non-recurring engineering costs associated with custom ASIC designs, engineers could forgo the unrestrained flexibility of ASICs for quick programmability of simple, fabricated devices called Complex Programmable Logic Devices (CPLD). As a solution between these two extremes, Field Programmable Gate Arrays (FPGA) appeared in the 1980's to combine the complexity of ASICs with the programmability of CPLDs. Since their inception, FPGAs have enabled designers to develop complex systems quickly, decreasing the time-to-market and providing the public with the latest technologies more rapidly.

The hardware architecture of FPGAs originates from concepts found within CPLDs. As seen in Figure 1-1, an array of Programmable Logic Blocks (PLB) and programmable interconnect composes the architecture. PLBs contain logic and register resources to implement both sequential and combinatorial circuits whereas programmable interconnect combines PLBs across the FPGA to realize complex circuit designs. PLBs may implement logic functions with multiplexers (MUX) or Look-Up Tables (LUT). LUTs may have three or four inputs connected to an 8x1 or 16x1 table that utilizes the inputs as an address to select the logic stored at the appropriate location.

In addition to being LUT-based and MUX-based, FPGAs may be configured through anti-fuses [Actel 2005] or Static Random Access Memory (SRAM) cells [Altera 2008]. Anti-fuse FPGAs are One-Time Programmable (OTP) and, as such, configure logic and interconnect by sending a high programming current to form links between routing and voltage levels. The logic and inter-

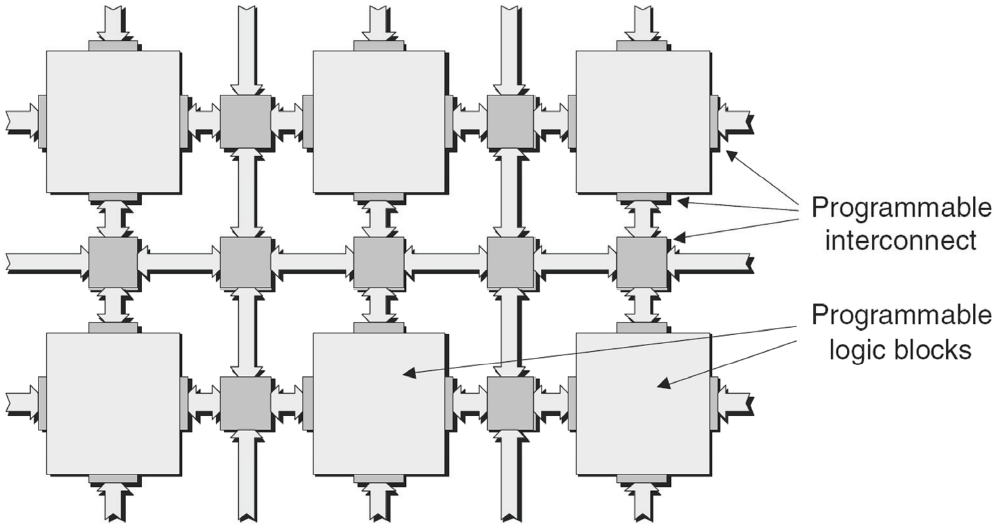


Figure 1-1: Top-down View of a Simple, Generic FPGA Architecture [Maxfield 2004]

connect configuration of an SRAM FPGA is stored by SRAM cells, allowing unlimited configurability. Whereas anti-fuse FPGAs provide more protection against design theft and some inherent protection from space-borne radiation, SRAM FPGAs allow end-users to change the configuration of a design multiple times. This flexibility allows designers to test prototype designs on the FPGA prior to production. In addition, users are able to update the configuration of a design during the operation life of the FPGA and even designate certain areas of the FPGA to house various modules depending on the needs of the system.

FPGAs have found use among various applications including data processing, networks, automotive, and industrial. The reconfigurability of FPGAs decreases the time-to-market of these hardware applications that would otherwise require its functionality to be hard-wired by a manufacturer. Additionally, the ability to reconfigure its functionality in the field mitigates unforeseen design errors. Both of these characteristics make FPGAs an ideal target for spacecraft applications such as ground support equipment, Reusable Launch Vehicles, sensor networks, planetary rovers, and deep space probes [Katz and Some 2003; Kizhner et al. 2007; Ratter 2004; Wells and Loo 2001].

1.2 Space Applications: Radiation-Induced Faults and Handling Techniques

In-flight devices encounter harsh environments of mechanical/acoustical stress during launch and high ionizing radiation and thermal stress while outside Earth's atmosphere. FPGAs must operate reliably for long mission durations with limited or no capabilities for diagnosis/replacement and little onboard capacity for spares. Mission sustainability realized by autonomous repair of these reconfigurable devices is of particular interest to both in-flight applications and ground support equipment for National Aeronautics and Space Administration (NASA) space missions [Yui et al. 2003].

When in the space environment, FPGAs are subject to cosmic rays and high-energy protons, which can cause malfunctions to occur in systems located on FPGAs. These malfunctions may be a result of *Single-Event Upsets* (SEU) or *Single-Event Latch-ups* (SEL) [Wirthlin et al. 2003]. SEUs are transient in nature, inverting bits stored in memory cells or registers, whereas SELs are permanent by inducing high operating current into logic or routing resources. Whereas 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 SRAM FPGAs, on the other hand, store its configuration in SRAM cells, which increases the risk to SEUs. Additionally, decreasing operating voltages and transistor gate widths further increases the risk to SEUs. Before the availability of radiation-hardened SRAM FPGAs, designers of satellites and rovers had no serious alternative to the OTP anti-fuse FPGA. If the inherent fault tolerant capability of anti-fuse FPGAs was not sufficient, designers were restricted to employing *Passive Fault-Handling Methods* such as Triple Modular Redundancy (TMR). Due to the reconfigurable nature of SRAM FPGAs, radiation-hardened SRAM FPGAs have enabled designers to consider other fault-handling methods such as the *Active Fault-Handling Methods* described in Sections 2.2 and 2.3.

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, other 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 for extended missions.

Scrubbing is a fault-handling technique commonly used to reprogram affected FPGA configuration memory cells with viable configuration data. Scrubbing depends upon reading back the configuration memory cells and *detecting* faults by comparing them to the original configuration. Upon *isolating* a fault, the FPGA can *recover* the correct bitstream through reconfiguration. The Mars Exploration Rovers (MER) landing system successfully implemented this method to mitigate SEUs occurring within the Lander Pyro Switching Interface (LPSIF) during the 200-day transit to Mars [Ratter 2004]. As seen in Figure 1-2, about 10 errors had occurred halfway through the transit and approximately 25 errors can be predicted for the entire transit. For a critical system such as the landing pyrotechnics, scrubbing ensured mission success.

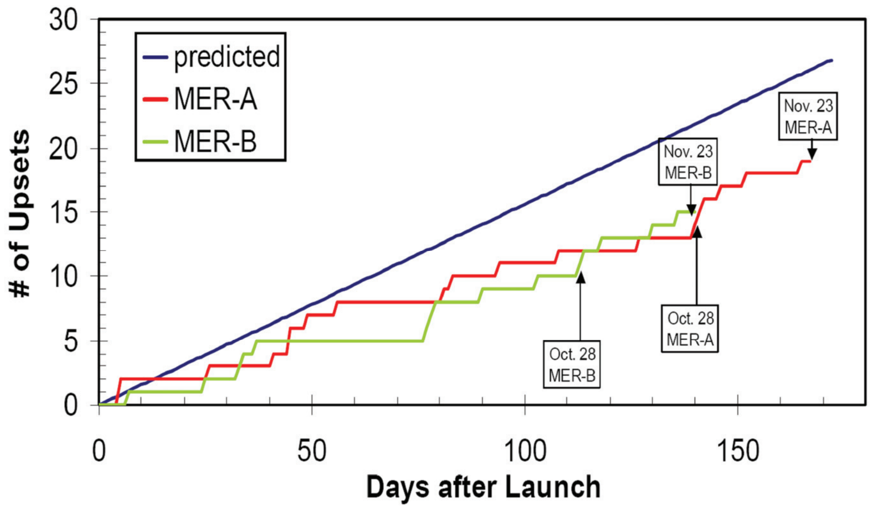


Figure 1-2: SEU Occurrences in Xilinx XQR4062XL FPGAs in the MER LPSIF [Swift 2006]

Whereas Scrubbing handles SEUs in the configuration memory, additional fault-handling methods are necessary to address both transient faults in non-configuration memory elements, such as flip-flops and the registers they compose, and other permanent faults in the remaining components of the FPGA. Sections 2.2, 2.3, and 2.4 classify such fault-handling methods and describe them in detail.

1.3 Partial Reconfiguration Overview

All SRAM FPGAs require a full-device reconfiguration upon power-up. Initialization involves programming the FPGA with a configuration bitstream file, which resets and configures all logic, interconnect, and Input/Output (I/O) resources. After initialization, *partial reconfiguration* is the capability to modify a fraction of the resources by programming the FPGA with a partial bitstream file. As discussed in detail within CHAPTER 4 and CHAPTER 5, a full bitstream may be as large as 1,712,614 bytes whereas a partial bitstream may be 2% of this size at 28,306 bytes. This multiple order-of-magnitude reduction in configuration file size can realize several benefits such as reduced reconfiguration time, reduced storage requirements, and dynamic allocation of functionality as described in detail below in Section 1.4. For use with a fault-handling method such as scrubbing, an additional advantage of partial reconfiguration includes allowing normal operation of resources not affected by the partial reconfiguration [Carmichael et al. 2000; Yui et al. 2003].

Xilinx provides many FPGA devices that support partial reconfiguration, ranging from the simplest Spartan-3 device to the most complex Virtex-5 device. Due to their popularity and wide range of devices, partial reconfiguration is discussed in the context of the capability provided by Xilinx FPGAs. Xilinx provides two forms of partial reconfiguration: static and active [Kao 2005]. *Static* partial reconfiguration modifies a portion of the FPGA configuration while the entire device remains inactive and non-operational. *Active* partial reconfiguration, on the other hand, occurs while the device is active and operational. In the active case, portions of the FPGA not affected by reconfiguration continue nominal operations during the reconfiguration process. Further discussions of partial reconfiguration deal solely with active partial reconfiguration.

Two methods of generating an FPGA partial bitstream file exist: difference-based and module-based. *Difference-based* partial reconfiguration enables designers to make small modifications to the configuration of logic resources [Xilinx 2007a]. After synthesis, translation, as well as Place and Route (PAR) are complete for a design and a Native Circuit Description (NCD) file is generated, these small logic modifications are made. The Xilinx FPGA_Editor design utility accesses this NCD file and allows users to modify LUT contents, I/O standards, and block RAM (Random Access Memory) contents. The Xilinx bitstream generation utility, BitGen, 1) generates a new full-device bitstream reflecting the modifications, 2) compares the new bitstream to the original bitstream, and 3) generates a partial bitstream only containing the differences between the two. When the FPGA is reconfigured using the partial bitstream, only those logic resources modified using the FPGA_Editor tool are modified. Depending on the type and number of modifications, the filesize of this partial bitstream is typically orders of magnitude less than the initial full-device bitstream and, consequently, requires a reconfiguration time orders of magnitude less. Difference-based partial reconfiguration is only applicable if the original and new FPGA configurations are available, which may not apply to Evolvable Hardware techniques.

The module-based design flow is a coarse-grained method where specific areas of the FPGA are designated as reconfigurable and can contain one or more modules within an application [Xilinx 2006]. Figure 1-3 shows a top-level view where the reconfigurable areas maintain a constant size and location throughout the life of the application. For each reconfigurable area, the design process forms boundaries into which all logic and interconnect resources of its module must reside. Additionally, *bus macros* define the static input/output ports through which all communication between its module and other modules must take place. The static nature of the reconfigurable area with respect to size and I/Os allows multiple versions of one module to be interchanged dynamically without affecting other portions of the FPGA. More detail on this topic is provided in CHAPTER 4, which

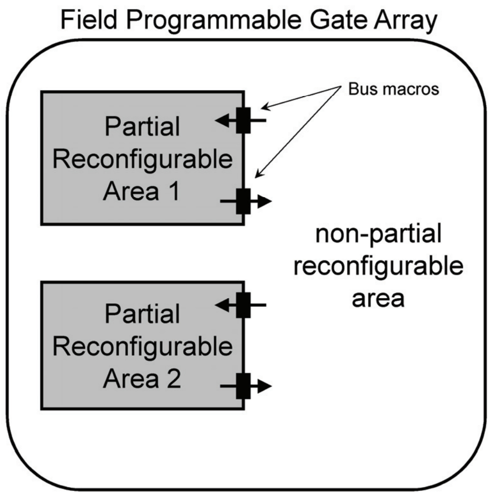

Field Programmable Gate Array
Bus macros
Partial Reconfigurable Area 1
non-partial reconfigurable area
Partial Reconfigurable Area 2

Figure 1-3: Module-based Design Layout with Two Reconfigurable Modules

explains the module-based design flow. It accompanies CHAPTER 5, which describes applications that utilize the module-based approach.

1.4 Benefits of Partial Reconfiguration

Full-device reconfiguration is the process of changing the arrangement of all utilized resources on the FPGA. Due to the unlimited programmability of SRAM FPGAs, the configuration may be modified many times during an extended mission. One immediate benefit of full-device reconfiguration is that unforeseen design errors may be resolved by revising the bitstream to reconfigure the FPGA. Additionally, an FPGA with reconfiguration may time-multiplex between two functions that would otherwise not fit within the allocated number of resources on the FPGA; this would allow the FPGA to be configured with Function A at one time and at another time the FPGA is configured for Function B. With reconfiguration, each function may utilize the total number of resources on the FPGA by loading each function separately, whereas without reconfiguration both functions are loaded together, of which the sum of resources cannot exceed the total number of resources on the FPGA.

Partial Reconfiguration is the process where only a portion of the FPGA is reconfigured. Partial reconfiguration provides all the benefits of full-device reconfiguration with two additional advantages: 1) the unchanged portion of the FPGA is not affected and, in some cases, may continue execution, and 2) a partial bitstream is smaller in filesize than a full bitstream. Since partial reconfiguration does not affect the unchanged portion, applications that require critical components to continue operation may be implemented on the same chip as modules that undergo many modifications.

Since the size of the bitstream is directly proportional to the number of resources being configured, partial reconfiguration utilizes a smaller bitstream than a full bitstream for the FPGA. The direct benefit is less space needed for storing the necessary configurations for operation. An additional benefit derived from a smaller bitstream is that the reconfiguration time is shorter. This savings in time may be particularly useful for systems that depend upon the configuration time such as repetitive intrinsic evolution processes utilizing Genetic Algorithms (GA).

With FPGAs increasing in size and capability, partial reconfiguration enables designers to realize implementations of multiple modules residing on one FPGA device. Whereas full reconfiguration implementations treat the entire FPGA as one module, partial reconfiguration decreases the module granularity from the size of an FPGA to a size as small as 16 PLBs in height [Lysaght et al. 2006]. Thus, an FPGA containing multiple modules operating simultaneously may be reconfigured to perform Function 2 instead of Function 1 while the remaining tasks continue to operate. As previously discussed, the partial bitstream filesize for Function 2 is smaller than the entire bitstream, thus this change between two functions may occur quicker. Without this finer granularity, a designer must generate and store a full bitstream for each combination of modules within the FPGA, exponentially increasing storage requirements for additional modules.

CHAPTER 4 and CHAPTER 5 demonstrate practical applications that exploit these benefits. For example, with partial reconfiguration a designer only generates one full bitstream with multiple partial bitstreams, each representing one module. For a significantly less amount of storage than what non-partial reconfiguration implementations require, a user may implement significantly more combinations of hardware arrangements, increasing the capability of one FPGA device. As suggested by CHAPTER 5, an increase in capability of smaller FPGAs through time-multiplexed pipelining of functions may be comparable to larger FPGAs not utilizing partial reconfiguration.

1.5 Thesis Outline

Various time and space optimizations, along with architectural approaches to realize dynamic functionality, are discussed throughout the thesis. CHAPTER 2, PREVIOUS WORK, surveys the current research of fault-handling techniques for FPGAs, some of which utilize partial reconfiguration to decrease the size of alternative bitstreams used to tolerate faults. The capabilities of these fault-handling methods develop a descriptive classification ranging from simple Passive techniques to robust Dynamic methods. Fault-handling methods not requiring modification of the FPGA device architecture or user intervention to recover from faults are examined and evaluated against overhead-based and sustainability performance metrics such as additional resource requirements, operational delay, fault tolerance, and fault coverage. This classification alongside these performance metrics forms a standard for useful comparisons of fault-handling methods.

CHAPTER 3, EVOLVABLE HARDWARE OPTIMIZATION STRATEGIES, expands the discussion of Standard GAs in CHAPTER 2 to investigate techniques that improve the ability of a GA to repair FPGAs. To evolve and design higher-performing antennas, previous research partitions the population of a standard GA according to the longevity of individual designs within the population using an Age-layered Population Structure (ALPS). Whereas this application may be viewed a continuous search space, CHAPTER 3 reviews the techniques proposed and applies them to the discontinuous and multimodal search space of FPGA repair. The performance of these optimization techniques is compared to a standard GA used for FPGA repair. Parameters are then optimized to increase further the performance of the ALPS strategy.

CHAPTER 4, PARTIAL RECONFIGURATION AND FPGA ARCHITECTURE ANALYSIS, proposes a case study to refine some of the benefits of partial reconfiguration. The proposed system switches between two hash algorithms, Message Digest Algorithm-5 (MD5) and

Secure Hash Algorithm-1 (SHA-1), demonstrating the ability of partial reconfiguration to time-multiplex between two applications while only requiring the spatial resources of one. In implementing the case study on a Virtex-II Pro FPGA, each step of the module-based design process is described in detail. Then, a comparison is made between Virtex-II Pro and Virtex-4 implementations to demonstrate architectural portability and assess how specific hardware devices affect the results of the software-based partial reconfiguration design flow.

CHAPTER 5, DYNAMIC PROCESSOR ALLOCATION STRATEGIES, introduces a scalable architecture for video compression functions on FPGAs that exploits a wide range of benefits provided by partial reconfiguration. More specifically, the scalable architecture focuses on the Discrete Cosine Transform (DCT) function, which is reviewed briefly in the context of the video compression process. A DCT hardware implementation is optimized to form eight discrete Processing Elements (PE), each of which adds functionality to the DCT process. Through partial reconfiguration, these PEs may be added or removed in order to satisfy dynamic requirements of the user. The architecture is shown to be scalable in both the number of PEs allocated to the DCT function and the precision with which the DCT function is calculated. CHAPTER 6 concludes the work described herein and proposes future work from this research.

1.6 Contribution of Thesis

The contributions of this thesis include the following:

1. *Novel Taxonomy:* Many different fault tolerance methods proposed by the research community, including those that detect, isolate, and repair faults, are considered to form a descriptive classification. Additionally, performance metrics that enable

quantitative comparisons of capabilities are applied to the SRAM FPGA fault-handling methods surveyed.

2. *FPGA Repair Optimization:* The Age-layered Population Structure (ALPS) is applied to the FPGA repair domain to prevent convergence of the population of candidate solutions by partitioning the population into sub-populations and injecting random individuals at regular intervals. As a result, ALPS explores more of the repair search space, which decreases the population fitness by 30%, and produces complete repairs with 300% greater frequency than a standard GA. Furthermore, introducing a new selection strategy and optimizing the selection probability increases the complete repair frequency to 500%.

3. The technique of utilizing the age of individuals to subside population convergence for evolutionary antenna design are applied to the problem domain of repairing digital circuits located on FPGAs. In doing so, Furthermore, improvements to the aging strategy are introduced and optimized to enhance the performance of ALPS. In repairing a 3-bit adder, the results presented quantify the benefit of aging by producing complete repairs with greater frequency.

4. *Architectural Analysis:* The partial reconfiguration implementation process is completed on two FPGA architectures, Xilinx Virtex-II and Virtex-4, to reveal the benefits of the newer Virtex-4 architecture. Analysis of the partial bitstream filesizes identifies the Virtex-4 to have a smaller granularity configuration frame, which generates bitstreams that more closely represent the resources intended to be reconfigured. Applications reconfiguring small portions of the Virtex-4 FPGA generate bit-

streams smaller in filesize than the Virtex-II, which results in shorter reconfiguration times.

5. *Adaptive Architecture Implementation:* Partial reconfiguration is shown to make viable a dynamic and scalable video architecture that makes use of the benefits previously discussed in Section 1.4. Without partial reconfiguration, time multiplexing of video processing functions is not possible due to long interruptions of the application from configuration times. Not only does partial reconfiguration allow portions of the FPGA not affected by the reconfiguration to operate without interruption, configuration times are decreased, which reduces the length of interruptions to areas being reconfigured. Additionally, multiple reconfigurable areas within one FPGA are shown to significantly increase the capability of the device while maintaining storage requirements similar to an application with one reconfigurable area.

CHAPTER 2
PREVIOUS WORK

2.1 Classification of Fault-Handling Techniques

As suggested by Cheatham et al. [2006], Figure 2-1 divides fault-handling approaches into two categories based on the provider of the method. *Manufacturer-Provided* fault recovery techniques [Cheatham et al. 2006; Doumar and Ito 2003] address faults at the level of the device, allowing manufacturers to increase the production yield of their FPGAs. These techniques typically require modifications to the current FPGA architectures that end-users cannot perform. Once the manufacturer modifies the architecture for the consumer, the device can tolerate faults from the manufacturing process or faults occurring during the life of the device. 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 location and number.

User-Provided methods, however, depend upon the end-user for implementation. These high-

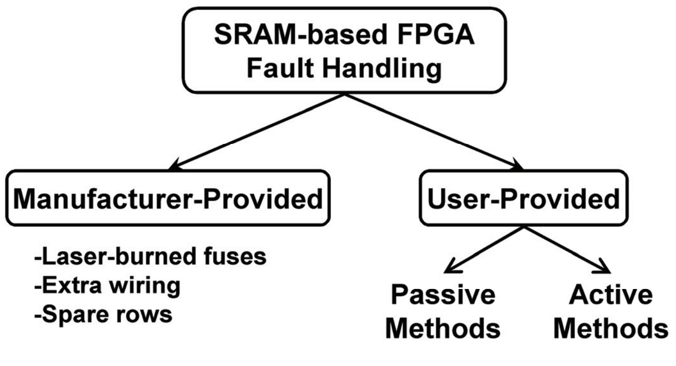


Figure 2-1: Classification of FPGA Fault-Handling Methods

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

Figure 2-1 further separates user-provided fault-handling methods into two categories based on whether an FPGA's configuration will change at run-time. *Passive Methods* embed processes into the user's application that mask faults from the system output. Techniques, such as TMR, are quick to respond and recover from faults due to the explicit redundancy inherent to the processes. Speed, however, does come at the cost of increased resource usage and power. Even when a system operates without any faults, the overhead for redundancy is continuously present. In addition to this constant overhead, these methods are not able to change the configuration of the FPGA. A fixed configuration limits the reliability of a system throughout its operational life. For example, a passive method may tolerate one fault and not return to its original redundancy level. This reduced reliability increases the chance of a second fault causing a system malfunction.

Active Methods strive to increase reliability and *Sustainability* by modifying the configuration of the FPGA to adapt to faults. This allows a system to remove accumulated SEUs and avoid permanently faulty resources to reclaim its lost functionality. In addition, active 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

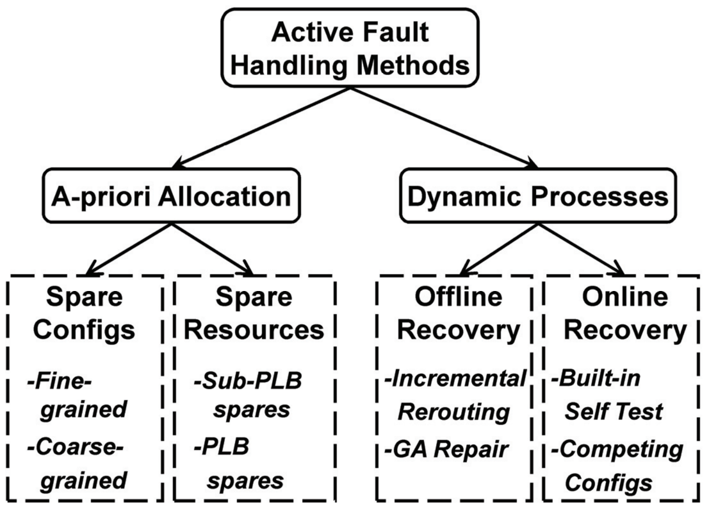


Figure 2-2: Classification of Active Fault-Handling Methods

or to generate the new configuration. Figure 2-2 illustrates two classes—*A-priori Allocation* and *Dynamic Processes*— respectively described in Sections 2.2 and 2.3.

This survey focuses on methods modifying an FPGA's configuration during run-time to address transient and permanent faults. Since SRAM FPGAs can be 1) radiation-tolerant, 2) reconfigured, and 3) partially reconfigured with the remaining portion remaining operational, research has also begun to focus on exploiting these capabilities for use in environments where human intervention is either undesirable or impossible. Table 2-I lists various considerations addressed in Section 2.4.

Table 2-I: Fault-Handling Characteristics and Considerations

	Metric	Description
Overhead	***Logic/Interconnect Resources***	**additional number of resources required due to fault-handling strategy**
	Operational Delay	**reduced rate of computations due to fault-handling strategy**
	Fault Latency	**amount of time required to begin addressing a detected and isolated fault**
	Unavailability	**amount of time system is offline to completely repair a fault**
	Recovery Goodput	**percentage of correct outputs provided during fault repair**
Sustainability	***Fault Occlusion***	**ability to bypass and/or exploit defective resources**
	Repair Granularity	**smallest arrangement of components that can be repaired**
	Fault Tolerance	**maximum number of faults handled**
	Fault Coverage	**handling of permanent, transient, logic, or interconnect faults**
	Critical Requirements	**external fault-handling components required relied upon as fault free**

2.2 A-priori Allocation

Since a typical FPGA application does not utilize 100% of the resources, the standby-spare size can be reduced from an entire FPGA to unused resources within the FPGA. A-priori Allocation takes advantage of the regularity of the FPGA architecture by assigning spare resources during design-time, independent of fault locations detected during run-time. These techniques may recover from a fault utilizing design-time compiled *spare configurations* or re-mapping and rerouting techniques utilizing *spare resources*. Spare configuration methods must provide sufficient configurations whereas spare resource methods must allocate sufficient resources to facilitate a repair without incurring too much overhead. Sections 2.2.1 and 2.2.2 respectively address these two types of A-priori Allocation.

2.2.1 Spare Configurations

2.2.1.1 Fine-grained Partitioning

Lach et al. [1998] implement a fine-grained partitioning technique where *tiles,* groups of logic and interconnect resources, are formed. The goal of the tiling technique is to partition FPGA resources in such a way that at least one spare Programmable Logic Block (PLB) is included within each tile to form *Atomic Fault-Tolerant Blocks* (AFTB). Since each AFTB contains at least one spare PLB, each tile is able to tolerate at least one PLB fault.

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

During design-time, tiling implements multiple arrangements of logic resources within an

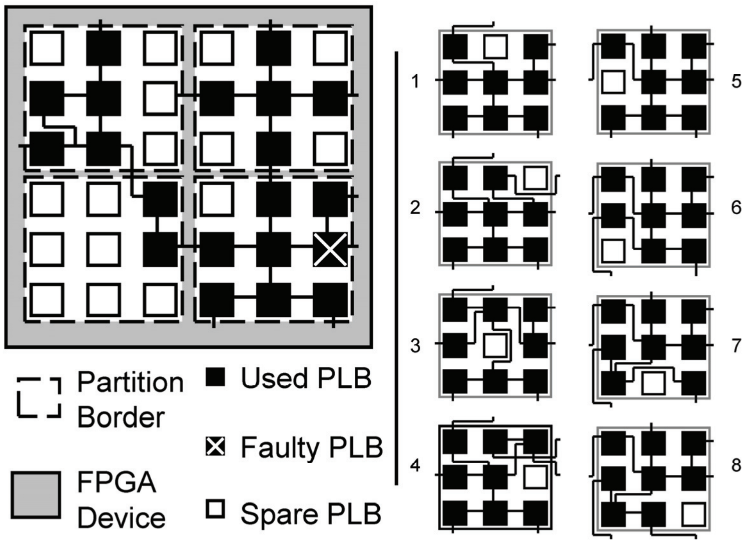


Figure 2-3: Alternate Fine-grained Configurations for a Faulty 3x3 Partition

AFTB as separate configurations such that each PLB is represented as a spare in at least one configuration. As seen in Figure 2-3, the bottom-right AFTB in the FPGA produces eight alternate configurations. To tolerate a fault during run-time, the system implements the configuration of the faulty AFTB that renders the faulty PLB as spare, effectively bypassing the fault. Figure 2-3 depicts configuration #4 as one such alternate. Fixed inter-AFTB interfaces between alternate configurations render the arrangement of each AFTB logically independent.

2.2.1.2 Medium-grained Partitioning

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

To increase system reliability, Zhang et al. [2006] combine TMR with Standby (TMRSB) to

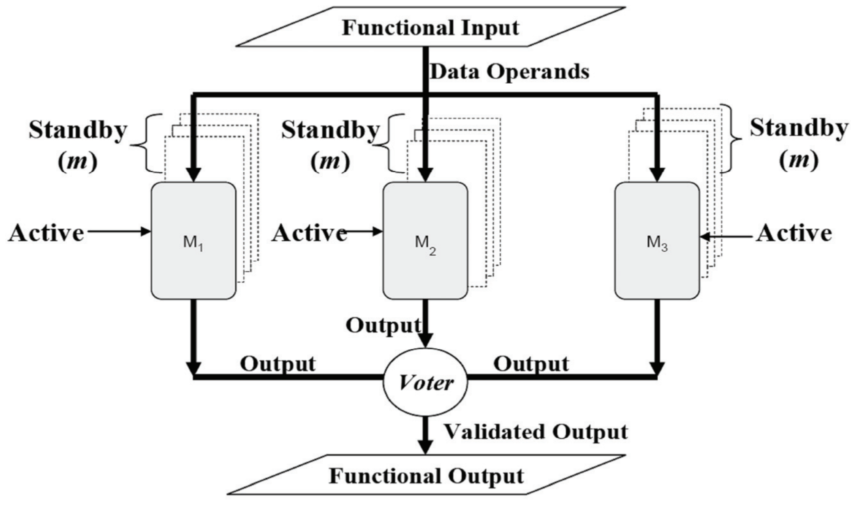


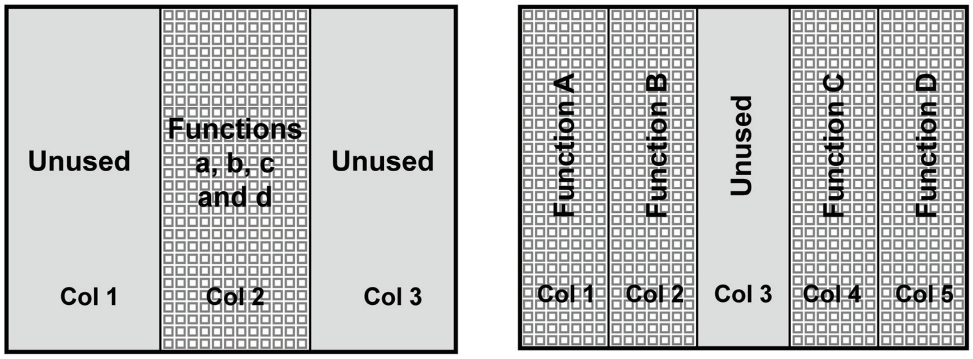
Figure 2-4: Triple Modular Redundancy with Standby Configurations [Zhang et al. 2006]

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

2.2.1.3 Coarse-grained Partitioning

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

Designers may partition the FPGA in one of two ways. If the application is small with respect to the FPGA device, then a *non-overlapping* method can be considered. The non-overlapping scheme separates the FPGA into columns, where one column contains the entire application. The remaining columns are not used by the application and are reserved as spares. As seen in Figure 2-5a, this method generates three distinct configurations, each of which utilizes non-overlapping FPGA resources. More generally, the number of generated configurations is $m+1$, where m equals the number of tolerable faulty columns.



a) Non-overlapping Scheme **b) Overlapping Scheme**
Figure 2-5: Coarse-grained Partitioning Schemes for an FPGA

For larger applications, Figure 2-5b displays a configuration that separates the FPGA application into columns while reserving at least one column as spare. Alternate configurations are generated during design-time so that within each configuration a different column becomes the spare column. In the case of one spare column and four columns containing the application, five distinct configurations are generated. More generally, the number of generated configurations is $\frac{(k+m)!}{k!m!}$, where k is the number of columns containing the application. This scheme is *overlapping* since the various configurations generated overlap in utilizing FPGA resources. Unlike the non-overlapping scheme, some configurations, such as Figure 2-5b, may require horizontal routing resources within the spare column to connect the separated logic resources.

2.2.2 Spare Resources

2.2.2.1 Sub-PLB Spares

Typical FPGA architectures implement logic functions with Look-Up Tables (LUT). As shown in Figure 2-6, *Basic Logic Elements* (BLE) combine each LUT with a flip-flop and output MUX to enable sequential logic implementation. PLBs, in turn, contain multiple BLEs as in the Virtex-4 architecture, which contains eight BLEs per PLB.

By implementing ten benchmark-circuits, Lakamraju and Tessier [2000] found that, on average, 40% of the utilized 4-input LUTs contained one or more spare input. This suggests that an FPGA application contains inherent spares at a finer granularity than the PLB-level as previously discussed. This PLB repair strategy reserves spare BLEs and implements a hierarchy of fault-handling strategies to take advantage of these spare resources, beginning with the finest granularity: LUT input swap, BLE swap, PLB I/O swap, incremental reroute, and complete reroute.

Given the identification of a faulty LUT input by a fault-detection technique, the sub-PLB fault-handling method attempts to swap the faulty resource with a spare input of the same LUT. Figure 2-6 shows input I2 of BLE1 as a faulty LUT input that may be swapped with a spare LUT input such as input I3 to avoid the fault. After swapping the LUT inputs, the contents of the LUT are modified to compensate for the input change. Whereas Figure 2-6 depicts a full PLB input routing matrix, some FPGA architectures contain only a partial routing matrix, restricting the number of PLB inputs to which a given LUT input may connect. For these architectures, the LUT input swapping method must consider whether the spare LUT input has access to the same PLB inputs as the faulty LUT input to prevent rerouting. If spare LUT inputs with similar connections are available, this method is ideal as it does not require logical or connection changes outside of the BLE. If a spare LUT input is not available, then the entire BLE is considered faulty.

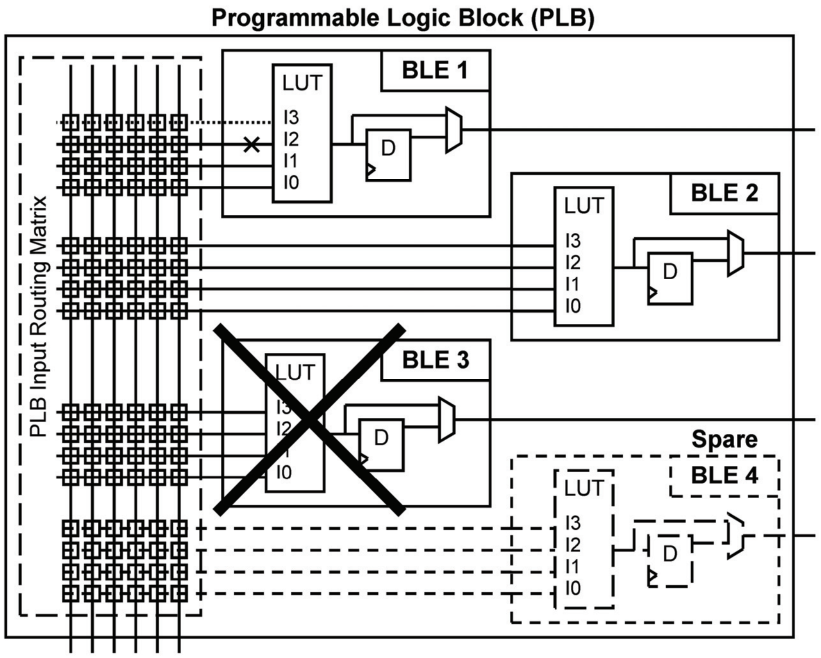


Figure 2-6: PLB Repair Strategies using Sub-PLB spares

When a BLE is considered faulty, as is the case with BLE 3 in Figure 2-6, it is swapped with the reserved spare shown as BLE 4. In the case of partial routing matrices, the BLE swapping method needs to ensure the spare BLE has access to the same PLB inputs as the faulty BLE to prevent rerouting. Figure 2-6 shows that BLE 3 can swap with BLE 4 because of the similarity in connectivity, thus the change only affects the PLB and not the remainder of the circuit. If a spare BLE is not available, then the entire PLB is considered faulty and *incremental rerouting* is required. Incremental rerouting is discussed further in Section 2.3.1.1. Similar to the LUT input swap, faulty PLB input/output wires may be swapped with spare wires that contain similar connections. If a spare PLB input/output wire is not available, then incremental rerouting is required.

2.2.2.2 PLB Spares

To tolerate logic and interconnect faults within a PLB, Hanchek and Dutt [1998] allocate the rightmost PLB of each row as spare. In the case of a fault, a string of PLBs beginning with the faulty PLB is shifted one PLB to the right. More formally, this technique is *node covering*, which allocates a cover PLB to each PLB. In the case of a fault occurring in a PLB, its cover replaces the functionality of the faulty PLB to avoid the fault. This covering continues within a row in a cascading fashion until the spare PLB at the end of the row is reached. For a PLB to become a cover, it must duplicate 1) its logic functionality and 2) its connectivity to other PLBs. Since PLBs within most FPGA architectures are identical, duplicating logic functionality between PLBs is inherent to the FPGA. Hanchek and Dutt ensure that cover cells duplicate connectivity by incorporating reserved wire segments during the design process.

As seen in Figure 2-7, some routing segments are utilized by the initial configuration whereas others are reserved, one of which is located above location 3. As is the case with Fault Scenario A, this reserved segment becomes utilized by the functionality of PLB B by shifting into location 3. Likewise, the two reserved segments above and to the right of location 4 become utilized by PLB D. Additionally, a design may contain inherent reserved segments where some utilized wire segments of the initial configuration also function as reserved wire segments in a fault scenario. This is seen in Fault Scenario A where PLB B allows its utilized wire segment above location 2 to be used by PLB A. During design-time, a custom tool determines the necessary reserved routing segments to enable the FPGA to tolerate one faulty PLB per row. Two heuristics that increase the efficiency of routing include *Segment Reuse* and *Preferred Routing Direction*. Segment Reuse allows a utilized net and a reserved net to map to the same wire segment if the utilized net will move off of the wire segment with the shifting the PLBs, therefore freeing up a wire segment for the reserved net. For nets that

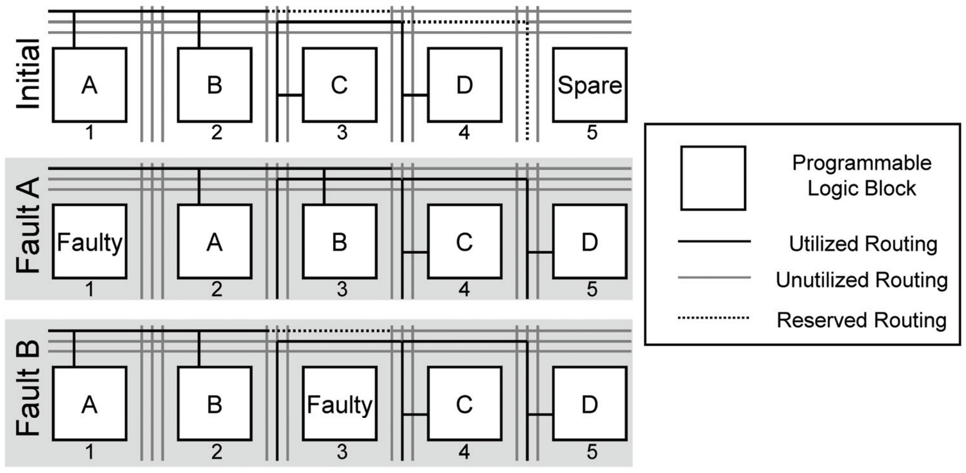


Figure 2-7: Fault scenarios with spare PLBs [Hanchek and Dutt 1998]

cross the FPGA, Preferred Routing Direction encourages the router to extend such nets to the right, horizontally, as far as possible before extending the net in either vertical direction. Providing longer continuous horizontal segments allows greater opportunities for a design to contain inherent reserved segments as discussed above.

Whereas the authors specify that both the logic and interconnect fault-handling technique requires modification to the FPGA architecture and, thus, is intended for manufacturer yield enhancement, end-users may choose to implement the node-covering strategy for tolerating logic faults. Since the design process has ensured that the cover cells can duplicate functionality and connectivity, the routing phase of the place-and-route process is finalized during design-time. To avoid a faulty PLB within a row, an end-user only needs to re-place the PLBs by shifting a row of PLBs into a fault-free configuration. The time to modify an existing configuration by re-placing a row of PLBs is significantly less than the time required either to generate a new configuration from scratch or to incrementally reroute an existing configuration.

2.3 Dynamic Processes

Methods using dynamic processes 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 specific fault scenarios, additional time is necessary to generate appropriate configurations to repair the specific faults. *Offline* recovery methods require the FPGA's removal from an operational status to complete the refurbishment. *Online* recovery methods maintain some degree of data throughput during the fault recovery operation, increasing the system's availability. Sections 2.3.1 and 2.3.2 respectively address these two types of Dynamic Processes.

2.3.1 Offline Recovery Methods

2.3.1.1 Incremental Rerouting Algorithms

The node-covering method discussed in Section 2.2.2.2 avoids a fault by re-placing 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 [2000] present a similar Incremental Rerouting approach that does not require design-time allocated spare resources. The fault recovery method assumes an FPGA to contain PLBs not utilized by the application, thus exploiting unused fault-free resources to replace faulty resources. Upon detecting and diagnosing a logic or interconnection fault by some 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.

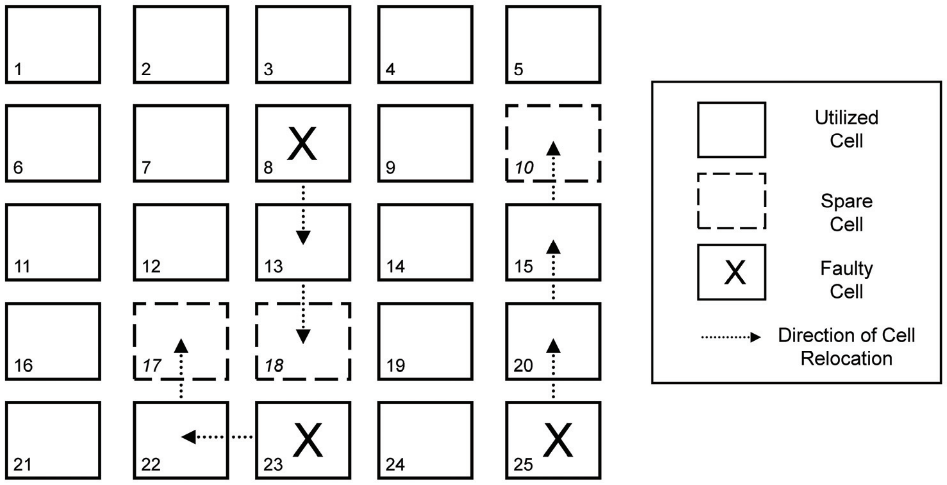


Figure 2-8: One Possible Minimax Fault-Handling Strategy for a 5x5 array

Since faulty PLBs may not be adjacent to a spare resource, a string of PLBs is created logically, starting with the faulty PLB and ending with the PLB adjacent to the spare resource. Figure 2-8 shows one such string, starting with PLB 25, including PLB 20, and ending with PLB 15. To avoid the fault, the string of PLBs shifts away from the faulty resource and towards the spare resource. In the case of node covering, every row has a spare resource so the string of PLBs within the row simply shifts to the right, leaving the faulty resource unused. Since this method does not allocate a spare resource for every row, the string of PLBs may extend into multiple rows to reach a spare PLB as shown in Figure 2-8.

This approach uses *Minimax Grid Matching* (MGM) to determine the optimum re-placement of faulty PLBs. Minimax refers to an algorithm that minimizes the maximum distance, *L*, between the faulty PLB and an unused, fault-free PLB. Beginning with *L* = 1, Figure 2-8 shows that the faulty cell 23 is adjacent to the spare cell 18 and thus a match, but faulty cells 8 and 25 do not have adjacent spares and thus no matches. Incrementing *L* to two, faulty cell 23 matches cell 17 while maintaining its match to cell 18. Additionally, faulty cell 8 matches cell 18 and cell 10 whereas faulty cell

25 still has no matching spare. Incrementing L to three, faulty cell 23 acquires no new matches, faulty cell 8 acquires cell 17 as a match and faulty cell 25 matches cell 10 and cell 18. Since all cells have a match at minimax length $L = 3$, one match is then chosen for each faulty cell. Figure 2-8 depicts one such possibility for the three faulty PLBs, where, for example, the logic in cell 23 shifts to cell 22 and the logic in cell 22 shifts to the spare cell 17.

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

2.3.1.2 Genetic Algorithm Repair

Genetic Algorithms (GA) are inspired by evolutionary behavior of biological systems to produce solutions to computational problems [Mitchell 1996]. Suitable for complex search spaces, GAs have proven valuable in a wide range of multimodal or discontinuous optimization problems. Previous research has investigated the capability of GAs to design digital circuits [Miller et al. 1997] and repair them upon a fault [Keymeulen 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 to 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.

2.3.1.3 Augmented Genetic Algorithm Repair

To decrease the amount of time required to generate a repair, Oreifej et al. [2006] augment Vigander's GA 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.

2.3.2 Online Recovery Methods

2.3.2.1 TMR with Single-Module Repair

In Section 2.2.1.2, faults in TMR arrangements were handled with a-priori, design-time configurations. Methods presented by Ross and Hall [2006], Shanthi et al. [2002], and Garvie and Thompson [2004] address faults dynamically through GA repair. As shown by Figure 2-9, genetic operators and reconfiguration are invoked when a defective module is detected. At design-time, Ross and Hall [2006] produce a population of diverse configurations for implementation. At 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 simple mutation genetic operator is ap-

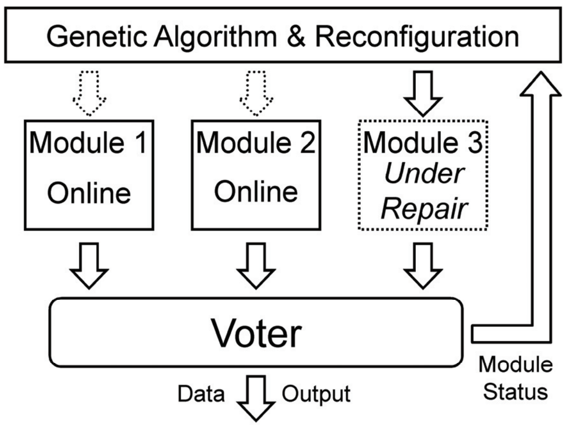


Figure 2-9: Single-Module Repair in TMR Arrangement

plied 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 [2002] 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 [2004] 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 GA has not modified the original configurations to tolerate a permanent fault. To address permanent faults, a (1+1) Evolutionary Strategy [Schwefel and Rudolph 1995] provides a minimal GA, which produces one genetically modified offspring from one parent and chooses the most fit between the parent and offspring. 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.

<u>2.3.2.2 Online Built-in Self Test</u>

Emmert et al. [2007] present an approach that pseudo-exhaustively tests, diagnoses, and reconfigures resources of the FPGA to restore lost functionality due to permanent faults. The application logic handles transient faults through a Concurrent Error Detection (CED) technique and by periodically saving and restoring the system's state through checkpointing. As shown in Figure 2-10, this method partitions the FPGA into an Operational Area and a *Self-Testing ARea* (STAR), which consists 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* (BIST) 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 can test and diagnose PLBs, two STARs are required to test and diagnose programmable interconnect—the H-STAR for horizontal routing resources and the V-STAR for vertical routing resources. Where they intersect, the two STARs may concurrently test both horizontal and vertical routing resources and the connections between them. Since faults

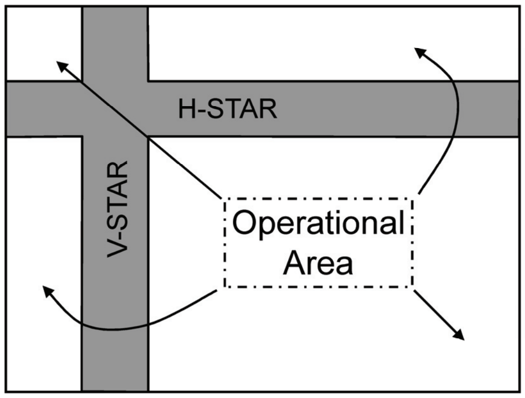


Figure 2-10: Roving STARs within an FPGA

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 when a STAR detects a fault, it 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 and the application can continue 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 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 it to rove the FPGA and test the remaining routing resources.

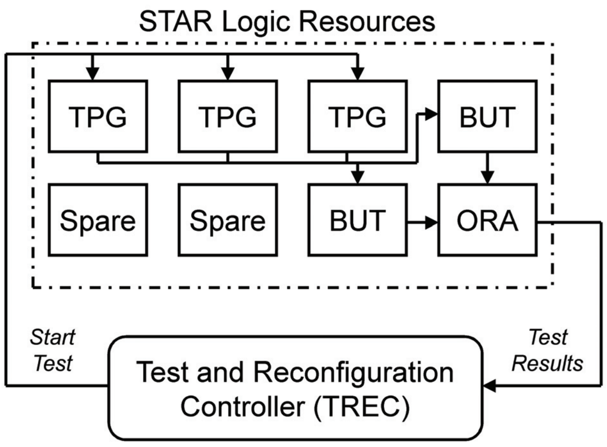


Figure 2-11: 4x2 Array Configured for a PLB BIST

As previously stated, testing and diagnosis occurs within resources of a STAR as shown by Figure 2-11. 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 GA methods utilize an N-MR voting element, the *Competitive Runtime Reconfiguration* (CRR) proposed by DeMara and Zhang [2005] handle faults through a pairwise functional output comparison. Similar to previous GA methods, each of the two individuals is a unique configuration on the target FPGA exhibiting the desired functionality. CRR divides the FPGA into two mutually exclusive regions, allocating the *Left Half* configuration to one individual and the *Right Half* configuration to another individual in the population of alternate configurations.

This detection method realizes a traditional CED arrangement that allocates mutually exclusive resources for each individual. The comparison results in either a discrepancy or a match between half-configuration outputs, which detects any single resource fault with certainty. This indicates the presence or absence of a FPGA resource fault for all inputs that articulate the fault when applied to a combinational logic module or a pipeline stage consisting of combinational logic.

The Left and Right individuals of the pairwise comparison are selected from their respective Left and Right populations to maintain resource exclusivity. Functionally identical, yet physically distinct, *Pristine* individuals developed at design-time compose the initial population. As Figure 2-12 shows, the Left and Right individuals remain Pristine as long as the Left and Right individuals exhibit matching outputs. Additionally, the fitness values of both individuals are increased to encourage selection of individuals exhibiting correct behavior. Upon detecting a discrepant output, however, the fitness state of both individuals are demoted and labeled as *Suspect*. Furthermore, the fitness values of both individuals are decreased to discourage selection of individuals exhibiting discrepant behavior. Over many pairings and evaluations, the fitness value of individuals utilizing faulty resources, and therefore its probability for selection, will be decreased regardless of pairing. Moreover, non-faulty individuals that were previously paired with faulty individuals will eventually be exonerated.

Figure 2-12 shows that the fitness state of individual *i*, which has been labeled as Suspect, is further demoted when its fitness (f_i) drops below the *Repair Threshold* (f_{RT}). Genetic operators are applied to the *Under Repair* individual, until its fitness rises above the *Operational Threshold* (f_{OT}). Selecting an Operational Threshold greater than the Repair Threshold increases confidence that the individual, in fact, is *Refurbished*. Further matching pairings with the Refurbished individual can result in either a *Partial* or *Complete Regeneration* of lost functionality. Nonetheless, if the

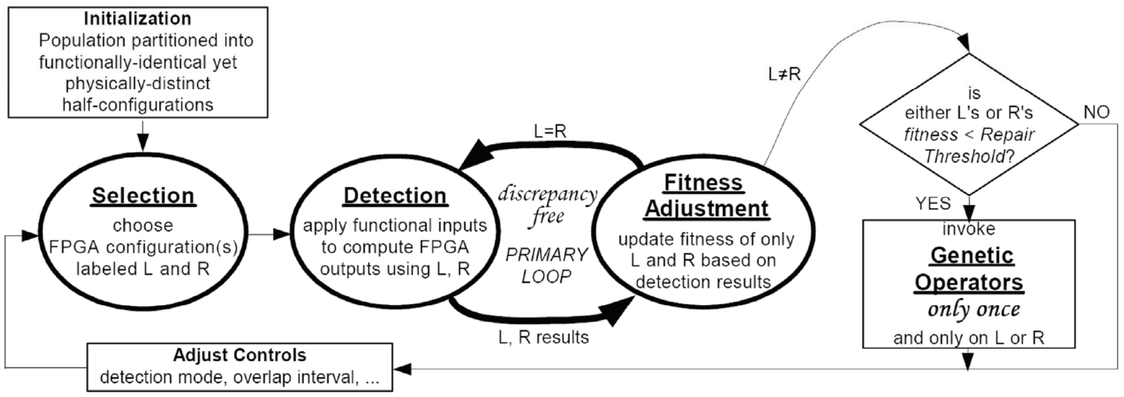

Initialization
Population partitioned into functionally-identical yet physically-distinct half-configurations
Selection
choose FPGA configuration(s) labeled L and R
Detection
apply functional inputs to compute FPGA outputs using L, R
L=R
discrepancy free
PRIMARY LOOP
L, R results
Fitness Adjustment
update fitness of only L and R based on detection results
L≠R
is either L's or R's fitness < Repair Threshold?
NO
YES
invoke
Genetic Operators
only once
and only on L or R
Adjust Controls
detection mode, overlap interval, ...

Figure 2-13: Procedural Flow for Competing Configurations [DeMara and Zhang 2005]

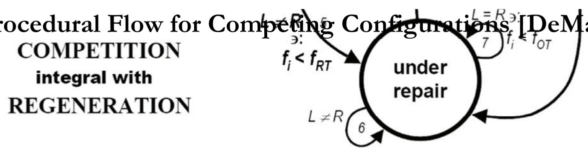


Figure 2-12: States of an Individual during its Lifetime [DeMara and Zhang 2005]

individual exhibits further discrepant behavior, its fitness state is returned to `Under Repair` and genetic operators are reapplied.

Figure 2-13 shows the CRR processes of *Selection*, *Detection*, *Fitness Adjustment*, and *Evolution*. These processes identify individuals utilizing faulty resources and refurbish those individuals in the midst of the fault. The Selection Process determines the two individuals that will occupy the Left and Right regions. Typically, one of the halves is reserved as a "control" configuration where fault-free operational individuals, such as `Pristine`, `Suspect`, and `Refurbished` in that order, are always preferred. The other half supersedes these operational individuals with `Under Repair` individuals at a rate equal to the *Re-introduction Rate*. `Under Repair` individuals that are genetically modified compete by being re-introduced into the operational throughput. The *Re-introduction Rate* can be adjusted to achieve a desired recovery goodput during the repair process. This assumes that alternative configurations exhibiting fault-free behavior over a window of recent inputs remain available or that the GA has already refurbished configurations within the population.

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

CRR exploits the normal operational inputs of the FPGA to evaluate the fitness of individuals. To establish confidence in an individual's fitness, more than one input is evaluated for each individual; the more inputs evaluated, the greater the confidence. Since this method is not an exhaustive evaluation, CRR utilizes an *Evaluation Window* that specifies the number of inputs needed to gain a certain confidence in the individual's fitness. Over many pairings and fitness evaluations, CRR eventually forms a consensus from a population of individuals for a customized fault-specific repair.

2.4 Comparison of Methods

2.4.1 Overhead-related Metrics

2.4.1.1 Resources

Overheads for both logic and interconnect resources are listed in Table 2-II. Resource overheads reported as a percentage of the application are values supplied by the respective authors for those methods. Overheads reported as a percentage of the FPGA are estimates based on the fault-handling strategy using the largest Virtex-4 device—XC4VLX200, 192x116 array—as a lower-bound and the smallest Virtex-4 device—XC4VLX15, 64x24 array—as an upper-bound [Xilinx 2007b]. Some fault-handling methods, such as the Coarse-grained method, partition discrete areas of the FPGA and, thus, do not differentiate between logic and interconnect resource requirements.

2.4.1.2 Operational Delay

Table 2-II lists operational delay values that are reported by the respective authors. Methods utilizing a stochastic repair method such as GAs have an inestimable operational delay.

2.4.1.3 Fault Latency

Once some fault detection technique has detected and located a fault, fault latency specifies the amount of time required for the specified fault-handling method to begin addressing the fault. The authors of the Online BIST method report an upper bound for fault latency as 1.34s for the ORCA OR2C15A FPGA, a 20x20 PLB array. Considering both the increased size and faster boundary scan clock of the XC4VLX200, the estimated fault latency is listed Table 2-II.

2.4.1.4 Unavailability

Since all fault-handling methods discussed address faults through FPGA reconfiguration, a portion of the unavailability is due to the reconfiguration time. Configuration times are calculated for the largest Xilinx Virtex-4 device, XC4VLX200, as an upper bound. The size of a full configuration file for this device is 6.12 MB. Using the Virtex-4 SelectMAP byte-wide parallel interface and a 100MHz configuration clock, configuration times are calculated using the following equation derived from [Xilinx 2008]:

$$T_{config} = (bytes + 3) \cdot \frac{1}{f_{cclk}}, \qquad \textbf{(2-1)}$$

where $bytes$ is the size of the configuration file in bytes and f_{cclk} is the frequency of the configuration clock. A full configuration for the XC4VLX200 device, thus, requires 64 ms, which is reported below in *italics*. In the cases where partial configurations are used such as the fine-grained method, configuration times are calculated from a partial configuration file, with a 16 PLB minimum configuration height for the Virtex-4 architecture [Lysaght et al. 2006] and a 116 PLB maximum width of the XC4VLX200 device. Given a partial bitstream size of 0.5 MB, the partial configuration time is 6 ms. To recover from a fault, Competing Configurations may require cycling through its entire population of half-configurations to implement a configuration not utilizing a faulty resource. If all configurations are adversely affected by the fault, then its stochastic repair process is unbounded.

2.4.1.5 Recovery Goodput

As most methods suggest the system be in an offline state during the entire fault-repair process, a particular fault may only articulate itself in a small percentage of the output space. In such a situation, an application with low sensitivity to faulty inputs may benefit from the faulty system remaining in an operational state during the fault-repair process. As defined in Table 2-I, recovery goodput measures the number of correct or useful outputs provided during the repair process [Sharma et al. 2007]. The total repair time is the sum of the Fault Latency and Unavailability metrics listed in Table 2-II. For methods with total repair times greater than their italicized reconfiguration times, goodput applies. Whereas goodput measurements are largely a result of the type of fault and application, most fault-handling methods do not consider goodput during fault recovery. As previously discussed, the Competing Configurations method manages a required goodput by adjusting the rate at which configurations under repair are implemented on the FPGA.

Table 2-II: Summary of Overhead-related Metrics

	Metrics				
	Resources		Operational Delay	Fault Latency	Unavailability for single fault
	Logic	Inter-connect			
Fine-grained	**2–10% of application**	**Not Addressed**	**14–45%**	**None**	***6 ms***
Medium-grained	**300% of application**		**Not Addressed**	**None**	**None**
Coarse-grained	**4–50% of FPGA**		**11–18%**	**None**	***64 ms***
Sub-PLB Spares	**0–20% of application**	**Not Addressed**	**Not Addressed**	**None**	**Place&Route +** ***64 ms***
PLB Spares	**1–41% of FPGA**	**31–43% of application**	**5–10%**	**None**	**Place+** ***64 ms***
Incremental Rerouting	**None**		**48–53ns**	**None**	**2–12 s+** ***64 ms***
GA Repair	**None**		**Inestimable**	**None**	**Unbounded**
Augmented GA Repair	**None**		**Inestimable**	**None**	**37% decrease from GA repair**
TMR w/ Single Module Repair	**300% of application**		**Inestimable**	**None**	**None**
Online BIST	**4–11% of FPGA**		**0–20%**	**0–15 s**	**None**
Competing Configurations	**200% of application**		**Inestimable**	**None**	**popSize****64 ms* **or Unbounded**

2.4.2 Sustainability Metrics

2.4.2.1 Fault Occlusion

By nature, all fault-handling methods typically bypass faulty resources. Techniques that reuse or exploit faulty resources further increase system reliability by converting previously ignored resources into conditionally available resources.

2.4.2.2 Repair Granularity

The repair granularity metric specifies the resolution with which a fault may be handled. Methods capable of addressing faulty resources finer in granularity and occlude faults by exploiting those resources further increase system reliability.

2.4.2.3 Fault Tolerance

Each method is capable of handling varying number and types of faults. Fault tolerance specifies the maximum number of faults handled.

2.4.2.4 Fault Coverage

As discussed in Section 1.2, transient faults are typically addressed by some scrubbing scheme. Whereas some fault-handling methods explicitly incorporate scrubbing or rollback [Emmert et al. 2007; Garvie and Thompson 2004; Ratter 2004], other fault-handling techniques may indirectly address transient faults by handling them as permanent faults and reconfiguring the faulty portion of FPGA to scrub away the fault. Table 2-III lists whether a fault-handling method addresses logic, L, or interconnect, I, faults.

2.4.2.5 Critical Requirements

Fault-handling components that may or may not be implemented by the FPGA itself are external requirements. Additionally, these external requirements are relied upon as fault-free making them critical requirements. Given that some FPGAs such as the Virtex-4 XC4VFX140 include embedded microprocessors or can realize such hardware equivalents with its PLB logic and interconnect, the FPGA device may implement processing functions such as Place and Route.

Table 2-III: Summary of Sustainability Metrics

	Metrics					
	Fault Occlusion	Repair Granularity	Fault Tolerance	Fault Coverage		Critical Requirements
				L	I	
Fine-grained	**Bypass**	**PLB**	**Single faulty PLB per tile**	✓		**Storage of Configurations**
Medium-grained	**Bypass**	**Small group of PLBs**	**Single faulty group of PLBs**	✓	✓	**Voter, Storage of Configs**
Coarse-grained	**Bypass**	**Large group of PLBs**	**Single faulty group of PLBs**	✓		**Storage of Configurations**
Sub-PLB Spares	**Bypass**	**Look-up Table**	**Single faulty resource per spare resource**	✓	✓	**Custom Placer and Router**
PLB Spares	**Bypass**	**PLB**	**Single faulty PLB per row**	✓		**Custom Placer**
Incremental Rerouting	**Bypass**	**PLB**	**Single faulty PLB per spare PLB**	✓	✓	**Custom Placer and Router**
GA Repair	**Exploit, Bypass**	**Variable**	**Indeterminate**	✓	✓	**Processor and Memory**
Augmented GA Repair	**Exploit, Bypass**	**Variable**	**Indeterminate**	✓	✓	**Processor and Memory**
TMR w/ Single Module Repair	**Bypass, Exploit**	**Variable**	**Indeterminate**	✓	✓	**Voter, Processor and Memory**
Online BIST	**Bypass, Exploit**	**Look-up Table**	**2 faulty PLB columns & 2 faulty rows**	✓	✓	**Processor and Memory**
Competing Configurations	**Bypass, Exploit**	**Variable**	**Indeterminate**	✓	✓	**Processor and Memory**

2.5 Chapter Summary

Methods that do not change the configuration of an FPGA during run-time are mentioned as being limited in the number and type of faults it can handle. As such, more robust fault-handling methods that reprogram the FPGA with a modified configuration are discussed and placed within the classification depicted by Figure 2-2. Since A-priori Allocation fault-handling strategies exploit the redundancy of the FPGA architecture, the number of faults they can handle, as indicated by the fault tolerance metric, is limited to a smaller number in a given area. This static fault-handling nature, however, does increase its availability since less time is required to determine a solution to handle the fault. Dynamic Processes, on the other hand, can adapt its fault-handling strategy and tailor its repair solution to a variety of fault scenarios. Whereas this capability increases its fault tolerance, its unavailability and critical requirements also increase to determine an appropriate repair solution.

As made evident by the metric tables in Section 2.4, no one method provides the best performance in all situations. Applications with strict availability requirements should consider A-priori Allocation methods due to the small amount of processing required to either reconfigure the FPGA with a stored configuration or modify the configuration by a pre-determined process. Applications with more lenient availability requirements and extended mission times may consider Dynamic Processes due to their need to handle many more fault scenarios. Perhaps the best solution is some combination of the methods discussed. For example, the coarse-grained partitioning approach quickly recovers from a fault by implementing a configuration with a spare column over the faulty resource. While the application continues to operate, an online BIST could test the resources of this spare column to search for the faulty resource. Upon detecting the specific resource that is faulty, Sub-PLB spares or Incremental Rerouting could modify all configurations that utilize the faulty resource to avoid it. After such a modification, a configuration could be implemented to return the application to its original redundancy level. Combining methods for a hybrid approach exploits the fast reconfiguration of an A-priori Allocation method with the dynamic fault tolerance of Dynamic Processes.

CHAPTER 3
EVOLVABLE HARDWARE OPTIMIZATION STRATEGIES

3.1 Genetic Algorithms

As previously discussed in Section 2.3.1.2, Genetic Algorithms (GA) have demonstrated proficiency for locating global optimums in large, discontinuous, and multimodal search landscapes. The search space of repairing a damaged digital circuit residing on an FPGA can be considered large, discontinuous, and multimodal. Hence, a GA may be an appropriate method to develop solutions that restore lost functionality. In attempting to apply GAs to the fault-repair problem domain, many techniques have developed. As evident by the number of methods proposed for repairing FPGAs, no ideal chromosomal representation has been developed, neither has an optimum set of Genetic Operators been defined.

Miller et al. [1997] investigated the specific ability of GAs to design innovative configurations of simple digital circuits. Their research resulted in successful designs of a 4-bit full adder and a 2-bit multiplier. Keymeulen et al. [2000] investigated the ability of GAs to evolve fault tolerant designs of an analog multiplier and a digital XNOR circuit, both created from an array of programmable transistors. When the fault tolerant designs could not perform acceptably, they successfully used GAs to evolve a repair to address the fault. Vigander [2001] expressed difficulty in evolving a complete repair of a 4-bit multiplier. To circumvent this difficulty, Vigander proposed a voting scheme where the majority vote of three partially repaired circuits is used for the functional output of the circuit. Lohn et al. [2003] utilize a representation similar to the FPGA architecture to repair a faulty Quadrature Decoder. Implementing 1) a more dynamic fitness function and 2) a higher-level chromosome representation allowed a greater degree of success for finding a correct solution.

Previous research has attempted to parallelize the GA to decrease running time and improve the quality of solutions. As first analyzed by Grosso [1985], partitioning the GA population into smaller, isolated sub-populations allowed favorable traits to spread more quickly throughout the smaller sub-populations, yet these smaller sub-populations each produced less fit individuals upon convergence. Allowing individuals to migrate between sub-populations at slow rates did not improve performance whereas intermediate rates improved the fitness of individuals upon convergence. Pettey et al. [1987] further investigated the effect migration has on population performance. By migrating the best individual from each sub-population to its neighbors after every generation, the parallel GA produced individuals of similar fitness with that of a traditional GA. As investigated below, a particular form of a parallel GA is used to surpass the performance of the traditional GA in the FPGA fault-repair problem domain.

3.2 Age-Layered Population Structure Overview

Populations of typical GAs quickly converge upon a single local optimum, which may not be the global optimum. Research has attempted to prevent convergence and promote diversity by 1) restricting breeding to similar individuals, 2) modifying the replacement method, or 3) adjusting the fitness function to penalize individuals that are similar to existing individuals. Each of these methods depends upon a similarity function to determine whether two individuals should mate, replace one another, or be penalized for being similar. GAs utilizing bitstring representations can readily assess similarity by a simple hamming distance calculation, although similarity in bitstrings may not necessarily translate to functional similarity. Other algorithms used to determine functional similarity might be too computationally expensive to be competitive with other searching algorithms.

As an alternative method to prevent convergence of populations, an Age-Layered Population Structure (ALPS) creates multiple sub-populations that partition individuals by age [Hornby 2006]. The age of an individual is set to zero during the initialization of a population and increments every time the GA selects the individual as a parent. Additionally, offspring created by parents receive the age of its oldest parent and furthermore increases the age by one. Therefore, age of individuals within the population becomes a measure of the time certain genetic material has existed within the population.

The age of an individual within ALPS is used to 1) partition the population into age layers to systematical replace the bottom age-level with random individuals and 2) restrict breeding to similarly aged individuals. Standard GAs typically escape basins of attraction through the mutation operator by introducing new genetic material into the population. As a more drastic method of preventing the population from converging on non-global maximums, ALPS supplements the mutation operator by replacing the bottom age-level with random individuals at regular intervals, enabling the GA to explore new areas of the search space. Additionally, genetic operators such as crossover use age to restrict selection of parents from within the same age-level or the age-level immediately below. Restricting breeding to individuals of similar ages prevents individuals, which contain more mature genetic material, from dominating the entire population and permits other local optimums to be explored within lower age-levels. Experiments in Section 3.5.1 explain and demonstrate this benefit in detail.

Research provided by Hornby verifies that ALPS outperforms a standard GA and two other GA implementations for evolving antenna designs from scratch. Whereas the results are clear, some implementation details of ALPS remain ambiguous. The implementation presented in Section 3.3 addresses these ambiguities and introduces additional parameters to improve performance. The experimental setup described in Section 3.4 applies the ALPS implementation to an FPGA repair problem to verify its viability for this problem domain. Section 3.5 reveals the benefits in performance of preventing convergence of the population and furthermore optimizes new ALPS parameters for the FPGA repair problem.

3.3 ALPS Implementation

Similar to a typical GA, ALPS follows a similar overall process flow. The GA initializes the population then subjects the population to a fixed number of generations, as specified by the *genNum* parameter in Figure 3-1. Within each generation, Selection, Genetic Operators, and Fitness Evaluation all repeat to form a new population equal in size to the original population, as specified by the *popSize* parameter. The *Representation* used along with each of the process modules of *Initialize Population*, *Evaluate Fitness*, *Selection*, *Genetic Operators*, and *Replacement* are explained in detail below.

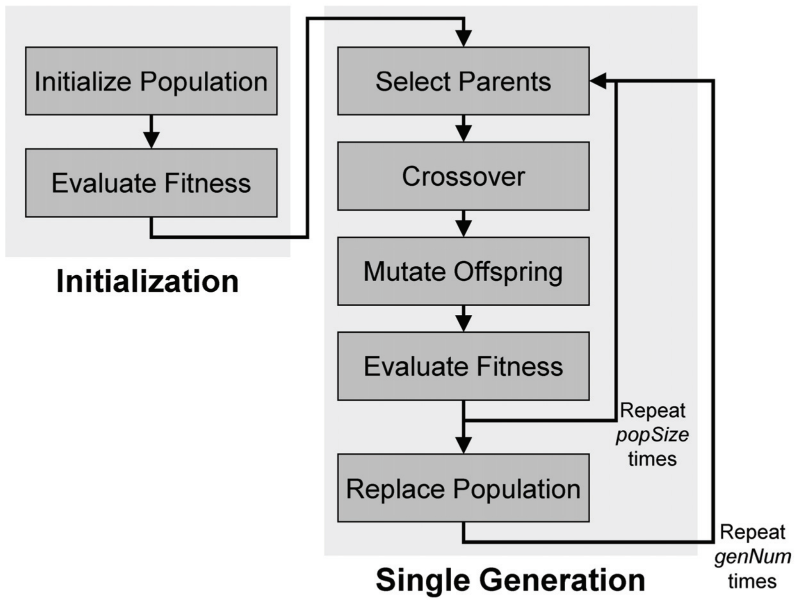


Figure 3-1: Standard Genetic Algorithm and ALPS Process Flow

3.3.1 Chromosome Representation

The ALPS FPGA repair system is implemented by Java code. Appropriate for an object-oriented programming language, the representations of the chromosomes are objects that mimic the architecture of the FPGA. As such, Figure 3-2 shows a chromosome as an array of LUT objects, each of which contains 4 inputs and a 16x1 memory to describe the behavior of the LUT. Each of the four inputs may be connected to either GND, VCC, a circuit input, or the output of another LUT. To simplify the fitness evaluation process, all individuals evaluated by the fitness function are verified to constitute a feed-forward network. This verification process merely checks that a LUT input connects to a LUT output with a lower-numbered label. As seen in Figure 3-2, Input 3 of LUT5 is connected to the output of LUT2, which is a valid connection to maintain a feed-forward network. Whereas it is possible that a LUT input be connected to a LUT output of a higher label ID value while maintaining a feed-forward network, this restrictive, yet simple, verification minimizes the impact on computation time as previously implemented by Miller et al. [1997] and Vigander

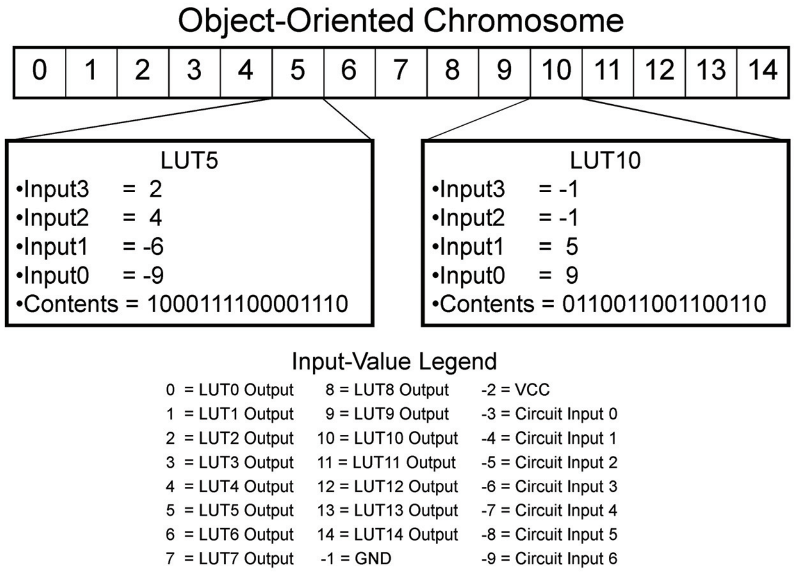


Figure 3-2: Detailed View of the GA Chromosome

[2001]. In case that the verification detects an non feed-forward individual as defined above, the invalid LUT input is modified to be a random, valid input—GND, VCC, a circuit input, or the output of a LUT with an lesser ID value.

3.3.2 Initialize Population

Since the goal of these experiments is FPGA repair, an existing working individual is presupposed. To allow repair, the initial population is seeded with 10 identical individuals exhibiting 100% functional behavior whereas random individuals populate the remaining portion. For the ALPS implementation, a population size of 100 per each age-level is used, where 10 seed individuals and 90 random individuals constitute the bottom age-level upon initialization. During initialization, only the bottom age-level is populated, allowing the age sort function discussed in Section 3.3.6 to populate additional age-levels as needed up to the limit of 10, as specified by the *age-level* parameter. As suggested by Hornby, the standard GA implementation uses a population size of 1000 to mimic the capacity of the ten-level ALPS implementation, where 10 seed individuals and 990 random individuals constitute the initial population.

Additionally, the FPGA repair process is initiated after a fault has occurred. To simulate a fault, a random LUT is selected from the available LUTs and one I/O port is held constant at logic 0 or 1. If the fault has no effect on the circuit, another fault is generated by randomly choosing another LUT, I/O port, and logic stuck-at level. This process repeats until the fault is manifested in the output. As specified by the parameters, only one manifested fault is implemented in the experiments. Generally speaking, if a fault only effects a small subset of the application's outputs, then the GA will most likely only have to make a small change to the chromosome of the ideal individual to repair the fault. To allow room for the GA to search, the GA is allocated 115% of the LUTs necessary to implement the initial optimized circuit design. As specified later in Section 3.4, the initial circuit design utilizes 13 LUTs and, therefore, the GA is allocated 15 LUTs to repair the fault.

3.3.3 Evaluate Fitness

The representation described in Section 3.3.1 explicitly defines the circuit inputs whereas the circuit outputs are not. Since the outputs are undefined, two fitness evaluation techniques are possible, fixed outputs and floating outputs. When evaluating individuals, the fixed-output fitness evaluation utilizes as outputs the same LUTs specified by the seeded individual, which exhibits correct functional behavior prior to a fault. This technique is useful when the output LUTs cannot be modified and, therefore, encourages individuals to utilize the same circuit outputs as defined by the initial individual. This method can be seen as an additional restriction that the GA must overcome in finding an acceptable solution within the search space.

In an attempt of increasing the scope of the search, Lohn et al. [2003] devised a second fitness evaluation method called *floating outputs*. In this scheme, the circuit outputs of each individual are not defined until fitness evaluation. The exhaustive set of input vectors are applied to each individual and the output responses of each LUT are retained. After all inputs are applied, the response of each LUT then is evaluated against each bit-wide output to determine which LUT is the best match for a given bit-wide output. In cases where the output LUTs may be relocated, increasing the chances of finding a fully functional repair may justify the increased computation time. The experiments investigated in this work solely make use of the fixed outputs fitness evaluation method.

After the replacement process creates the final, new population, the chromosomes are sorted in ascending order of raw fitness values and are assigned a rank where the highest fit individual receives the highest rank value. The rank is used to calculate the proportional fitness of an individual as

$rank / \sum_{population} rank$. This proportional fitness value has the same effect as placing all individuals within the population on a roulette wheel for the selection process, each with a specific probability.

3.3.4 Selection

In the implementation proposed by Hornby, tournament selection is used for both the standard GA and ALPS. For the standard GA in this implementation, selection of both parents functions as a roulette wheel, where a random, fractional value between 0 and 1 is selected and the proportional fitness values of individuals within a population are accumulated until the sum is larger than the selected random value. As such, individuals with larger fitness values have a higher rank and, thus, larger proportional values, giving them a greater chance for selection.

In the ALPS implementation, the first parent is selected from the current age-level using proportional fitness selection as the standard GA. The second parent, however, may be selected from the current age-level or the age-level immediately below the current level. Two selection processes are investigated by the experiments. The first method, which is assumed to be used by Hornby, *combines* both the current age layer sub-population with the one immediately below it. Whereas Hornby uses a tournament selection, this implementation continues use of proportional fitness values. After the merge, it calculates the scaled fitness values for the combination of the two sub-populations and assigns proportional fitness values accordingly.

This paper proposes a second method of ALPS selection that keeps the two age-levels *separate* by selecting one 2nd Parent candidate from the current age-level using proportional selection and then selecting another 2nd Parent candidate from the age-level immediately below using proportional selection. The GA then randomly chooses between the two candidates, favoring the candidate from the current age-level with a probability of 0.25, 0.50, 0.75, or 1.00 as specified by the *Age-level Man-*

agement parameter. A probability of 0.25 slightly favors selecting the second parent from the age-level below whereas a 0.75 probability slightly favors selecting the second parent from the same age-level. A 0.50 probability does not favor either age-level and a probability of 1.00 always favors the same age-level, preventing cross-level breeding. The results of each of these probabilities are investigated by experiments detailed in Section 3.5.3.

3.3.5 Genetic Operators

3.3.5.1 Crossover

For the object-oriented representation previously described, crossover points reside between two LUT objects within the array. Two-point crossover has the ability to create either one or two offspring from two parents. To produce one offspring, Parent 1 receives the genetic material of Parent 2 that is located within the two crossover points. A second offspring is produced when Parent 2 receives the genetic material of Parent 1 that is located within the two crossover points. In other words, a second offspring is produced when Parent 1 receives the genetic material of Parent 2 that is located outside of the two crossover points. The experiments conducted utilize 2-pt crossover at a rate of 0.90 and produce one offspring per crossover operation. Upon producing an offspring, the chromosome is validated as a feed-forward circuit and modified accordingly. Together the selection process and crossover operator repeat, producing a number of offspring equal to the current population size. Whenever an individual contributes genetic material by being a parent in crossover, its age is increased by the value of 1 per generation, regardless of the number of times it is used as a parent in a generation. After increasing the ages of the parents accordingly, the offspring inherits the greater age of the two parents and then increases its own age by the value of 1.

3.3.5.2 Mutation

Since crossover cannot modify the contents of an LUT, connectivity or routing, mutation is given more pressure to introduce variation to the individual LUTs by changing the LUT inputs to a random value that maintains the feed-forward network or by mutating the LUT contents on a bit-by-bit basis. Each LUT object of a newly produced offspring chromosome is given a chance for mutation. Each input and each bit of the 16-bit LUT contents is changed with a 0.005 probability as specified by the mutation rate. If an input is mutated, the input is changed to a random input that maintains a feed-forward circuit—GND, VCC, circuit input, or an output of a LUT with a lesser ID value. If a bit in the contents is mutated, a random Boolean value is selected. Mutations, even if they occur, may result in no change if the input or Boolean values selected are the same as the previous values.

3.3.6 Replacement

As suggested by Hornby, a polynomial progression is used to define the age limits of the ten age-levels as 1, 2, 4, 9, 16, 25, 36, 49, 64, and ∞, respectively. To increase further the separation between age-levels, this polynomial progression is multiplied by an *agegap* parameter of 20, resulting in age limits of 20, 40, 80, 180, etc. These age limits are used to determine which age-level offspring should reside as its age increases.

To store the offspring created by the genetic operators, a new population containing the same number of age-levels as the original population is created. Offspring are placed within the age-level of their oldest parent upon creation. As suggested by Hornby, the ages of both parents, are increased by 1 since they have donated genetic material. In cases where the age of the oldest parent is equal to the age-level limit, the offspring that receives this increased age still is placed into the current age-level. Additionally, the age of the offspring is increased by 1 after receiving its age, which further exceeds the age limit. After selection and crossover are complete for a population, a sort based on age is performed on the new population to create new age-levels and redistribute individuals into the correct age-levels as necessary.

After the fitness of all individuals within the population have been calculated, the top number of individuals as specified by the *elitism* parameter are marked as Elite individuals and are removed from their respective population, original or new. If the age sort function discussed above causes an age-level to exceed the population limit as specified by the *popsize* parameter, individuals are removed at random until the population becomes less than the sum of the population size and elitism size. Next, the elite individuals are moved into the new population by considering each age-level in sequence. The ALPS implementation marks one elite individual per age-level totaling ten elite individuals, whereas the standard GA utilizes 10 elite individuals.

As previously discussed, individuals increasing in age may move into higher age-levels, causing the size of age-levels in the new population to decrease below the population size specified in the parameters. If this is the case, the GA moves individuals of the same age-level from the original population into the new population. Since most of the individuals of the original population were parents of the offspring in the new population, the original population must also undergo an age sort function to redistribute individuals in the case that a parent contributing genetic material to an offspring causes it exceed the age limit of its level. After the age sort function, individuals are randomly selected to be moved into the new population until either the new population is full or the age-level of the original population becomes depleted. The last step of the replacement process is to replace the entire bottom age-level with random individuals every 20 generations as specified by the *agegap* parameter.

3.3.7 GA Parameter Summary

Below is a summary of the GA Parameters described above, some of which only apply to ALPS implementations. Along with a comparison between the ALPS and standard GA implementations, various implementations of the Age-level Management parameter are investigated.

Table 3-I: Summary of GA Parameters

Number of Age Levels	Standard	1
	ALPS	10
Population Size	Standard	1000
	ALPS	100 per each age-level
Number of Faults		1
Max # of LUTs		15
Fitness Evaluation		Fixed Outputs
Fitness Scaling		Rank
Selection		Proportional
Age-level Management during Selection		Combined
		Separate (0.25, 0.50, 0.75, 1.00 probabilities)
Crossover Type		2-pt, 1 offspring produced
Crossover Rate		0.90
Mutation Rate		0.005
Elitism	Standard	10
	ALPS	1 per each age-level
Age Gap		20 generations

3.4 Experimental Setup

Figure 3-3 illustrates the 3-bit full adder circuit implemented with LUTs. The legend in the figure shows each LUT having an ID, four 1-bit inputs, one 1-bit output, and sixteen 1-bit memory locations. For readability of the connectivity, the order of the LUTs as they appear in the chromosome is rearranged. As discussed in Section 1.1, the inputs of the LUT are used as an address to determine which of the memory locations is utilized as the LUT output. The 3-bit full adder has two 3-bit inputs (A & B) with a carry (C) and one 4-bit output (OUT). LUT inputs either connect to input A, input B, carry C, the output of another LUT, GND, or VCC.

The fitness of individuals within the population is calculated by exhaustive evaluation, where

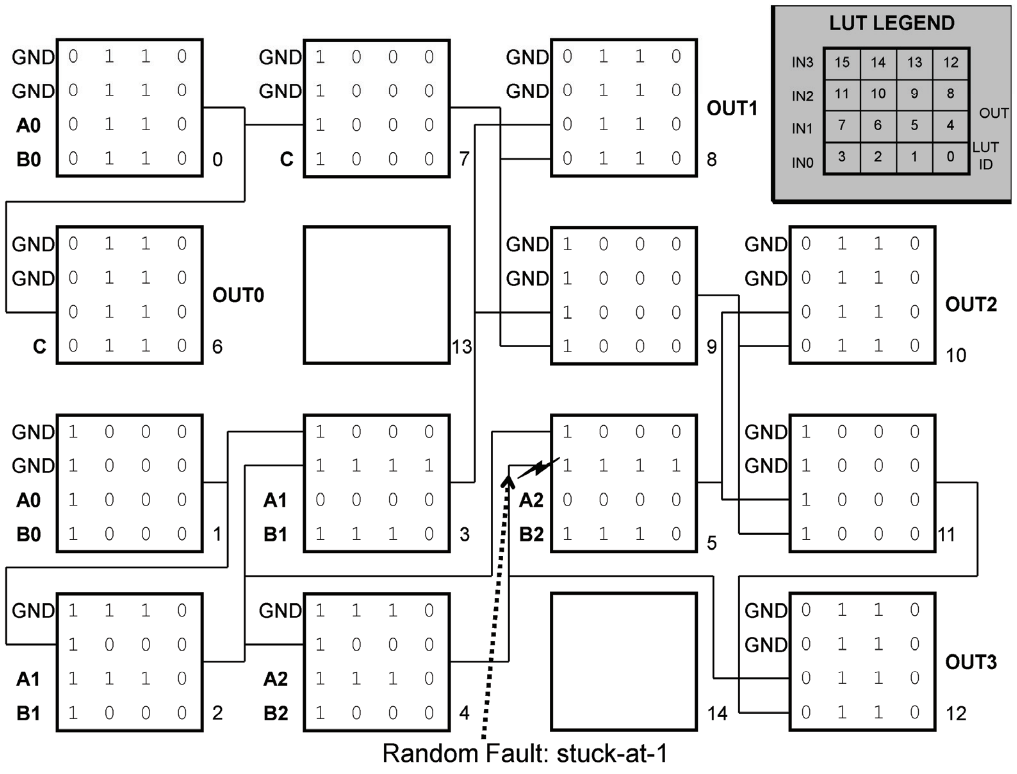


Figure 3-3: 3-bit Full Adder Implementation
(┼ wire junctions are not connected)

the functionality of the FPGA is simulated by applying, in turn, the exhaustive set of inputs and comparing the results to the 3-bit full adder truth table. As such, the raw fitness value for an individual is the sum of correct responses for the entire output space. The output space of a logic circuit is $2^{inputs} \times outputs$, where *inputs* equals 7 and *outputs* equals 4 for the 3-bit full adder. Thus, the maximum fitness value an individual may have is $2^7 \times 4 = 512$.

To simulate a stuck-at fault, a random LUT input is forced to be logic 0 or 1. As seen in Figure 3-3, the randomly generated fault for these experiments is a stuck-at-1 fault on Input 2 of LUT 5. This fault causes the fitness of the 3-bit full adder to drop from 512 to 452. The experiments compare the performance of the standard GA implementation with various ALPS implementations as previously discussed.

3.5 Experimental Results

Unless otherwise specified, the experiments shown below include 3000 generations per run to discover long-term trends of the standard GA and ALPS, where each run is repeated 100 times to demonstrate statistical significance in the performance of the various GA implementations.

3.5.1 ALPS Overview

Figure 3-4 shows the first 200 generations of a single run of ALPS with the best individuals of each age-level at each generation number. It reveals how individuals increase in age and are promoted to higher age-levels. As discussed later in detail, it also reveals how ALPS decreases the influence the best overall individual has on the population as a whole.

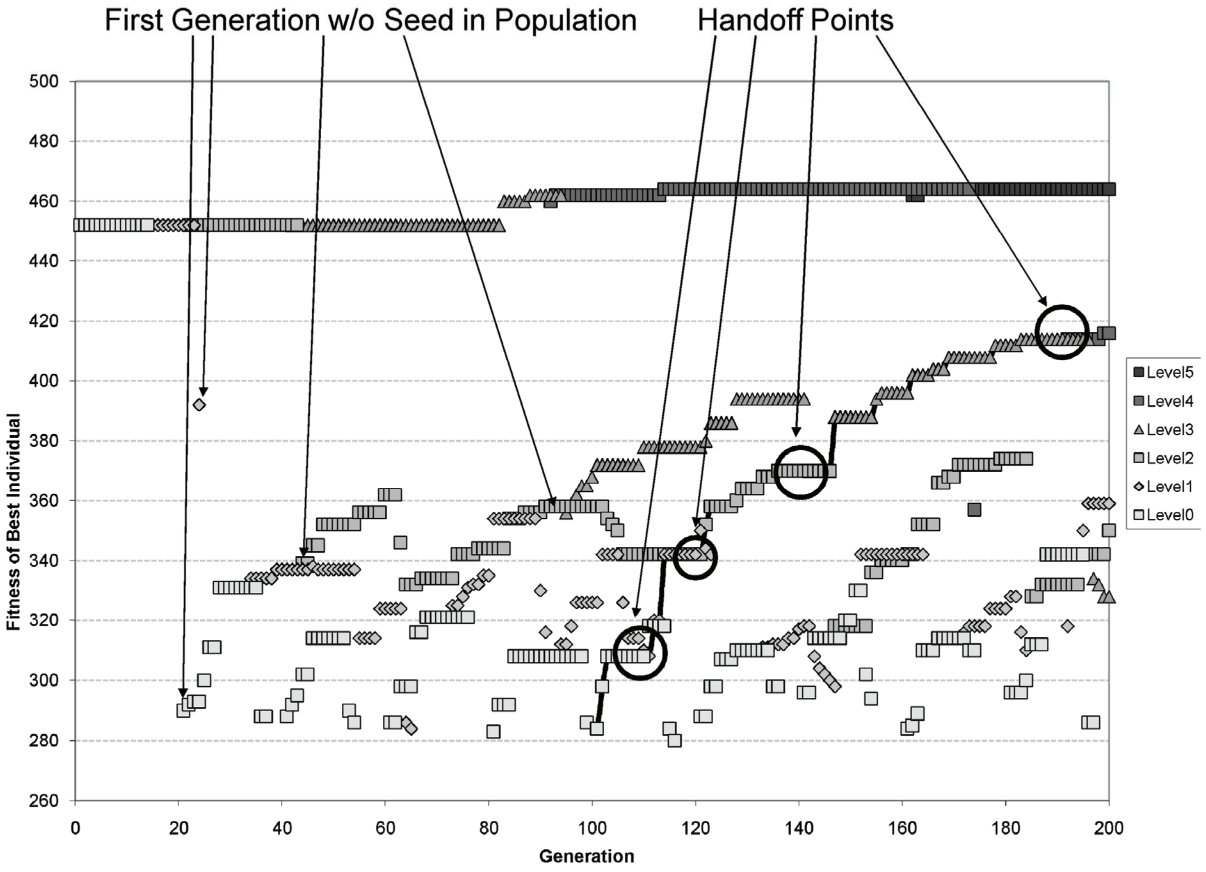


Figure 3-4: Best Individuals of each Age-level during the Initial Generations

At Generation 1, the seeded individual is seen to exist within age-level 0 with a starting fitness of 452. Within 15 generations, at least one of the 10 seeded individuals ages-out and becomes the first individual to reside within age-level 1. Within another 20 generations, it quickly moves into age-levels 2 and 3. The same elite individual remains the best of the entire population until Generation 80 when the GA creates a higher-fit individual with fitness of 460. This individual continues to undergo genetic operators and, thus, further increase in fitness.

As the seeded individual becomes older and moves to higher age-levels, the lower age-levels change their focus from repairing the seeded individual to designing an individual from scratch. The arrows in Figure 3-4 show the first generation that these lower age-levels do not have the seeded individual within its population so they can begin evolving solutions from randomly generated individuals. These individuals, then, become older eventually reaching the higher age-levels to compete with the original, seeded design and contribute genetic information. The evolution of a randomly generated individual is highlighted with connecting lines. As shown by the figure, the individual is randomly generated in the bottom age-level at Generation 100 and moves into higher age-levels, as shown by the handoff points. As previously discussed, the bottom layer is replaced with random individuals every 20 generations as specified by the *agegap* parameter, a cyclic pattern constantly supplying the GA with new information.

When the entire single run is considered, additional individuals are seen to evolve from the bottom age-level and increase in age to compete in higher age-levels. While the seeded individual evolves from a fitness of 452 to a fitness of 495 in the top age-level, Figure 3-5 shows multiple candidate solutions being evolved from scratch, six of which are shown by black lines. Age levels 1, 3, and 5 are omitted from the figure for visual clarity. Since the seeded individual is kept within its own sub-population, the other sub-populations are free from its dominance and can provide alternative solutions that may reside in areas outside of the basin of attraction that the GA is currently searching. Interestingly, an alternative candidate moving from age-level 8 to age-level 9 at Generation 1400 coincides with a better-fit individual being produced in age-level 9, a direct benefit caused by the newly developed genetic material entering the sub-population.

The results of this single run also reveal three coarse partitions of the fitness landscape. The bottom partition extends from fitness 260 to 340 and represents fitness values easily obtained by randomly generated individuals and a minimal number of genetic operators. The middle division from 340 to 420 represents fitness values that randomly generated individuals cannot attain and, thus, require a moderate number genetic operations applied. Individuals within the top division are

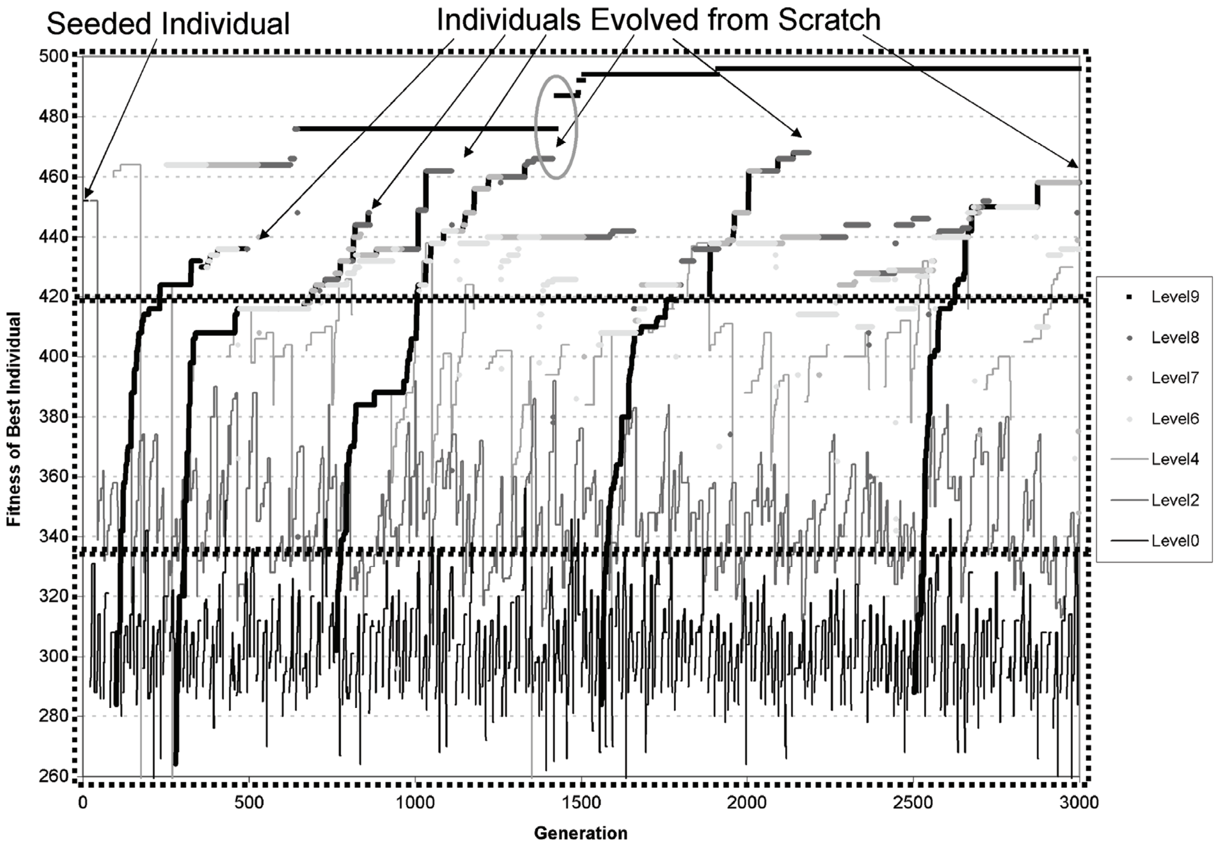


Figure 3-5: Best Individuals of each Age-level at each Generation

rarer since fitness values from 420 to 512 are the most difficult to obtain. Figure 3-5 reveals that strong individuals are produced approximately once every 700 generations and move to the top age-level to contribute genetic material to the overall best individual.

3.5.2 Standard GA and ALPS Comparison

To gauge the effectiveness of ALPS, 100 runs of the standard GA were completed using the GA parameters previously specified by Table 3-I, producing a best overall individual at each generation for each run. Figure 3-6 averages the 100 best individuals for a given generation and compares them to the *Combined* implementation of ALPS. Whereas the standard GA consistently provides an individual with fitness of 480 within 100 generations, ALPS requires 1500 generations to achieve the

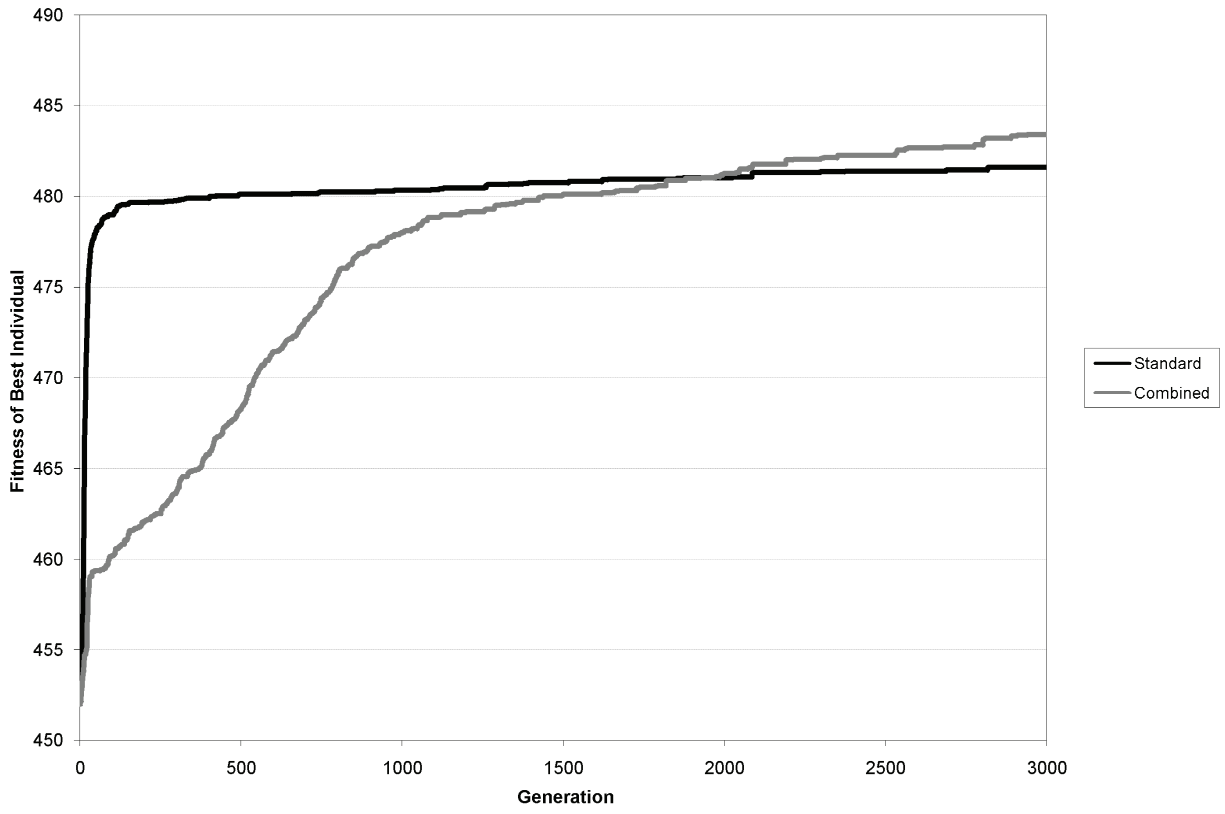


Figure 3-6: Best Individuals at each Generation (Averaged over 100 Runs)

same fitness. This sluggishness is attributed to 1) partitioning the population into sub-populations that contain non-stationary individuals, 2) replacing the bottom age-level every 20 generations, and 3) the beginning population size of ALPS being one-tenth the size of the standard GA. Partitioning the population into sub-populations restricts the rate at which individuals may communicate genetic information to the population as a whole. By implementing sub-populations, only a select few are given access to the genetic information of the overall best individual. In this implementation, elitism allows the overall best individual to rise quickly to the top age-level and remain there to recombine with 99 other individuals. Only through time and surviving many replacements can individuals in the bottom layer combine its genetic information with individuals in the top, slowing down the rate of population convergence.

Secondly, replacing the bottom age-level constantly injects new information into the genetic pool, preventing the genetic operators from merely producing combinations of the same genetic material provided during the initialization phase. Figure 3-7 shows the 95% confidence intervals of the populations for both implementations, averaged over the 100 runs. Whereas the standard GA has a consistent confidence interval, the fitness of the ALPS population oscillates due to the regular replacement of the bottom age-level. The continual production of random individuals causes ALPS to become less deterministic and, thus, have a larger confidence interval.

Thirdly, the population size of each age-level is smaller than the standard GA, which decreases the amount of genetic material available to a single individual at a given time and restricts the fitness growth rate for ALPS. Furthermore, the total population size of ALPS begins at one-tenth

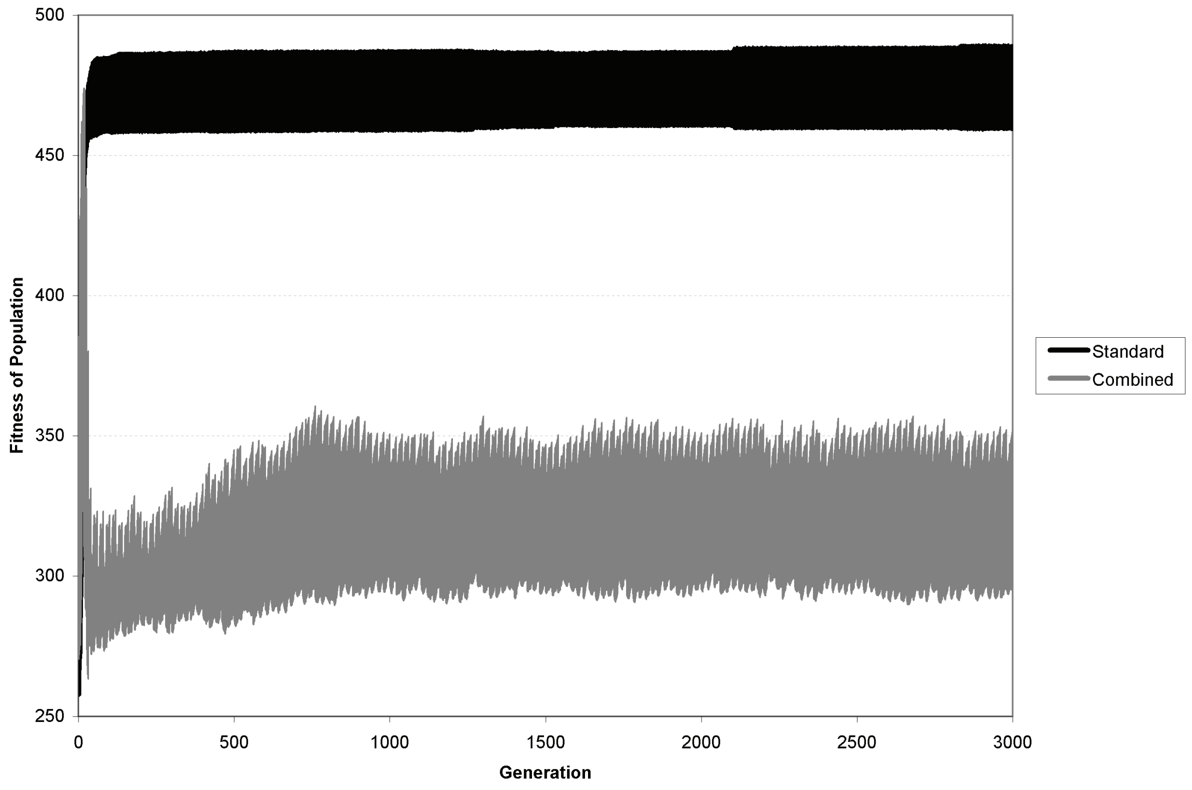


Figure 3-7: Fitness of Population at each Generation (Averaged over 100 Runs)

the size of the standard GA by initializing only the bottom age-level. Figure 3-8 shows the population of ALPS increasing as higher age-levels are initialized due to the age of individuals increasing. Individuals leaving the bottom age-level and replacing individuals at higher age-levels cause the population to decrease, whereas replenishing the bottom age-level with random individuals increases the population—an oscillation seen throughout the evolutionary process. At Generation 700, the top age-level is initialized providing the ALPS population the capacity to compete with the standard GA.

As shown in Figure 3-6, the ALPS fitness value of the best individual increases at an approximate rate of 1 per 400 generations after the population saturates. As previously discussed and shown by Figure 3-5, ALPS converts random individuals into competitive individuals through genet-

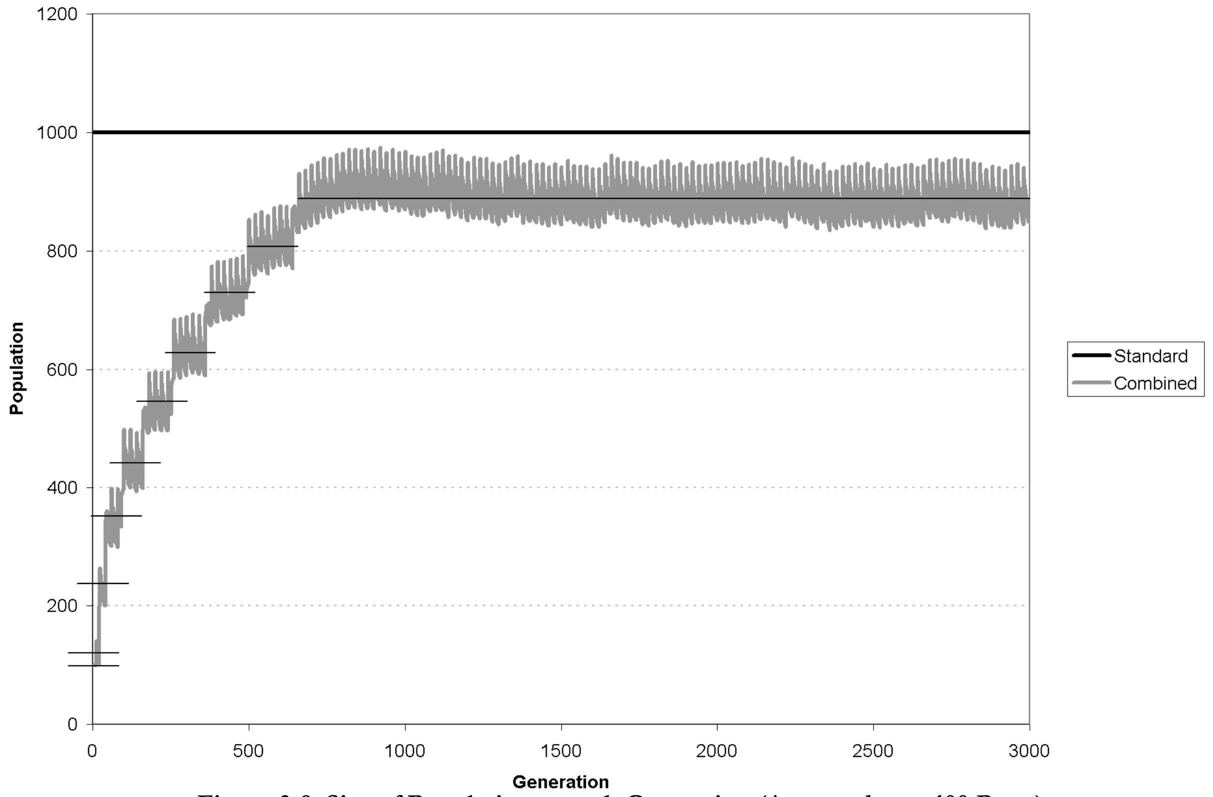


Figure 3-8: Size of Population at each Generation (Averaged over 100 Runs)

ic operators. The top age-level is provided with an elite individual once every 200–300 generations. This continuous supply of new genetic material allows ALPS to surpass the performance of the standard GA during the last third of the runs as shown by Figure 3-6. The best fitness averages and standard deviations of 100 runs at Generation 3000 are 481.62 ± 7.92 for the standard GA and 483.42 ± 9.07 for the ALPS method. Whereas a t-test reveals the difference not to be statistically significant with $P \geq 0.05$, the number of complete repairs produced during these runs reveals the real benefit of ALPS. Of the 100 runs for each implementation, one run of the standard GA produced a complete repair with a 512 fitness value, whereas 3 runs of the ALPS implementation produced a complete repair resulting in a 300% improvement.

3.5.3 Age-Level Management Optimization

As previously discussed in Section 3.3.4, selection of a *2nd parent* for crossover may occur by several methods. The *Combined* method previously compared to the standard GA views two adjacent age-levels as one, calculates proportional fitness for each individual within the combined population, and selects the second parent using the roulette wheel method. After selecting one parent, the *Separate* method, however, selects a 2nd parent to be one of two candidate individuals: one candidate from the current age-level of the first parent and one candidate from the age-level immediately below. The 2nd parent is selected to be one of these two candidates with a probability of 0.25, 0.50, 0.75, or 1.00, where a probability of 1.00 always selects the candidate from the same age-level as the 1st parent.

Four trials each using a different *Separate* probability value were run with 100 runs per trial to determine if the age-level management strategy affects the performance of repairing the faulty 3-bit adder. As shown in Figure 3-9, a probability of 0.75 outperforms the other probabilities as well as

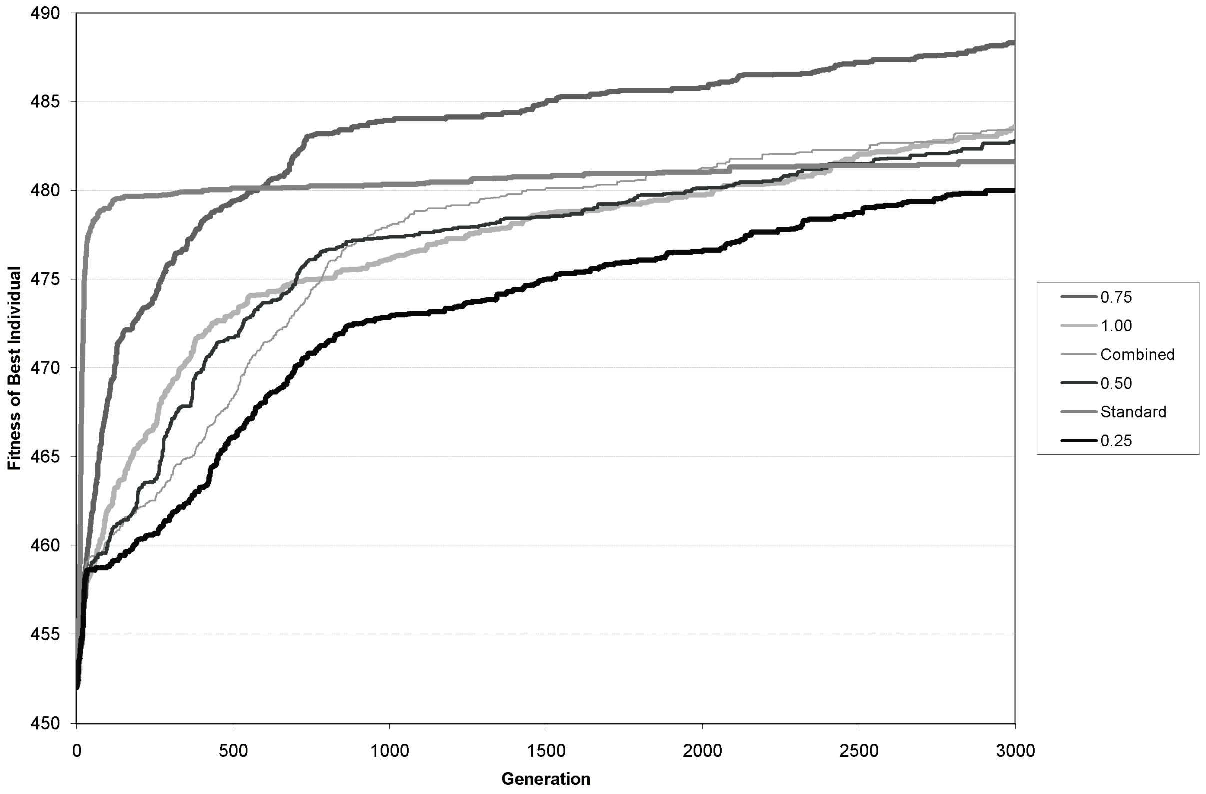


Figure 3-9: Best Individuals at each Generation (Averaged over 100 Runs)

the previous implementations already discussed. It is observed that the 0.75 implementation is not as sluggish as the *Combined* method, whereas the 0.25 probability is more sluggish. The performance difference between 0.75 and the next best implementation of 1.00 is statistically significant with $P < 0.001$. The best fitness averages and standard deviations for 100 runs at Generation 3000 for 0.75, 1.00, 0.50, and 0.25 probabilities respectively are 488.31±9.77, 483.62±10.14, 482.81±8.63, and 479.98±9.24.

For all implementations, Figure 3-10 shows the fitness of the population averaged over 100 runs at each generation. The populations of each of the *Separate* implementations are seen to be more fit than the *Combined* population, implying that keeping the age-levels separate during selection allows fitter individuals to emerge. Figure 3-10 also shows that increasing the probability for indi-

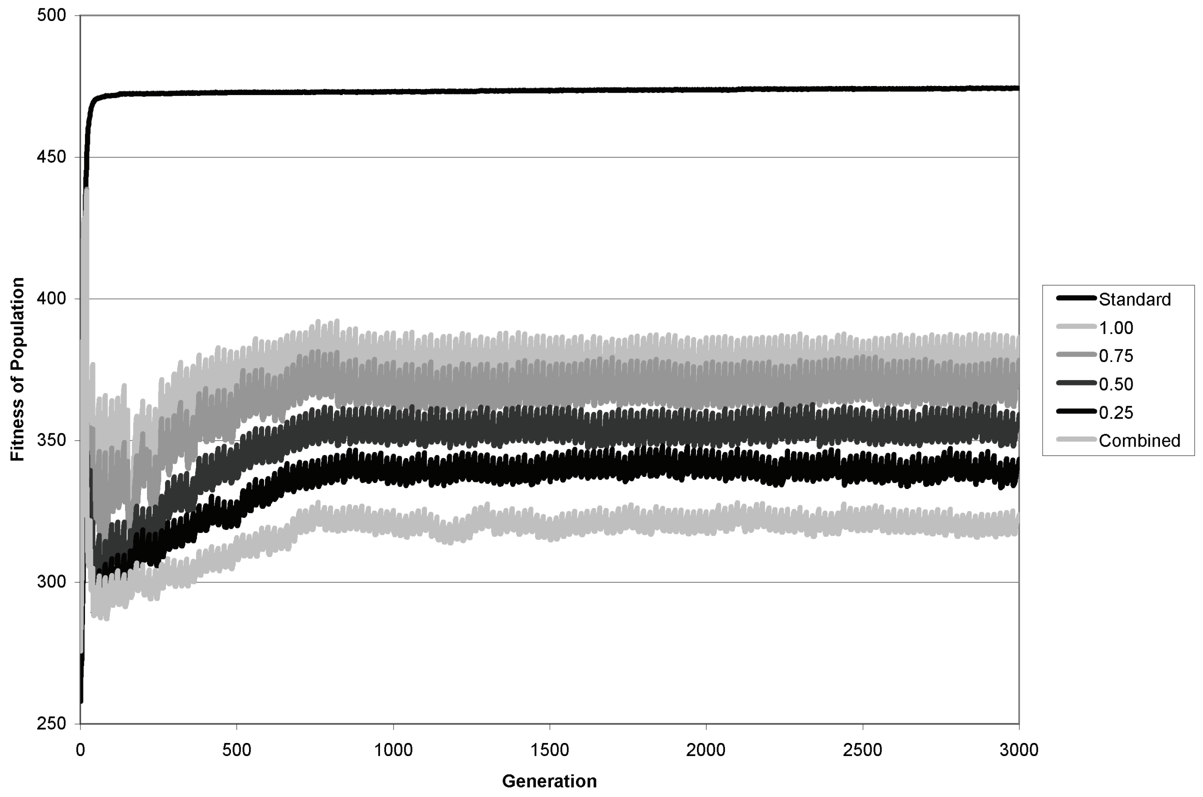


Figure 3-10: Fitness of Population at each Generation (Averaged over 100 Runs)

viduals to breed within its own age-level further increases the fitness of the population, which implies that breeding across age-levels creates less-fit individuals and may be regarded as unproductive breeding. As previously seen with the comparison of the standard GA, having a highly fit population does not necessarily correlate to producing fitter individuals or complete repairs with greater frequency. Figure 3-10 further verifies this, showing that the population using a probability of 0.75 has a lower fitness than a probability of 1.00 yet, on average, produces fitter individuals.

For each of the probabilities implemented above, Figure 3-11 categorizes by fitness the best individuals produced at the end of each run. It also shows the frequency with which each fitness range occurs for a given probability value. Representing 100 runs each, the 1.00 graph shows a somewhat normal distribution centered on its mean of 483.62, whereas the 0.75 probability appears

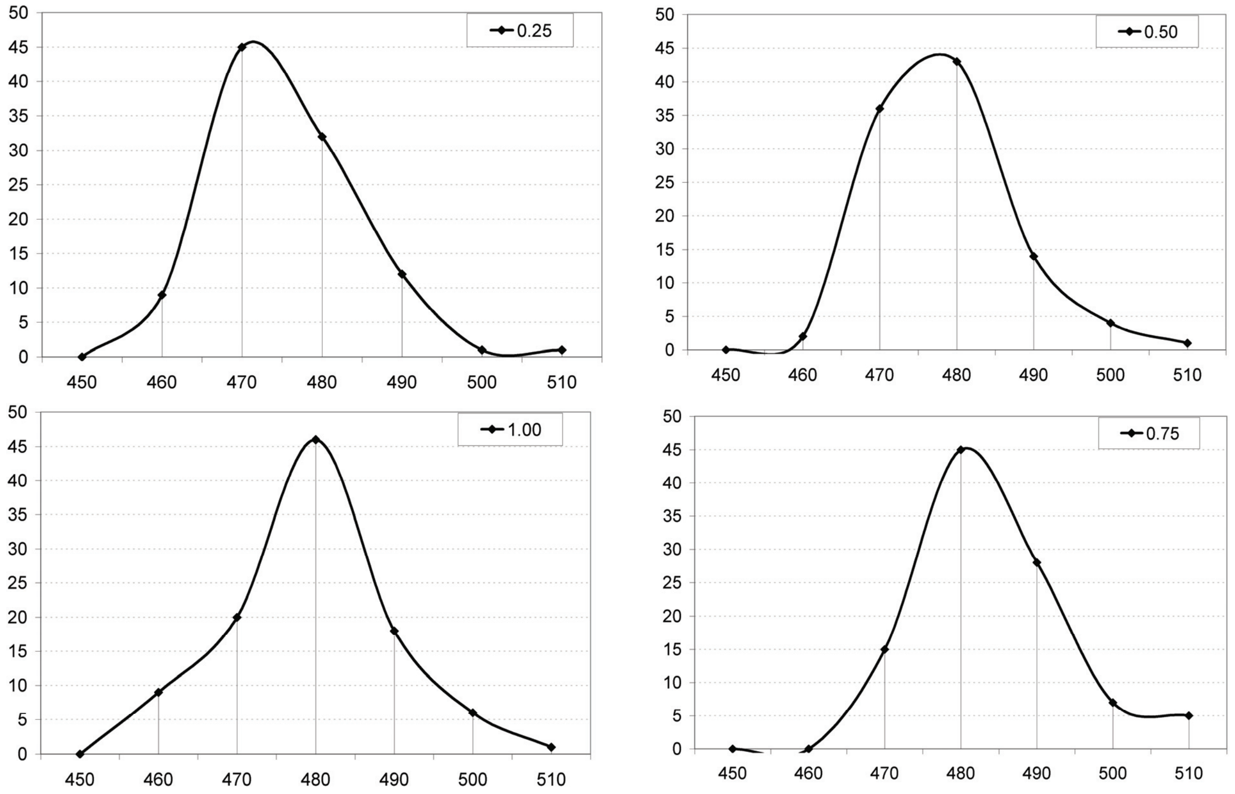


Figure 3-11: Distribution of Best Individuals from 100 Runs

skewed to the right displaying more occurrences of fitter individuals. Coincidently, all individuals within the 510 category for each graph are complete repairs with a fitness of 512. The 0.75 probability had a 5% success rate by producing five complete repairs whereas the other probabilities only produced one complete repair each.

As previously discussed in Section 2.3.1.2, GAs are classified as Dynamic Processes that address many specific fault scenarios during run-time. As such, additional time is required to determine an appropriate solution. Running times for the experiments discussed show that ALPS requires less time due to its variable population size remaining less than the standard GA. Whereas an application utilizing an offline recovery method such as a GA may not require repair solutions to be provided quickly, this decrease in running time can benefit any application.

3.6 Chapter Summary

The work presented herein verifies the viability of utilizing age-levels to prevent population convergence for the fault-repair problem domain. As previously shown, partitioning the population prevents one highly fit individual from quickly dominating the population. For the fault-repair domain, the seeded individual is generally a highly fit individual compared to other randomly generated individuals. Constraining the seeded individual to a sub-population allows the random individuals to evolve into competitive alternative solutions. These developed individuals may have more of a chance of contributing useful genetic material to increase the fitness of the best individual.

Preventing convergence is shown to increase the fitness of the best individual produced along with increasing the probability of evolving a complete repair. To further increase this probability, a modification to the ALPS algorithm is introduced to improve the fitness of the best individual evolved. Results reveal an optimized parameter value for this modification that increases fitness and the probability of completely repairing the faulty circuit.

While the current results of ALPS are promising, better results may be obtained by optimizing the GA parameters specified by Table 3-I. This thesis does not explore the effect that population size, number of age-levels, selection strategy, crossover and mutation rate, elitism, or age gap has on the success of FPGA repair. The parameters that were selected may be the worst-case set, meaning that the repair performance may be increased.

An extension for this work includes simulating additional faults or implementing a more difficult circuit such as the 4-bit multiplier, which Vigander [2001] attempted to repair. To help overcome these difficulties, the GA could utilize other implementations of the FPGA repair process such as the Floating Output fitness evaluation method proposed by Lohn et al. [2003]. Hornby [2006] showed other population management strategies such as deterministic crowding [Mahfoud

1992] to be successful in creating fitter individuals, which may be another option for repairing more difficult digital circuits. Additionally, the complete repairs created by the GA process may be analyzed to develop new selection and replacement strategies.

CHAPTER 4
PARTIAL RECONFIGURATION AND FPGA ARCHITECTURE ANALYSIS

4.1 Introduction

As previously outlined in Section 1.3, *partial reconfiguration* is a process that reconfigures a specific region of FPGA resources without disturbing the remaining resources. By designating a portion of the FPGA as reconfigurable as in Figure 4-1, multiple modules may occupy that space throughout the life of the FPGA. If a system does not need two modules to operate concurrently, partial reconfiguration may time-multiplex between them to decrease spatial resource requirements. Since the size of the partial reconfiguration module determines the reconfiguration speed, smaller modules allow a system to reconfigure more quickly.

Various design flows are available from Xilinx to implement partial reconfiguration designs onto their FPGAs. The three design flows that are currently available include the design flow for ISE 6.3i, *Early Access Partial Reconfiguration (EAPR)* design flow for ISE 8.2i/9.1i [Xilinx 2006], and

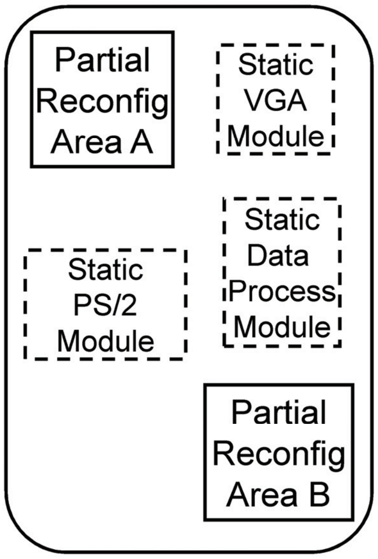


Figure 4-1: Various Modules in an FPGA

the PlanAhead design flow for ISE 9.1i. Obtaining the tools and user guides for the EAPR flow requires registration and authorization from Xilinx. The tools and user guides for the other two flows are available to the public. This paper investigates the specifics of the EAPR design flow and compares its advantages to the module-based design flow for ISE 6.3i.

Section 4.2 explains the design process for partial reconfiguration. Section 4.3 proposes an application for use with partial reconfiguration and details the results obtained from implementing the application. Section 4.4 makes a comparison between the implementations of two devices from the Xilinx Virtex Family. Section 4.5 provides conclusions of the partial reconfiguration process.

4.2 Early Access Partial Reconfiguration Design Flow

Each of the steps of the partial reconfiguration design flow is outlined by Figure 4-2 and described below.

Step 1: Hardware Description Language (HDL) Design and Synthesis

The initial steps in the EAPR design flow are similar to the initial steps in the standard modular design flow. The process begins by designing a top-level design that does not contain any logic.

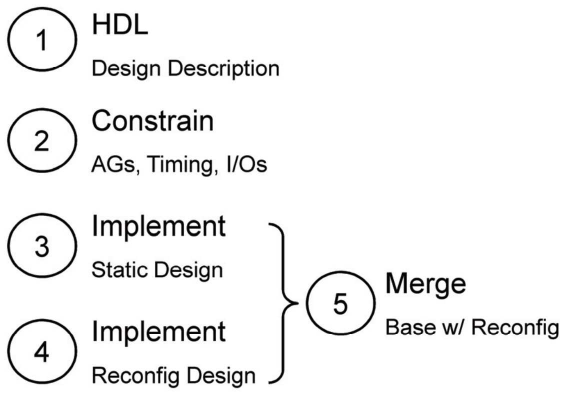


Figure 4-2: Early Access Partial Reconfiguration Flow [Xilinx 2006]

The top-level module only contains I/O instantiations, clock primitives, static module instantiations, partial reconfiguration module instantiations, and signal declarations. In addition, the top-level module must define bus macros. Bus macros are hard macros that facilitate communication between static modules and partial reconfiguration modules.

The static modules contain logic that will remain constant during reconfiguration. The static modules cannot instantiate global clock signals, but may utilize clock signals declared in the top-level module. Similar to the static modules, the partial reconfiguration modules must also not contain global clock signals, but may use those from the top-level module. When designing multiple reconfigurable modules to utilize the same reconfigurable area, the component name and port configuration of each module must match the reconfigurable module instantiation located in the top-level module.

After the HDL design, all of the modules are synthesized with the `Keep Hierarchy` attribute set to `yes`. In addition, the top module is synthesized with the `Add I/O Buffers` attribute enabled, whereas each sub-module—static and each reconfigurable module—is synthesized with this attribute disabled. This provides an `.ngc`, `.ngo`, or `.edf` file format for the implementation step. To ensure that each synthesized module does not interfere with another, creating a design project for each module, each project within its own directory, is recommended.

Step 2: Set Design Constraints

Before implementing the synthesized design, constraints must be specified for the top-level design. Mandatory constraints include the AREA_GROUP, RANGE, MODE, and LOC constraints. The AREA_GROUP constraint specifies which modules in the top-level module are static and which are reconfigurable. Each module instantiated by the top-level module is assigned to a group. The RANGE constraint is only applied to the reconfigurable group to specify its range of resources, which may be any-sized rectangle. All resources within the designated area must be covered by a RANGE constraint, including SLICE, RAMB16, MULT18X18, TBUF (Tri-state Buffer), FIFO16 (First In, First Out), and DSP48 (Digital Signal Processor) resources. The static group is allowed to use all other resources not specified by the reconfigurable group. The MODE constraint is also only applied to the reconfigurable group, which specifies that the group is reconfigurable.

Every pin, clocking primitive, and bus macro in the top-level design must contain a LOC constraint. Bus macros are located so that they straddle the reconfigurable boundary as set by the RANGE constraint. Constraining the location of the bus macros enforce their position during all iterations of the implementation step. This one user constraint file for the top-level directory is used to implement the static module and each of the reconfigurable modules.

Step 3: Static Module Implementation

Before the static modules are implemented, the top-level is translated to ensure that the constraints file has been properly created. After the top-level translation is successful, the static module implementation begins within the context of the top-level module and constraints. The implementation process translates the synthesized top-level module while using all synthesized static modules created in Step 1 and the top-level constraints file created in Step 2. The synthesized bus macros, which are provided by Xilinx, must also be included in the directory where translation is to take place. After a successful translation, MAP creates the mapped design.

PAR is invoked to provide the `.ncd` output file that is used to generate a bitstream for programming the FPGA. In addition to the `.ncd` file, PAR provides a `static.used` file that lists the routing resources within the reconfigurable area used by the static implementation. This file is used by the reconfigurable implementation to prevent using the same routing resources.

Step 4: Reconfigurable Module Implementation

The reconfigurable modules are implemented using the top-level module, reconfigurable modules, and synthesized bus macro design files. NGDBuild, MAP, and PAR are run similarly to the Step 3, but the `static.used` file created in Step 3 is renamed to `arcs.exclude` and included in the reconfigurable implementation directory. PAR automatically uses the `arcs.exclude` file to prevent the reconfigurable modules from using routing resources allocated for the static modules. The reconfigurable module implementation process is performed once for each reconfiguration module occupying the same reconfigurable area, where each implementation process contains either one module or the other.

Step 5: Merge Implementations

To generate the appropriate full and partial bitstreams for programming the FPGA, the `pr_verifydesign` and `pr_assemble` routines are run using the `.ncd` files output by the PAR processes in Step 3 and Step 4. The merge process is run once to merge the static design with the first reconfigurable module. The merge process is run a second time to merge the static design with the second reconfigurable module. The merge process outputs an FPGA `.bit` file used to initialize the FPGA with the static design and a partial bit file for each reconfigurable module that is used to partially reconfigure the FPGA.

4.3 Application Using Partial Reconfiguration

4.3.1 Case-study Application

The application investigated uses partial reconfiguration to switch between two hash algorithms: MD5 [Rivest 1992] and SHA-1 [NIST 1995]. A hypothetical system is considered where both algorithms are required for use and space is limited, but does not require simultaneous use of the two algorithms. Since space, and not time, is at a premium, partial reconfiguration is a viable option.

Both MD5 and SHA-1 operate by accepting a message and padding the message until its bit length is a multiple of 512. Afterwards, the algorithm divides each 512-bit section into sixteen 32-bit segments. As shown in Figure 4-3, these segments (M_i) are inputs to a process that produces a message digest. The process includes a step box, F(b,c,d), that uses three 32-bit registers for inputs and

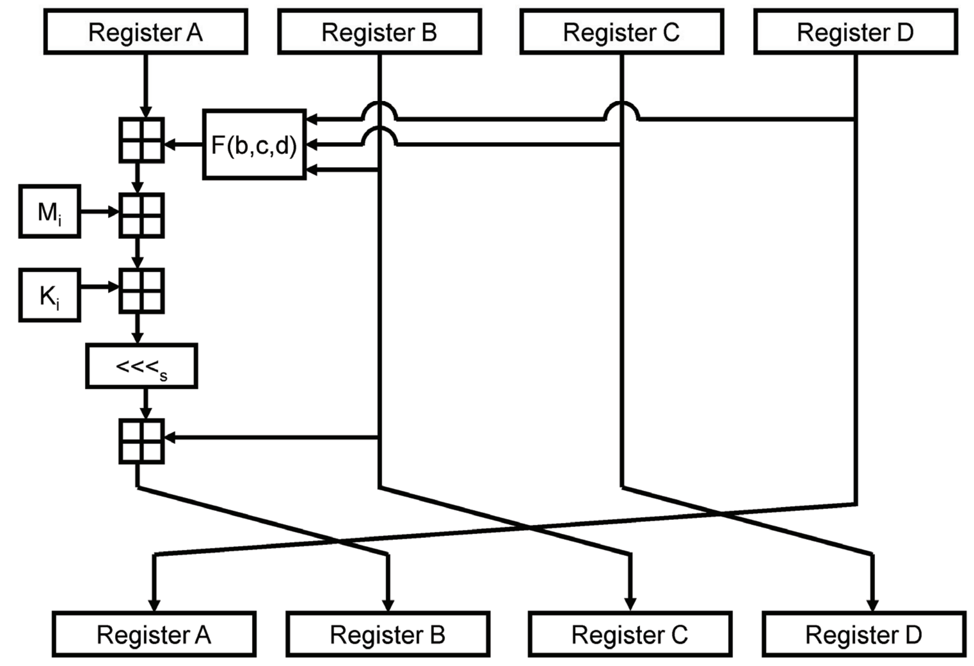


Figure 4-3: MD5 Hash Algorithm Overview [Wikipedia Current]

rotates between four functions to determine its output. The algorithm additionally utilizes shifting and addition modulo 2^{32} operations. This process repeats until the algorithm has used all of the message segments as inputs. MD5 provides a final message digest of 128 bits whereas SHA-1 provides a digest of 160 bits.

Since these algorithms are similar in structure and method, research has optimized the two algorithms for systems that require both algorithms. One method presented in [Järvinen et al. 2005] optimizes the two algorithms by combining the hardware used by the algorithms. Through resource sharing, this approach saves space by not implementing redundant hardware. Another method in [Tan and DeMara 2007] rotates the four functions used by the step-box through partial reconfiguration. Both of these methods are only applicable in systems that do not require both algorithms to operate concurrently.

The proposed approach moves from the fine granularity in [Tan and DeMara 2007] to a courser granularity by considering the entire hash algorithm. Partial reconfiguration is utilized by allocating a portion of the FPGA as reconfigurable to contain either the MD5 module or the SHA-1 module at any given time. The remaining portion of the FPGA contains a keyboard module to accept a user input used as the message and a Video Graphics Array (VGA) module to display the user input along with the output of the hash algorithm.

4.3.2 Overview of Design using Partial Reconfiguration

A top-level view of the hash algorithm system is shown in Figure 4-4. The system allows a user to input a 32-bit hexadecimal message and receive the corresponding message digest. Depending on which hash algorithm is loaded in the reconfigurable area, the user will receive the appropriate message digest. A Personal System 2 (PS/2) keyboard provides the input while a VGA monitor

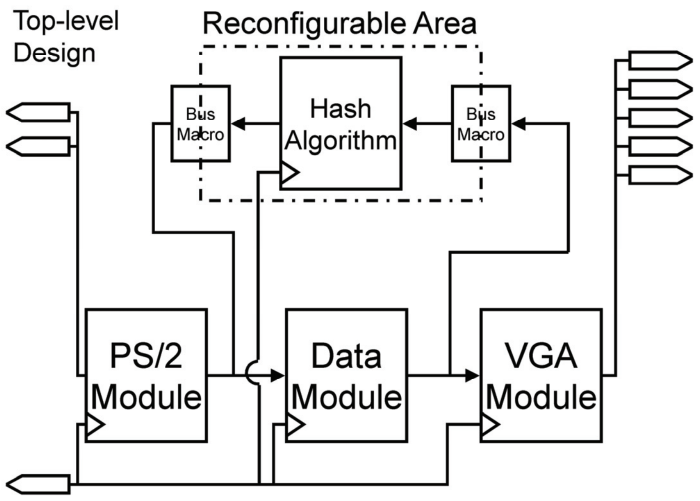


Figure 4-4: Top-level View of Partial Reconfiguration Design

provides the output. The static design includes a VGA module, keyboard module, and modules necessary for processing the inputs of the keyboard and the outputs for the VGA monitor. The data module accepts a user input of eight hexadecimal values as the message and displays these values on the VGA monitor.

The reconfigurable design includes one hash algorithm, either MD5 or SHA-1. The focus of the application is to demonstrate the two different hash algorithms, especially the difference in message digest length. The MD5 provides a 128-bit message digest whereas the SHA-1 algorithm provides a 160-bit message digest. To accommodate the two algorithms, the MD5 algorithm pads its message digest with 32bits of zeros before sending it to the VGA module. Keeping the total message digest length constant between the two algorithms allows the VGA processing module to remain static for both hash algorithms. It is possible, however, that additional data be passed to the VGA processing module to inform it which algorithm is present to more appropriately display the message digest.

The total number of bits being passed at one time between static and reconfigurable logic is 192—32 bits for the user input and 160 bits for the message digest. Since all inputs and outputs between static and reconfigurable logic are required to pass through bus macros, twenty-four bus macros, 8 bits wide each, are required to pass the 32-bit user input and 160-bit hash output. Whereas the FPGA can physically accommodate 24 bus macros, a more efficient implementation facilitated a design of a parallel-to-serial encoder/decoder module for transferring the data.

Before passing the user input to the hash algorithm, the data is sent serially from the static logic to the reconfigurable logic. At the same time, the reconfigurable logic receives each bit and converts it into an array for use within the algorithm. Likewise, the reconfigurable logic sends the 160-bit message digest bit-by-bit to the static logic to display on the VGA monitor. After storing user inputs and message digests into registers, the values are written to RAM locations for readback by the VGA module.

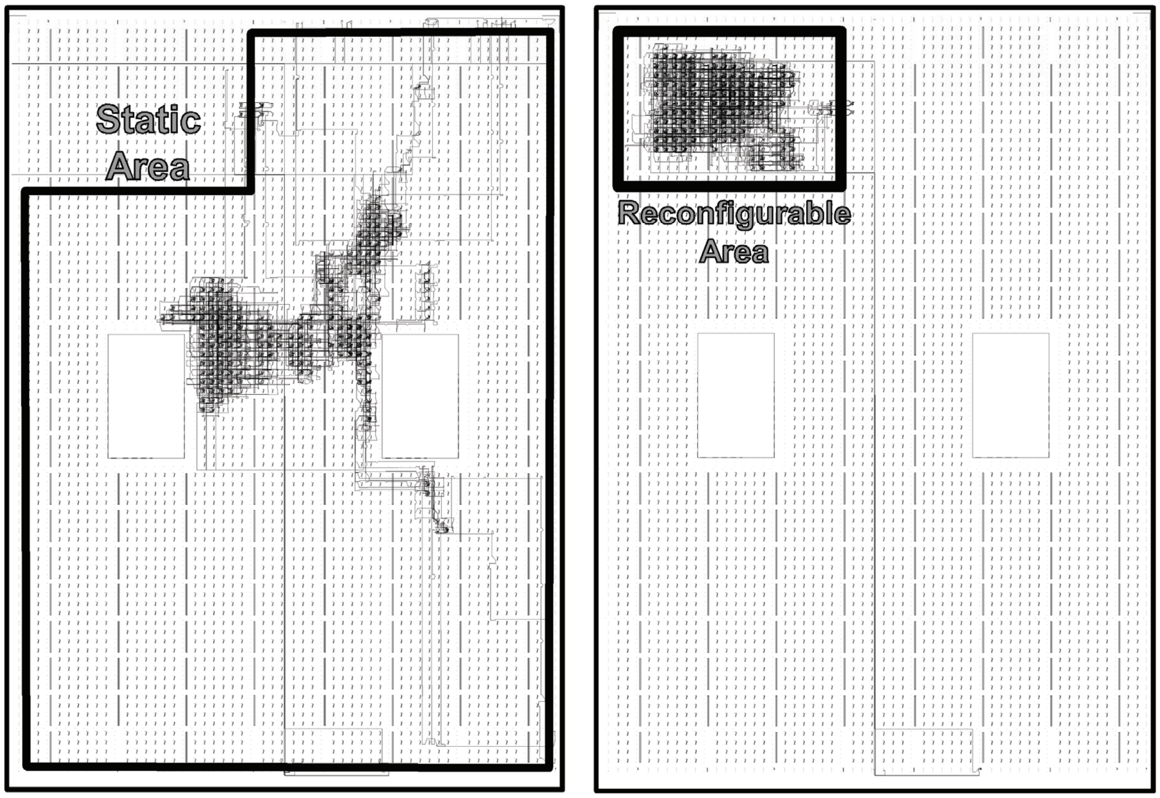
With this implementation strategy, only two bus macros are required to pass data from static logic to the reconfigurable logic, one for right-to-left transmissions and the other for left-to-right transmissions. Only six of the sixteen bus macro channels are used, so ten additional parallel-to-serial encoder/decoder modules may be implemented to decrease the time for data transfer. To determine whether the additional channels are necessary, data transmission and digest calculation latencies are calculated.

Digest calculation time is calculated to be 3.2μs with an additional data transmission time of 1.92μs. The Data Processing module updates the RAM with new user inputs and message digest values during the 64μs V-sync pulse, which occurs at a frequency of 28.8 KHz. As expected, tests indicate that the data latency between user input, sending the user input, calculating the message digest, sending the message digest, and displaying the message digest is negligible.

4.3.3 FPGA Implementation

Partial reconfiguration was successfully implemented on a XC2VP30-FF896 Virtex-II Pro FPGA. As seen in Figure 4-5a and Figure 4-5b, the reconfigurable area is located in the upper-left portion of the FPGA to allow the static logic access to the VGA Input/Output Blocks (IOB) in the upper right and PS/2 IOBs located in the lower right portions of the FPGA.

Figure 4-5a demonstrates the latest feature of the EAPR design flow by allowing a two-dimensional reconfigurable area. Since the Virtex-II Pro configuration frame extends the entire height of the device, glitchless reconfiguration prevents interruption of the static logic found underneath the reconfigurable area when implementing the hash algorithm [Lysaght et al. 2006]. To dem-



a) Static Modules **b) SHA-1 Module**

Figure 4-5: FPGA Implementation and Resource Utilization

onstrate further that the static logic is not interrupted, the constant connection between the FPGA and the VGA monitor is visually verified while partially reconfiguring the FPGA device.

4.4 Virtex Family Comparison

The Virtex-II, Virtex-II Pro, and Virtex-4 FPGAs all support partial reconfiguration. Only the most recent design flow has allowed two-dimensional reconfiguration areas for the Virtex II and Virtex II Pro. Before the EAPR design flow, the reconfiguration areas on these devices were required to extend the entire height of the device to match the height of the configuration frames. EAPR removed this restriction due to an inherent capability of the Virtex-II and Virtex-II Pro, "glitchless reconfiguration." *Glitchless reconfiguration* enables routing and logic resources to remain operational if the configuration setting for that resource is the same before and after the reconfiguration. This capability is applied to partial reconfiguration when a reconfiguration area is smaller than the height of the device. Whereas the Virtex-II/ Pro still reconfigures the entire height of the device, glitchless reconfiguration maintains any static routing and logic resources residing below or above the reconfiguration area. Since a configuration frame extends the entire height of the device, reconfigurable areas cannot overlap vertically, lest they change one another's configurations upon partial reconfiguration.

Virtex-4 FPGAs differ from the Virtex-II and Virtex-II Pro in that the height of its configuration frame is only 16 CLBs. Because of this smaller configuration granularity, reconfigurable areas are allowed to overlap vertically, so long as they do not share the same configuration frame. In the case of the XC4VFX60 Virtex-4 FPGA, which is 128 CLBs in height, up to eight reconfigurable areas may reside in any one CLB column.

Bitstream Size Comparison

Before the availability of partial reconfiguration, users were required to program the entire FPGA to switch between applications residing on a mere portion of the FPGA. One downside to this method is the reconfiguration time—larger bitstream files increase the reconfiguration time. Additionally, the portion of the FPGA that will not be changed is interrupted until the reconfiguration process is complete.

Partial reconfiguration addresses the bitstream size by reducing the filesize of the partial bitstream representing the reconfigurable module. As listed in Table 4-I, the MD5 partial bitstream generated by the EAPR flow is 22.1% of an initial device bitstream. Based on the filesize alone, one could predict that the configuration time is decreased by 77.9% of a full-device configuration. Experiments validated this prediction with a reconfiguration time decrease of 71.4%. In addition, the partial reconfiguration was demonstrated not to interrupt the operation of the static logic, including the keyboard and VGA modules.

Table 4-I also lists the area allocated for the MD5 and SHA-1 reconfigurable modules. As previously discussed, the Virtex-II Pro configuration frame extends the entire height of the device. Therefore, even if a small section of the FPGA is to be reconfigured, such as the 2.8% area used by

Table 4-I: Virtex-II Pro Bitstream and Area Sizes

	xc2vp30-7ff896, 80CLB configuration frame			
	Bitstream Filesize (bytes)	Area Allocated (slices)	Area Used (slices)	Time to Configure (seconds)
Full Device	1,448,817	13,696	13,696	7
MD5	320,597 (22.1%)	1280 (9.3%)	389 (2.8%)	2 (28.6%)
SHA-1	356,702 (24.6%)	1280 (9.3%)	457 (3.3%)	2 (28.6%)

the MD5 module, the full height of the device is included in the configuration bitstream. This explains the discrepancy between the 28.6% bitstream filesize and 2.8% area actually used by the reconfigurable module. As expected, the partial bitstreams of the Virtex-II Pro FPGAs are much larger than required.

For comparison, the same application was implemented on a Virtex-4 FPGA. While the selected Virtex-4 device was not available to configure, bitstreams were generated nonetheless. The Virtex-II Pro user constraints file was modified to accommodate the change in device, including LOC constraints and resource RANGE constraints. Additionally, the Virtex-II Pro bus macros were replaced with Virtex-4 bus macros. The EAPR design flow was then followed as outlined by Figure 4-2.

As previously discussed, since the height of the configuration frame in Virtex-4 devices are shorter, the partial bitstream filesize should more closely represent the number of resources actually being reconfigured. Table 4-II verifies this prediction by listing the MD5 partial bitstream as 3.7% of a full bitstream more closely representing the 2.8% slice utilization. Since the Virtex-4 has nearly twice the available resources of the Virtex-II Pro, the possible savings in configuration time is even more dramatic at 96.3%.

The Virtex-II Pro bitstream filesizes can also be compared to the Virtex-4 bitstream filesizes.

Table 4-II: Virtex-4 Bitstream and Area Sizes

	xc4vfx60-11ff672, 16CLB configuration frame		
	Bitstream Filesize (bytes)	Area Allocated (slices)	Area Used (slices)
Full Device	2,625,438	25,280	25,280
MD5	95,962 (3.7%)	1,280 (5.1%)	405 (1.6%)
SHA-1	97,619 (3.7%)	1,280 (5.1%)	472 (1.9%)

When comparing the filesizes of the partial bitstreams for the MD5 algorithm, it is obvious that the Virtex-4 has a more efficient bitstream. As proven by Table 4-I, a smaller bitstream decreases the reconfiguration time. In comparing the two MD5 bitstream filesizes, the Virtex-4 can conceivably reconfigure the approximate same number of resources 30% quicker than the Virtex-II Pro. Actual reconfigurations with a Virtex-4 need to be performed to verify these predictions.

4.5 Chapter Summary

A simple application of switching between two hash algorithms was demonstrated to successfully exhibit the benefits of partial reconfiguration. For systems that require more modules than spatial resources allow, partial reconfiguration is a viable option. Partial reconfiguration reduces the bitstream filesize when compared to a full reconfiguration. Additionally, partial reconfiguration allows applications to switch without interrupting the static portion of the FPGA.

The Virtex-4 was proven more efficient than the Virtex-II Pro for partial reconfiguration. By decreasing the size of the bitstream, the Virtex-4 requires less time to reconfigure the same approximate number of resources. This savings in time may be particularly useful for systems that depend upon the configuration time such as a repetitive intrinsic evolution process utilizing GAs. Additional work includes verifying the predictions in the savings of configuration time in the Virtex-4 device.

CHAPTER 5
DYNAMIC PROCESSOR ALLOCATION STRATEGIES

In the previous chapter, partial reconfiguration was shown to time-multiplex between two different modules utilizing the same hardware resources. Another application that exploits the reduced reconfiguration granularity and the time multiplexing of partial reconfiguration is a scalable architecture for video processing. Within a scalable architecture, generic processing elements (PE) are implemented for use between multiple video processing functions such as Discrete Cosine Transform (DCT) [Lee et al. 2006a] and motion estimation [Lee et al. 2006b]. If a user requires varying degrees of video quality, partial reconfiguration can be used to add and remove PEs while the overall process continues to run. Huang et al. [2008] implement such an architecture as a two-level scalable architecture using partial reconfiguration. On one level, the architecture modifies the number of Processing Elements (PE) allocated to the DCT video processing function. On the second level, the architecture modifies the precision with which the DCT function is processed.

5.1 Video Compression Overview

Video compression is used by applications to reduce the size of the information for either transmission or storage. Lossless video compression techniques reduce the size in such a way that the original content may be reconstructed perfectly from the compressed information. Lossy video compression techniques, on the other hand, cannot reconstruct the original video perfectly; during the compression, some information is discarded or lost. The goal of lossy techniques is to maximize the compression ratio while minimizing perceptible or objectionable differences between the compressed and original videos. Since videos are merely a sequence of still images referred to as frames, video compression may be understood generally by exploring common image compression techniques.

One such common image compression technique includes the specification by the Joint Photographic Experts Group (JPEG). The first step in JPEG compression is to partition the image into 8x8pixel blocks, as seen in Figure 5-1. Each of these blocks then is *transformed* into a frequency domain representation by an 8x8 DCT. Frequency information is useful in the compression process since lower frequencies correspond to highly perceptible features in the image whereas the higher frequencies are less perceptible. DCT produces 64 frequency-domain coefficients from the 64 spatial pixels of each 8x8 block. If lower quality is acceptable, some of the higher frequency coefficients may be discarded to produce blocks of various sizes, such as 7x7 or 1x1. Following DCT, the *quantization* process favors the lower frequency coefficients by encoding them with a higher degree of precision than the higher frequency coefficients. After quantization, most of the higher frequency

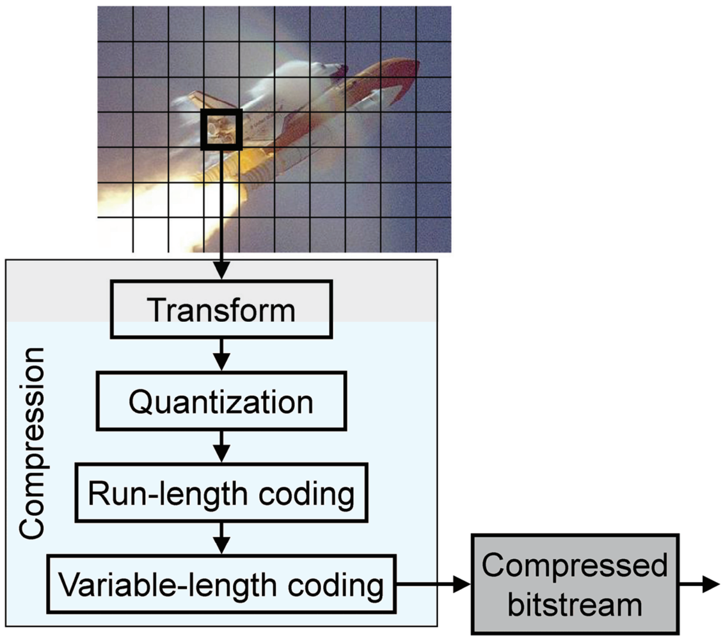


Figure 5-1: Generic Image Compression Method

DCT coefficients become zero. A *run-length coding* method capitalizes on this trend by grouping a run of zero-valued coefficients together, compressing the image bitstream. *Variable-length coding* usually follows, converting commonly occurring symbols representing quantized DCT coefficients or runs of zero-valued coefficients with shorter length code words. On average, variable-length coding further reduces the image bitstream size.

5.2 Scalable Architecture for DCT and Motion Estimation

The scalable architecture is designed for applications that vary in required quality and, thus, dynamically control the parameters of video processing functions. For example, a user may not require full quality of a video signal. As such, the proposed architecture may compensate by implementing PEs that provide partial precision of its calculations of the DCT computation to reduce dynamic power consumption. Similarly, when a user requires higher quality video, full precision PEs may be implemented. Additionally, the compression ratio may be a parameter to the video processing function that varies with time. To compensate for higher compression ratios, the scalable architecture may reduce the number of PEs allocated to the DCT function. A direct benefit of removing PEs from operation is, again, a reduction in dynamic power consumption.

One alternative to partial reconfiguration is to implement some control logic to facilitate the precision of the PE and enable or disable the PEs from operation. Such an implementation, while it may equally reduce dynamic power consumption, requires a greater number of FPGA resources. A tradeoff is apparent between the time to reconfigure the FPGA with partial bitstreams and the space to implement logic to switch between precisions and number of PEs. An additional benefit of partial reconfiguration that switching logic cannot provide is the ability to reallocate its resources to other functions. In the case of DCT and Motion Estimation (ME), the PE for each function is sufficiently different that simple logic cannot switch between the two types of PEs without incurring significant resource overheads. The scalable architecture exploits the power of partial reconfiguration by reconfiguring unused DCT PEs to other types of PEs for video processing functions that may benefit from the extra processing power. As an example, motion estimation is considered as another video processing function that may benefit from unused DCT processing elements.

Partial reconfiguration also allows dynamic resource management to increase the capability of an FPGA device. Whereas a smaller FPGA device is limited in logic resources, it may use partial reconfiguration to implement a time-multiplexed pipeline of video processing functions. Whereas more time is required, a smaller FPGA becomes capable of producing video equal in quality to a much larger FPGA device. This time multiplexing between functions on a small scale demonstrates the capability that partial reconfiguration enables for FPGAs.

5.3 Scalable Architecture Implementation

Using the Xilinx Early Access Partial Reconfiguration (EAPR) design flow [Xilinx 2006], the scalable architecture is implemented on the Xilinx Virtex-4 SX35 Video Starter Kit. The scalable architecture previously described naturally allows each PE to reside within a separate reconfiguration area for modification of its configuration without disturbing the remaining portion of the FPGA—meaning that the system clocks do not stop and the rest of the FPGA can continue to function. Figure 5-2 shows an implementation of the scalable architecture with the locations of the eight reconfiguration areas.

Partial reconfiguration allows flexibility in selecting the quality of precision of a specific PE along with the total number of PEs allocated to the DCT application. Each reconfigurable region is able to implement one PE. In 8x8 2D-DCT computations, for example, each reconfigurable area is

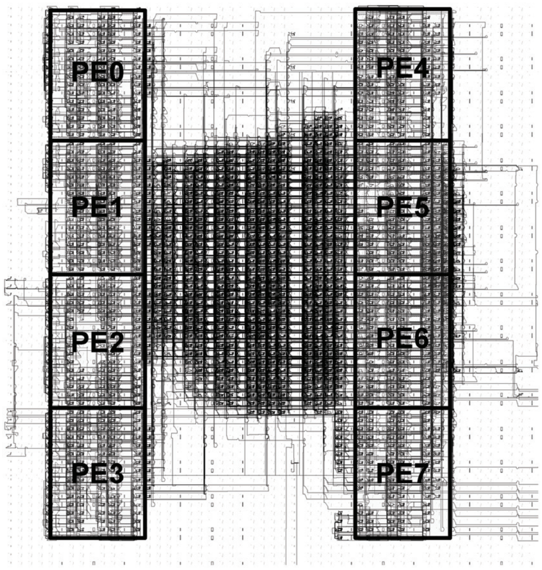


Figure 5-2: Location of 8 PEs on a V4SX35 device [Huang et al. 2008]

configured to contain one PE each, totaling 8 PEs. In 1x1 computations, one reconfigurable area contains one PE while the other 7 reconfigurable areas are made available to other video functions such as motion estimation. In the scalable architecture, three types of PEs are designed: a full precision DCT PE, a partial precision DCT PE, and an Empty PE. The Empty PE allows those reconfiguration areas not being used by video processing functions to contain no switching logic to reduce dynamic power consumption.

Since the Full Precision PE is the largest of the three configurations, its resource requirements determine the boundaries of the reconfiguration areas. The Virtex-4 architecture has a configuration frame resolution of 16 CLBs in height—reduced from the Virtex-II architecture whose configuration frame resolution includes the entire height of the device [Lysaght et al. 2006]. Therefore, the reconfiguration areas span the minimum of 32 slices in height, whereas the width of each reconfiguration area is minimized to encompass its specific PE design, making each of the *Slices within Area* values listed in Table 5-I a multiple of 32.

A partial bitstream is generated for each reconfiguration area and for each type of PE. For example, 24 partial bitstreams are generated in the implementation of 8 reconfiguration areas and 3 types of PEs. Table 5-I lists the area size, resource utilization, and bitstream filesizes for each of the Full Precision PE partial bitstreams generated. Since the Full Precision PE has the largest resource utilization, larger than the Partial Precision or Empty PEs, its bitstream sizes are the upper bounds for all types of PEs. For comparison, a bitstream filesize of an Empty PE is 10,586 bytes.

Before partial bitstreams are used, the FPGA is initialized first with a full bitstream. In designing the initial full bitstream, the user determines the most useful combination of type and number of PEs to be the initial configuration of the FPGA—full or partial precision and the type of DCT, 1x1, 2x2, etc. The size of the initial bitstream is always 1,712,614 bytes, regardless of whether all 8 Full Precision PEs are implemented or only 1 Full Precision PE with 7 Empty PEs are implemented. In comparison to a full bitstream, partial bitstream filesizes are significantly smaller and reducing the storage space required for the various bitstreams. The results show that the filesize of a Full Precision PE bitstream is about 1.6% of a full bitstream. As demonstrated in CHAPTER 4, this decrease in bitstream filesize proportionally decreases the reconfiguration time.

Table 5-I: Full Precision PE Implementation Results

	Slices within Area (Slice Utilization)	Bitstream Filesize in bytes
PE0	320 (94.38%)	22,306
PE1	384 (95.05%)	27,794
PE2	384 (84.38%)	28,306
PE3	384 (92.97%)	28,158
PE4	320 (91.25%)	22,306
PE5	384 (88.54%)	27,354
PE6	384 (87.76%)	27,618
PE7	384 (95.57%)	27,654

Table 5-II lists a comparison between one non-partial reconfiguration scenario and two partial reconfiguration scenarios. In the case of non-partial reconfiguration, a full bitstream needs to be generated and stored for each 2D-DCT configuration. For example, a full bitstream of 1,712,614 bytes is required for a 1x1 Full Precision DCT configuration. To implement an 8x8 Full Precision DCT function, another full bitstream is required. To implement a 4x4 Full Precision DCT function with 4 Motion Estimation PEs, a third full bitstream is required. For three distinct hardware arrangements, 4.90 MB of storage space is required. To switch between each of these hardware arrangements, the entire FPGA is reconfigured, stopping all video processing elements. The shortest configuration time needed to switch between hardware arrangements is equal to the worst time at 17 ms. The configuration time is estimated based on the timing of SelectMAP interface using conti-

Table 5-II: Size and Configuration Times of Bitstreams [Huang et al. 2008]

		Bitstream Filesize	Configuration Time
Non-PR	1x1 Full 2D-DCT	1,712,614 bytes	17 ms
	4x4 DCT & 4 ME PEs	1,712,614 bytes	17 ms
	8x8 Full 2D-DCT	1,712,614 bytes	17 ms
	3 H/W Arrangements	4.90 MB	17 ms/17 ms (Best/Worst)
PR	Initial (8x8)	1,712,614 bytes	17 ms
	8 Full Precision PEs	8 × 28,306 bytes	8 × 0.283 ms
	8 Partial Precision PEs	8 × 28,306 bytes	8 × 0.283 ms
	8 Empty PEs	8 × 10,586 bytes	8 × 0.106 ms
	16 H/W Arrangements	2.15 MB	0.106/2.265 ms (Best/Worst)
PR	Initial (8x8)	1,712,614 bytes	17 ms
	8 Full Precision PEs	8 × 28,306 bytes	8 × 0.283 ms
	8 Partial Precision PEs	8 × 28,306 bytes	8 × 0.283 ms
	8 Empty PEs	8 × 10,586 bytes	8 × 0.106 ms
	8 Motion Estimation PEs	8 × 28,306 bytes	8 × 0.283 ms
	80 H/W Arrangements	2.36 MB	0.106/2.265 ms (Best/Worst)

nuous data loading as previously shown in Section 2.4.1.4:

$$T_{config} = (bytes + 3) \cdot \frac{1}{f_{cclk}}, \qquad \textbf{(5-1)}$$

Here, $bytes$ is the number of bytes of the bitstream stored in the external PROM and f_{cclk} is the clock frequency of the SelectMAP configuration clock set to 100 MHz in the estimations in Table 5-II.

In an implementation of the scalable architecture using partial reconfiguration, a user stores one full-device bitstream and all partial bitstreams on an external ROM. In calculating the storage requirements, the worst-case Full Precision PE partial bitstream filesize— 28,306 bytes— is used for partial bitstream totals. The total space required for implementing the initial bitstream and all three types of 2D-DCT PEs—Full, Partial, and Empty—is approximately 2.15 MB. In comparison to the non-pr implementation shown in Table 5-II, partial reconfiguration results in a 2.3-fold decrease in storage whereas the number of distinct hardware arrangements possible is increased 5.3-fold. Additionally, switching between these hardware arrangements does not disturb logic residing outside of the reconfiguration areas. The shortest configuration time to switch between arrangements is 0.106 ms by implementing one Empty PE, for example, to switch from 8x8 DCT to 7x7 DCT. The longest configuration time is estimated to be 2.265 ms to switch, for example, from 8x8 Partial Precision to 8x8 Full Precision, which is much less than the 17 ms required by a full bitstream.

5.4 Scalable Architecture Hardware Arrangements

As seen in Table 5-II, the addition of eight motion estimation PE bitstreams only increases the storage requirement by 0.21 MB. For a 1.1-fold increase in storage overhead, the capability of the FPGA is increased 5-fold, expanding the number of possible hardware arrangements from 16 to

80. Table 5-III lists all 80 hardware arrangements possible with the Full Precision, Partial Precision, Empty, and Motion Estimation PEs.

As indicated by the first row of Table 5-III, the Full Precision DCT function can vary from 1x1 to 8x8, where Empty PEs fill up the unutilized PE locations. Since the number of PEs are varied with the DCT function, this results in 8 unique hardware arrangements. When the Motion Estimation PE is used in conjunction with the Full Precision, and the Empty PEs fill the unutilized PE locations, Row 2 shows that one Motion Estimation PE may be implemented while a 7x7 Full Precision DCT function is implemented, resulting in one hardware arrangement. Row 3 shows that a 6x6 Full Precision DCT function leaves available two PE locations where Motion Estimation may use both or Motion Estimation may use one and an Empty PE may occupy the other, resulting in two

Table 5-III: Partial Reconfiguration Hardware Arrangements for 8 PE Locations

		Variable	Unique H/W Arrangements
Full Precision filled w/ Empty PEs		DCT [1x1–8x8]	8
Full Precision and Motion Estimation (ME) PEs filled w/ Empty PEs	7x7 DCT	ME PE [1]	1
	6x6 DCT	ME PEs [1–2]	2
	5x5 DCT	ME PEs [1–3]	3
	4x4 DCT	ME PEs [1–4]	4
	3x3 DCT	ME PEs [1–5]	5
	2x2 DCT	ME PEs [1–6]	6
	1x1 DCT	ME PEs [1–7]	7
Partial Precision filled w/ Empty PEs		DCT [1x1–8x8]	8
Partial Precision and Motion Estimation (ME) PEs filled w/ Empty PEs	7x7 DCT	ME PE [1]	1
	6x6 DCT	ME PEs [1–2]	2
	5x5 DCT	ME PEs [1–3]	3
	4x4 DCT	ME PEs [1–4]	4
	3x3 DCT	ME PEs [1–5]	5
	2x2 DCT	ME PEs [1–6]	6
	1x1 DCT	ME PEs [1–7]	7
No DCT—Motion Estimation Only		ME PEs [1–8]	8
		TOTAL	**80**

hardware arrangements. Rows 4-8 list the remaining Full Precision DCT arrangements with the corresponding number of Motion Estimation PEs that may be implemented and the hardware arrangement possible.

The next 8 rows of the table duplicate the possible hardware arrangements when implementing Partial Precision DCT functions. The last row of the table lists an additional 8 hardware arrangements possible when neither Full or Partial Precision DCT function is used and only Motion Estimation PEs are used. Similar to the Full and Partial PEs, the number of Motion Estimation PEs that may be implemented at any one time ranges from 1 to 8, which also results in 8 unique hardware arrangements, increasing the total to 80.

5.5 Chapter Summary

The scalable architecture exploits all of the benefits of partial reconfiguration discussed in Section 1.4. Any change made by partial reconfiguration to the type or number of PEs for the DCT function does not affect the other video compression functions. Since partial reconfiguration reduces the reconfiguration granularity, adding small-sized partial bitstreams increases the capability of the FPGA without large storage requirements. As demonstrated, a 1.63 MB bitstream must be generated and stored for each hardware arrangement when not using partial reconfiguration. Without the use of partial reconfiguration, 16 hardware arrangements require 16 full-device bitstreams totaling 26.13 MB. By using partial reconfiguration, 16 hardware arrangements require 1 full-device bitstream and 24 partial bitstreams only totaling 2.15 MB.

With respect to reducing configuration time, the best-case reconfiguration time of a full-device reconfiguration scheme is estimated to be 17 ms. The worst-case reconfiguration time in the partial reconfiguration implementation is decreased significantly to 2.265 ms. An additional 8 partial bitstreams totaling 221 KB further increases the capability of the FPGA to 80 hardware arrangements. When compared to the non-partial reconfiguration option, partial reconfiguration increases the capability of the FPGA with a significant decrease in reconfiguration time while maintaining similar storage requirements.

CHAPTER 6
CONCLUSION

Whereas the vast majority of FPGA fault-handling is deterministic, either by depending upon knowledge of the fault location or providing alternative solutions prior to the fault occurring, this thesis develops and optimizes techniques of active regeneration of lost functionality. Considering the experiments repairing the 3-bit adder in CHAPTER 3, many of these deterministic methods could repair the FPGA circuit given the simulation model. For example, the Incremental Rerouting method discussed in Section 2.3.1.1 could shift the LUTs away from the fault towards one of the two available spares. Since the simulation only considers the logic portion of the application, modifying the input values of the appropriate LUTs would take negligible time. Moreover, as long as spare LUTs are provided the success of these deterministic methods would remain constant while the complexity of the application increases.

Nonetheless, methods incorporating GAs are stochastic processes so their results are not deterministic. GAs perform best when solving problems whose solution quality can be definitively assessed yet the paths to obtain such solutions cannot be precisely defined. Although the standard GA was improved, the best implementation shown in CHAPTER 3 could repair a simple 3-bit adder only 5% of the time. Additionally, the computational time required to generate a complete repair is significantly larger than that required by an Incremental Rerouting algorithm. Since FPGA fault-detection techniques are available, the solution to a specific fault scenario can be determined, as used by Emmert et al. [2007]. As discussed in Section 2.3.1.3, improvements in the performance of GAs have been observed when considering the location of a fault—an improvement that this thesis does not make.

The simulations used only consider the logical portion of the FPGA application. Future work includes incorporating routing information into the simulator to consider the limited number of routing channels on an FPGA device. This addition for evaluating the fitness of individuals within the population would result in additional computation complexity for the GA. Since the success of GAs is heavily dependant upon the simplicity of the circuit, increasing the complexity of the application would amplify its limitations compared to deterministic approaches. The limitations of the GA, however, are offset by its ability to exploit faulty resources. Whereas an incremental rerouting algorithm can only bypass faulty PLBs, possibly leaving behind some functional aspects, a GA can provide a solution that uses the faulty PLB in a constructive manner. This makes the GA fault-handling method more amenable to more difficult fault scenarios, including the case where more faults exist than there are spare PLBs.

Solutions derived in simulated environments may not easily translate to the actual faulty device. In lieu of simulations, intrinsic evolution creates individuals with genetic operators and tests them directly on the FPGA device to obtain its fitness value. Research has manipulated the FPGA bitstream directly [Oreifej et al. 2007], eliminating the need to create an object-oriented chromosome representation. Crossover and mutation may have more success manipulating a string of bits rather than the object-oriented chromosome utilized by the simulations herein. Additionally if the repair area is small relative to the device, intrinsic evolution would benefit from the shorter reconfiguration times of partial reconfiguration, as each repair process requires configuring the FPGA a number of times equal to the population size multiplied by the number of generations. CHAPTER 4 also benefits from this reduction in configuration time whereas CHAPTER 5 mandates this reduction to implement a dynamic architecture for video processing where time is critical to the operation of the application.

The size of the reconfigurable area directly affects the configuration time. The partial reconfiguration design flow for ISE 6.3i enables reconfigurable areas to reside on the FPGA device. Since the configuration frame of the Virtex-II extends the entire height of the device, each reconfigurable area must also extend the entire height of the device, whereas its width may be variable which makes it a partial configuration instead of a full configuration. Modules whose width is less than the entire device will generate partial bitstreams with reduced configuration times.

For reconfigurable modules that do not require use of the entire height of the device, the Early Access Partial Reconfiguration (EAPR) design flow exploits the glitchless reconfiguration hardware feature of Virtex-II devices to bypass this software limitation. As such, reconfigurable modules may be any-sized rectangle where non-reconfigurable or static modules utilize unused resources within the same column, minimizing the area utilization of the application. Although reconfigurable modules may only use a small portion of the device, the full-column frame forces the configuration time of the reconfigurable module to remain the same. To reduce the configuration time, the Virtex-4 hardware architecture reduces the size of its configuration frame from the entire device as in the Virtex-II family to 16 CLBs. The capability of the EAPR design flow, therefore, is expanded to support multiple reconfigurable modules within one column of resources.

CHAPTER 4 shows how configuration times are reduced when the finer reconfiguration granularity offered by the EAPR software design flow is replicated by the hardware architecture as in the newer Virtex-4 FPGA device. The DCT function benefits from the EAPR software design flow and the corresponding hardware architecture by implementing eight partial reconfiguration areas, four of which reside within the same columns of resources on the left side of the device whereas the other four reside within the same columns of resources on the right side. The reduced area utilization from the EAPR software design flow results in eighty unique hardware arrangements that only require 2.36 MB of external storage, whereas a non-partial reconfiguration implementation requires 130.66 MB. The configuration time of a reduced frame results in a worst-case time of 2.265 ms and a best-case time of 0.106 ms, whereas a non-partial reconfiguration implementation requires 17 ms. Additionally, partial reconfiguration can increase the number of resources available to the video processor as a whole by time multiplexing between various sub-processing functions, such as DCT and motion estimation. Lastly, the reconfiguration process occurs without disturbing the remaining components of the FPGA, allowing sub-processing functions to adapt to user's requirements independently.

The ideal FPGA implementation for partial reconfiguration includes utilizing a soft-core processor or one of the internal PowerPC processors to control the reconfiguration process. The application implemented in CHAPTER 4 requires the user to configure the desired hash algorithm using the Xilinx tools. Utilizing an internal processor would allow a user to simply press a key on the PC keyboard attached to the FPGA to select the desired hashing algorithm. Upon selecting the desired hashing algorithm, the FPGA would then partially reconfigure itself to change the hash algorithm. Additionally, the SelectMAP interface could be used to decrease the observed configuration time from 2 seconds to 3.6μs, making the transition delay negligible to the user. The application presented in CHAPTER 5 would also greatly benefit from implementing an internal processor. In this implementation, a user would request a desired video quality using some communication channel and the FPGA would automatically adjust its configuration to implement the necessary subfunctions to achieve such a quality. Under either of these implementations, the partial reconfiguration process would only need an external storage element such as an Erasable Programmable Read-Only Memory (EPROM) or compact flash device, creating a dynamic, self-contained solution.

As promising as these results are, the complete capability of partial reconfiguration remains to be discovered. Current implementations of genetic algorithms, for example, may require processors external to the FPGA increasing spatial requirements of fault-handling techniques for deep-space missions where user intervention is limited or non-existent. As software design-flows improve, FPGA architectures become more integrated, and clock frequencies increase, FPGAs will increasingly become standalone platforms for evolvable and adaptable systems.

APPENDIX:
FIGURE 1-1—PERMISSION TO REPRINT

Dear Matthew Parris,

We hereby grant you permission to reprint the material detailed below at no charge in your thesis subject to the following conditions:

1. If any part of the material to be used (for example, figures) has appeared in our publication with credit or acknowledgement to another source, permission must also be sought from that source. If such permission is not obtained then that material may not be included in your publication/copies.

2. Suitable acknowledgment to the source must be made, either as a footnote or in a reference list at the end of your publication, as follows:

   "This article was published in Publication title, Vol number, Author(s), Title of article, Page Nos, Copyright Elsevier (or appropriate Society name) (Year)."

3. Your thesis may be submitted to your institution in either print or electronic form.

4. Reproduction of this material is confined to the purpose for which permission is hereby given.

5. This permission is granted for non-exclusive world English_rights only. For other languages please reapply separately for each one required. Permission excludes use in an electronic form other than submission. Should you have a specific electronic project in mind please reapply for permission.

6. Should your thesis be published commercially, please reapply for permission.

This includes permission for UMI to supply single copies, on demand, of the complete thesis. Should your thesis be published commercially, please reapply for permission.

Kind regards,
Marek Gorczyca
Rights Assistant
Elsevier LTD
Phone number: +441865843841
Fax number: +441865853333
m.gorczyca@elsevier.com

LIST OF REFERENCES

ACTEL 2005. Axcelerator Family FPGAs, http://www.actel.com/documents/AX_DS.pdf.

ALTERA 2008. Stratix IV Device Handbook, http://www.altera.com/literature/hb/stratix-iv/stratix4_handbook.pdf.

CARMICHAEL, C., CAFFREY, M. and SALAZAR, A. 2000. Correcting Single-Event Upsets Through Virtex Partial Configuration. *Xilinx Application Notes 216*.

CHEATHAM, J.A., EMMERT, J.M. and BAUMGART, S. 2006. A Survey of Fault Tolerant Methodologies for FPGAs. *ACM Transactions on Design Automation of Electronic Systems (TODAES) 11*, 501-533.

DEMARA, R.F. and ZHANG, K. 2005. Autonomous FPGA fault handling through competitive runtime reconfiguration. In *Proceedings of the NASA/DoD Conference on Evolvable Hardware*, Washington D.C., U.S.A., 2005, 109-116.

DUTT, S., SHANMUGAVEL, V. and TRIMBERGER, S. 1999. Efficient Incremental Rerouting for Fault Reconfiguration in Field Programmable Gate Arrays. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 1999, 173-176.

EMMERT, J.M. and BHATIA, D.K. 2000. A Fault Tolerant Technique for FPGAs. *Journal of Electronic Testing 16*, 591-606.

EMMERT, J.M., STROUD, C.E. and ABRAMOVICI, M. 2007. Online Fault Tolerance for FPGA Logic Blocks. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on 15*, 216-226.

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

GROSSO, P.B. 1985. Computer simulations of genetic adaptation: parallel subcomponent interaction in a multilocus model University of Michigan, 198.

HANCHEK, F. and DUTT, S. 1998. Methodologies for tolerating cell and interconnect faults in FPGAs. *Computers, IEEE Transactions on 47*, 15-33.

HORNBY, G.S. 2006. ALPS: the Age-layered Population Structure for Reducing the Problem of Premature Convergence. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, Seattle, Washington, USA, 2006 ACM.

HUANG, J., PARRIS, M., LEE, J. and DEMARA, R.F. 2008. Scalable FPGA Architecture for DCT Computation using Dynamic Partial Reconfiguration. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, Las Vegas, Nevada, USA, 2008.

JÄRVINEN, K.U., TOMMISKA, M.T. and SKYTTÄ, J.O. 2005. A Compact MD5 and SHA-1 Co-Implementation Utilizing Algorithm Similarities. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, Las Vegas, Nevada, USA, 2005.

KAO, C. 2005. Benefits of Partial Reconfiguration. *Xcell Journal Fourth Quarter, 2005*, 65-67.

KATZ, D.S. and SOME, R.R. 2003. NASA advances robotic space exploration. In *Computer*, 52-61.

KEYMEULEN, D., ZEBULUM, R.S., JIN, Y. and STOICA, A.A.S.A. 2000. Fault-tolerant evolvable hardware using field-programmable transistor arrays. *Reliability, IEEE Transactions on 49*, 305-316.

KIZHNER, S., PATEL, U.D. and VOOTUKURU, M. 2007. On Representative Spaceflight Instrument and Associated Instrument Sensor Web Framework. In *Proceedings of the IEEE Aerospace Conference*, 2007, U.D. PATEL Ed., 1-10.

LACH, J., MANGIONE-SMITH, W.H. and POTKONJAK, M. 1998. Low overhead fault-tolerant FPGA systems. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on 6*, 212-221.

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

LEE, J., VIJAYKRISHNAN, N., IRWIN, M.J. and CHANDRAMOULI, R. 2006a. Block-based frequency scalable technique for efficient hierarchical coding. *Signal Processing, IEEE Transactions on [see also Acoustics, Speech, and Signal Processing, IEEE Transactions on] 54*, 2559-2566.

LEE, J., VIJAYKRISHNAN, N., IRWIN, M.J. and WOLF, W. 2006b. An Efficient Architecture for Motion Estimation and Compensation in the Transform Domain. *Circuits and Systems for Video Technology, IEEE Transactions on 16*, 191-201.

LOHN, J., LARCHEV, G. and DEMARA, R. 2003. Evolutionary fault recovery in a Virtex FPGA using a representation that incorporates routing. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, 8 pp.

LYSAGHT, P., BLODGET, B., MASON, J., YOUNG, J.A.Y.J. and BRIDGFORD, B.A.B.B. 2006. Invited Paper: Enhanced Architectures, Design Methodologies and CAD Tools for Dynamic Reconfiguration of Xilinx FPGAs. In *Proceedings of the International Conference on Field Programmable Logic and Applications*, 2006, 1-6.

MAHFOUD, S.W. 1992. Crowding and preselection revisited. In *Parallel Problem Solving from Nature*, R. MÄNNER and B. MANDERICK Eds.

MAXFIELD, C.M. 2004. *The Design Warrior's Guide to FPGAs*. Newnes.

MILLER, J.F., THOMSON, P. and FOGARTY, T. 1997. Designing Electronic Circuits Using Evolutionary Algorithms. Arithmetic Circuits: A Case Study. In *Genetic Algorithms and Evolution Strategies in Engineering and Computer Science*, D. QUAGLIARELLA, J. PERIAUX, C. POLONI and G. WINTER Eds. Wiley, 105–131.

MITCHELL, M. 1996. *An Introduction to Genetic Algorithms*. Mit Pr.

MITRA, S., HUANG, W.J., SAXENA, N.R., YU, S. and MCCLUSKEY, E.J. 2004. Reconfigurable architecture for autonomous self-repair. *Design & Test of Computers, IEEE 21*, 228-240.

NIST 1995. SECURE HASH STANDARD. *Federal Information Processing Standards Publications 180-1*.

OREIFEJ, R.S., AL-HADDAD, R.N., HENG, T. and DEMARA, R.F. 2007. Layered Approach to Intrinsic Evolvable Hardware using Direct Bitstream Manipulation of Virtex II Pro Devices. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, 299-304.

OREIFEJ, R.S., SHARMA, C.A. and DEMARA, R.F. 2006. Expediting GA-Based Evolution Using Group Testing Techniques for Reconfigurable Hardware. In *Proceedings of the IEEE International Conference on Reconfigurable Computing and FPGA's*, 2006, 1-8.

PETTEY, C.B., LEUZE, M.R. and GREFENSTETTE, J.J. 1987. A parallel genetic algorithm. In *Proceedings of the Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application*, Cambridge, Massachusetts, United States, 1987 Lawrence Erlbaum Associates, Inc.

RATTER, D. 2004. FPGAs on Mars. *Xcell Journal Third Quarter, 2004*, 8-11.

RIVEST, R. 1992. The MD5 Message-Digest Algorithm. *Request for Comments 1321*.

ROSS, R. and HALL, R. 2006. A FPGA Simulation Using Asexual Genetic Algorithms for Integrated Self-Repair. In *Proceedings of the First NASA/ESA Conference on Adaptive Hardware and Systems*, 2006, 301-304.

SCHWEFEL, H.-P. and RUDOLPH, G. 1995. Contemporary Evolution Strategies In *Advances in Artificial Life: Third European Conference on Artificial Life Granada, Spain, June 4–6, 1995 Proceedings* Springer Berlin / Heidelberg, 891-907.

SHANTHI, A.P., VIJAYAN, B., RAJENDRAN, M., VELUSWAMI, S. and PARTHASARATHI, R. 2002. GA Based On-line Testing and Recovery for Critical Digital Systems. In *Proceedings of the HiPC Workshop on Soft Computing*, Bangalore, India, 2002, 81-89.

SHARMA, C.A., DEMARA, R.F. and SARVI, A. 2007. Self-Healing Reconfigurable Logic Using Autonomous Group Testing. submitted to *ACM Transactions on Autonomous and Adaptive Systems (TAAS) of Special Issue on Organic Computing*. May, 2007.

SWIFT, G.M. 2006. Radiation Effects and Field Programmable Gate Arrays. In *Proceedings of the Single Event Effects Symposium*, Long Beach, CA, 2006.

TAN, H. and DEMARA, R.F. 2007. A Multi-layer Framework Supporting Autonomous Runtime Partial Reconfiguration. accepted to *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*.

VIGANDER, S. 2001. Evolutionary Fault Repair of Electronics in Space Applications. In *Department of Computer and Information Science* Norwegian University of Science and Technology (NTNU), 50.

WELLS, B.E. and LOO, S.M. 2001. On the Use of Distributed Reconfigurable Hardware in Launch Control Avionics. In *Proceedings of the 20th Digital Avionics Systems*, Daytona Beach, Florida, USA, 2001.

WIKIPEDIA Current. MD5, http://en.wikipedia.org/wiki/MD5.

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

XILINX 2006. Early Access Partial Reconfiguration. *User Guide 208*.

XILINX 2007a. Difference-Based Partial Reconfiguration. *Application Note 290*.

XILINX 2007b. Virtex-4 Family Overview. *Xilinx Data Sheet 112*.

XILINX 2008. Virtex-4 FPGA Configuration. *Xilinx User Guide 071*.

YUI, C.C., SWIFT, G.M. and CARMICHAEL, C. 2003. SEU Mitigation of Xilinx Virtex II FPGAs for Critical Flight Applications. In *Proceedings of the IEEE Nuclear and Space Radiation Effects Conference*, 2003.

ZHANG, K., BEDETTE, G. and DEMARA, R.F. 2006. Triple Modular Redundancy with Standby (TMRSB) Supporting Dynamic Resource Reconfiguration. In *Proceedings of the IEEE Systems Readiness Technology Conference*, 2006, 690-696.