HEXARRAY: A NOVEL SELF-RECONFIGURABLE HARDWARE SYSTEM

by

Fady Hussein

A dissertation

submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy in Electrical and Computer Engineering

Boise State University

May 2017

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the dissertation submitted by

Fady Hussein

Dissertation Title: HexArray: A Novel Self-Reconfigurable Hardware System

Date of Final Oral Examination: 13 March 2017

The following individuals read and discussed the dissertation submitted by student Fady Hussein, and they evaluated his presentation and response to questions during the final oral examination. They found that the student passed the final oral examination.

Nader Rafla, Ph.D.	Chair, Supervisory Committee
Elisa Barney Smith, Ph.D.	Member, Supervisory Committee
Jennifer A. Smith, Ph.D.	Member, Supervisory Committee

The final reading approval of the dissertation was granted by Nader Rafla, Ph.D., Chair of the Supervisory Committee. The dissertation was approved by the Graduate College.

This is for y’all, Mona, Eliana, Joanna, and Kareem.

ACKNOWLEDGMENTS

I am using this opportunity to express my gratitude to everyone who supported me throughout the course of this dissertation. I am thankful for their aspiring guidance, invaluably constructive criticism and friendly advice during the project work, specifically my advisor (Dr. Nader Rafla) and his research team, Luka Daoud and Shelton Jacinto.

AUTOBIOGRAPHICAL SKETCH

Fady Hussein is a PhD Candidate at Electrical and Computer Engineering at Boise State University. He joined the PhD program on August 2013. He received his MSc degree from Louisiana State University and bachelors degree in Electrical Engineering at Birzeit University, Palestine. Fady is, currently, a senior test engineer for DRAM at Micron Technology in Boise, Idaho. His main research focuses on evolvable hardware and reconfigurable computing. Currently, he is developing a framework for an evolvable system that can be utilized in design automation, image processing, reverse engineering and fault-tolerant systems.

ABSTRACT

Evolvable hardware (EHW) is a powerful autonomous system for adapting and finding solutions within a changing environment. EHW consists of two main components: a reconfigurable hardware core and an evolutionary algorithm. The majority of prior research focuses on improving either the reconfigurable hardware or the evolutionary algorithm in place, but not both. Thus, current implementations suffer from being application oriented and having slow reconfiguration times, low efficiencies, and less routing flexibility. In this work, a novel evolvable hardware platform is proposed that combines a novel reconfigurable hardware core and a novel evolutionary algorithm.

The proposed reconfigurable hardware core is a systolic array, which is called HexArray. HexArray was constructed using processing elements with a redesigned architecture, called HexCells, which provide routing flexibility and support for hybrid reconfiguration schemes. The improved evolutionary algorithm is a genome-aware genetic algorithm (GAGA) that accelerates evolution. Guided by a fitness function the GAGA utilizes context-aware genetic operators to evolve solutions. The operators are genome-aware constrained (GAC) selection, genome-aware mutation (GAM), and genome-aware crossover (GAX). The GAC selection operator improves parallelism and reduces the redundant evaluations. The GAM operator restricts the mutation to the part of the genome that affects the selected output. The GAX operator cascades, interleaves, or parallel-recombines genomes at the cell level to generate better genomes. These operators improve evolution while not limiting the algorithm from exploring all areas of a solution space.

The system was implemented on a SoC that includes a programmable logic (i.e., field-

programmable gate array) to realize the HexArray and a processing system to execute the GAGA. A computationally intensive application that evolves adaptive filters for image processing was chosen as a case study and used to conduct a set of experiments to prove the developed system robustness. Through an iterative process using the genetic operators and a fitness function, the EHW system configures and adapts itself to evolve fitter solutions. In a relatively short time (e.g., seconds), HexArray is able to evolve autonomously to the desired filter.

By exploiting the routing flexibility in the HexArray architecture, the EHW has a simple yet effective mechanism to detect and tolerate faulty cells, which improves system reliability. Finally, a mechanism that accelerates the evolution process by hiding the reconfiguration time in an "evolve-while-reconfigure" process is presented. In this process, the GAGA utilizes the array routing flexibility to bypass cells that are being configured and evaluates several genomes in parallel.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

2D – Two Dimensional

AOC – Array Output Controller

AIC – Array Input Controller

ALU – Arithmetic Logic Unit

API – Application Programming Interface

ASMBL – Advanced Silicon Modular Block

AXI – Advanced Extensible Interface

BRAM – Block Random-Access Memory

CAD – Computer-Aided Design

CLB – Configurable Logic Block

CMOS – Complementary Metal-Oxide Semiconductor

CPLD – Complex Programmable Logic Device

DMA – Direct Memory Address

DNA – Deoxyribonucleic Acid

DPR – Dynamic Partial Reconfiguration

DSP – Digital Signal Processor

EA – Evolutionary Algorithm

EDA – Electronic Design Automation

EEPROM – Electrically Erasable Programmable Read-Only Memory

EHW – Evolvable Hardware

EmHW – Embryonic Hardware

EPROM – Erasable Programmable Read-Only Memory

FFT – Fast Fourier Transform

FIFO – First In, First Out

FIR – Finite Impulse Response

FPAA – Field-Programmable Analog Array

FPGA – Field-Programmable Gate Array

FPTA – Field-Programmable Transistor Array

GA – Genetic Algorithm

GAC – Genome-Aware Constrained

GAGA – Genome-Aware Genetic Algorithm

GAL – Generic Array Logic

GAM – Genome-Aware Mutation

GAP – Genome-Aware Pruner

GAX – Genome-Aware Crossover

HCPLD – High Capacity Programmable Logic Device

HDL – Hardware Description Language

HexArray – Hexagonal Systolic Array

HexCell – Hexagonal Cell

HWICAP – Hardware Internal Configuration Access Port

ICAP – Internal Configuration Access Port

IOB – Input/output Block

IP – Intellectual Property

ISE – Integrated Synthesis Environment

JTAG – Joint Test Action Group

LAB – Logic Array Block

LE – Logic Element

LUT – Look-Up Table

MAE – Mean Absolute Error

MIMD – Multiple Instruction, Multiple Data

PE – Processing Element

PCAP – Processor Configuration Access Port

PL – Programmable Logic

PAL – Programmable Array Logic

PLA – Programmable Logic Array

PLD – Programmable Logic Device

PROM – Programmable Read-Only Memory

PS – Processing System

PSNR – Peak Signal-to-Noise Ratio

RAM – Random-Access Memory

rDPA – reconfigurable Data Path Array

RectCell – Rectangular Cell

ROAM – Read-Only Associative Memory

SNR – Signal-to-Noise Ratio

SoC – System on Chip

SPLD – Simple Programmable Logic Device

SQL – Structured Query Language

SRAM – Static Random Access Memory

TECS – Transactions on Embedded Computing Systems

VHDL – VHSIC Hardware Description Language

VHSIC – Very High Speed Integrated Circuit

VRC – Virtual Reconfiguration Circuit

LIST OF SYMBOLS

$\simeq$	is similar or equal to
λ	number of child genomes (offsprings)
μ	number of parent genomes or micro (10^{-6}) if used for time
∞	infinity
g_i	genome i
g_r	randomly generated genome
g_m	genome generated by mutation
g_c	genome generated by crossover
g_p	selected parent genome
g_s	selected child genome
$G_{parents}$	list of parent genomes
$G_{children}$	list of children genomes
$r \times c$	an array or a window of a r rows and c columns
$\in$	is member of
$\mathbb{R}$	number of rows in HexArray
$\mathbb{C}$	number of columns in HexArray
$\mathbb{L}$	number of rows or columns in a symmetric HexArray

$\oplus$ bitwise XOR operation

M_{GAM} number of bits to be mutated using GAM

$M_{mutation}$ number of bits to be mutated using traditional mutation

A the North input port of a HexCell

B the North West input port of a HexCell

C the South West input port of a HexCell

f the output port of the functional unit of a HexCell

X the North East output port of a HexCell

Y the South East output port of a HexCell

Z the South output port of a HexCell

Sel_x the selection signal for the North East output port of a HexCell

Sel_y the selection signal for the South East output port of a HexCell

Sel_z the selection signal for the South output port of a HexCell

Sel_{func} the selection signal for functional unit of a HexCell

f_i function i – a possible value for Sel_{func}

$\lfloor\ \rfloor$ floor operation of the given number, e.g., $\lfloor 12.6 \rfloor = 12$

& bitwise AND

| bitwise OR

$\sim$ bitwise NOT

$\ll$ shift left

$\gg$ shift right

$0x..$ base 16 integer number (i.e., hexadecimal number)

L image (horizontal) length in pixels

W image (vertical) width in pixels

$Out(m,n)$ HexArray output pixel

$Ref(m,n)$ reference pixel

$\sum|entries|$ summation of the absolute value of all entries

$PE_{1,2}$ processing element at row 1 and column 2

O_3 HexArray output port number 3

f_{system} frequency of the system

MAE_{Norm} normalized mean absolute error

MAE_{Init} initial mean absolute error (i.e., training image MAE)

Sel_o the selection signal for the Cartesian systolic array output

log_{10} logarithmic scale with base 10

Listings

List of Algorithms

CHAPTER 1

INTRODUCTION

The design specifications given to hardware designers for today's applications have become more challenging. Systems that support increasingly complex functions are required to have shorter design times and higher flexibility and adaptability to changing environments; these requirements are needed while meeting time, space, and power constraints. Furthermore, some systems may have an unpredicted environment, or the design specifications are not able to fully describe the problem. Consequently, for these applications and many others, *evolvable hardware* is used to automate the design process or dynamically and autonomously adapt the system to a changing environment. By examining the trend of advances made in the evolvable hardware field over the past 50 years, one can predict that there will be further advances and widespread adoption in the near future.

As shown in figure 1.1, computational intelligence is a subfield of artificial intelligence that studies the mechanisms that enable intelligent behavior in complex and changing environments [19]. The main focus of computational intelligence is to design and utilize heuristic algorithms to solve complex real-world problems. The branches of this field are machine learning, evolutionary computation, fuzzy logic, and probabilistic reasoning. Evolutionary computation is the field in which the theory of evolution is used in computing systems. Its main areas are artificial neural networks, bio-inspired hardware, swarm intelligence, and artificial immune systems. Bio-inspired hardware, short for biologically

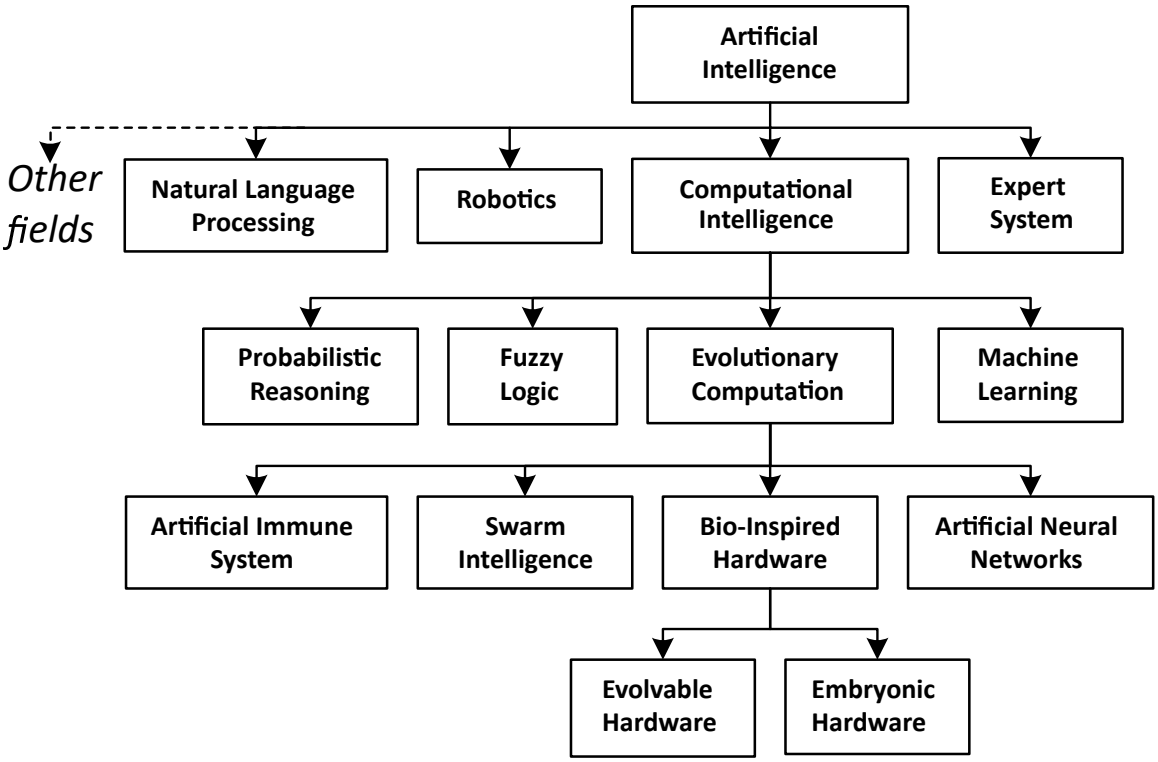


Figure 1.1: Evolvable hardware and embryonic hardware are the main branches of bio-inspired hardware.

inspired hardware, is the research domain that relates the natural principles with electronic systems. Evolvable hardware and embryonic hardware (EmHW) are the two main types of bio-inspired hardware [20].

Evolvable hardware is defined as a hardware system that is capable of real-time adaptation by reconfiguring internal hardware dynamically and autonomously [21]. Conversely, EmHW can be defined as a hardware system with self-healing capability. Self-healing (self-diagnosis, self-repair ability, or self-replication) is accomplished by in-cell failure detection mechanisms, such as double modular redundancy or triple modular redundancy [22], or out-of-cell mechanisms, where nearby cells can detect failing cells [20]. Since systems with real-time adaptation are the focus of our research, evolvable hardware is

discussed here.

1.1 Evolvable Hardware (EHW)

Evolvable hardware[1] is the field in which biological concepts are implemented in electrical hardware using computer science algorithms, as shown in figure 1.2.

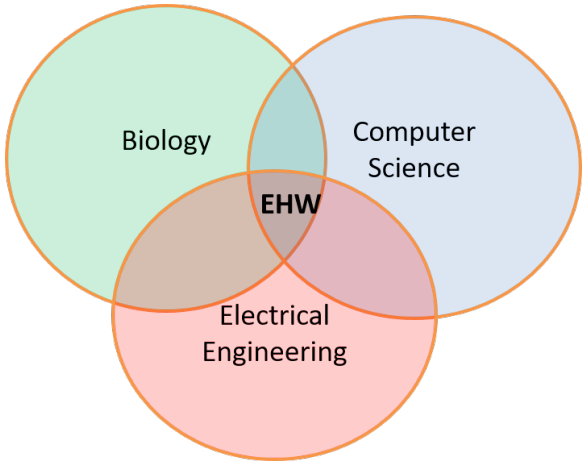


Figure 1.2: EHW is the field where biology, electrical engineering, and computer science meet.

The applications of EHW are classified into two main categories:

- *EHW for Solving Design Problems:* For complicated design problems, EHW can be used to find a solution based on a set of specified criteria. The environment where the EHW is running is generally fully described, and the output is deterministic. Since the search space is large (2^N, where N $>$ 100) and the EHW is often running in an extrinsic mode (e.g., simulation), finding a solution typically takes a few days up to weeks depending on the size of the search space, the complexity of the problem,

[1] Unfortunately, in many studies, evolvable hardware, evolutionary computation, artificial evolution, and evolutionary electronics are often used interchangeably.

and the objective function. At the end of the evolution, many possible solutions may be generated, but only one solution must be selected to be implemented on hardware. Most of the early applications of EHW can be categorized in this category [23, 24, 25, 26, 27].

- *EHW for Online Adaptation:* For applications with a changing environment, EHW can be used to improve adaptivity at different levels, including fault detection and tolerance. EHW is fast because it is often running in an intrinsic mode (online mode, i.e., on hardware) and evolution is completely autonomous. Running in this mode requires reconfigurable hardware platforms, which were not common until recently. Therefore, many of these implementations are recent, as in [28, 1, 29, 30].

1.2 Main Components of Evolvable Hardware

As shown in figure 1.3, EHW consists of two components: a reconfigurable hardware core and an evolutionary algorithm – the body and the brain.

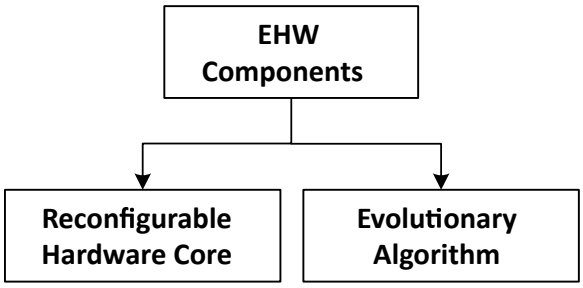


Figure 1.3: Main components of EHW.

1.2.1 Reconfigurable Hardware Core

A reconfigurable hardware core is the medium to embody possible solutions generated by the evolutionary algorithm. The hardware core can be implemented in one of four possible architectures:

1. Programmable logic devices used for digital designs.
2. Field-programmable analog arrays (FPAAs) used for analog designs.
3. Field-programmable transistor arrays (FPTAs) used for low-level mixed signal designs.
4. Custom hardware architectures used for application-specific digital, analog, or mixed systems.

Since the focus of our research is digital systems, programmable logic devices, specifically field-programmable gate arrays (FPGAs), and digital custom hardware architectures will be discussed in Chapter 2 in more detail.

A hardware architecture is considered to be a reconfigurable hardware core if it meets certain requirements, including the following:

- Supporting reconfiguration multiple times – the architecture can be reconfigured many or unlimited times.
- Supporting fast reconfiguration – since the search space is vast and millions of reconfigurations are needed, the reconfiguration speed is critical.
- Supporting partial reconfiguration – reconfiguration can occur partially on a subregion of the programmable logic (called dynamic region) without the need for reconfiguring the entire array. In other words, the architecture should allow fine-grained

reconfiguration, where small regions of the programmable logic are reprogrammed independently.

- Supporting dynamic reconfiguration – reconfiguring a (dynamic) region of a system should not affect other (static) regions.

- Being inexpensive, flexible, and reliable – general requirements for any usable architecture.

1.2.2 Evolutionary Algorithms (EAs)

In biology literature, *evolution* is defined as the change that occurs on inherited characteristics of populations over consecutive generations to better adapt to the environment [31]. In EHW, the term gene is used to represent a building block of a chromosome that represents the aggregation of heredity information. Chromosomes are collections of genes, and the collection of chromosomes of an individual is called a *genome*. Genomes in EHW can be represented by integers, real numbers, strings, graphs, or trees and can fully describe an individual (solution) [13]. Figure 1.4 shows the evolution cycle in biology with the equivalent cycle in electronics. Evolution is achieved by heuristic algorithms, algorithms that trade solution optimality for speed – evolutionary algorithms.

Evolutionary algorithms are bio-inspired computer algorithms that feature natural evolution and self-adaptation. These are search and optimization algorithms that attempt to find optimal (or at least suboptimal) solutions in a large search space where classical search methods are too slow. The search process, also known as the evolution process, is performed iteratively using *genetic operators* and one or more *evaluation functions*. EAs rely heavily on randomness, which makes the search process nondeterministic. This means

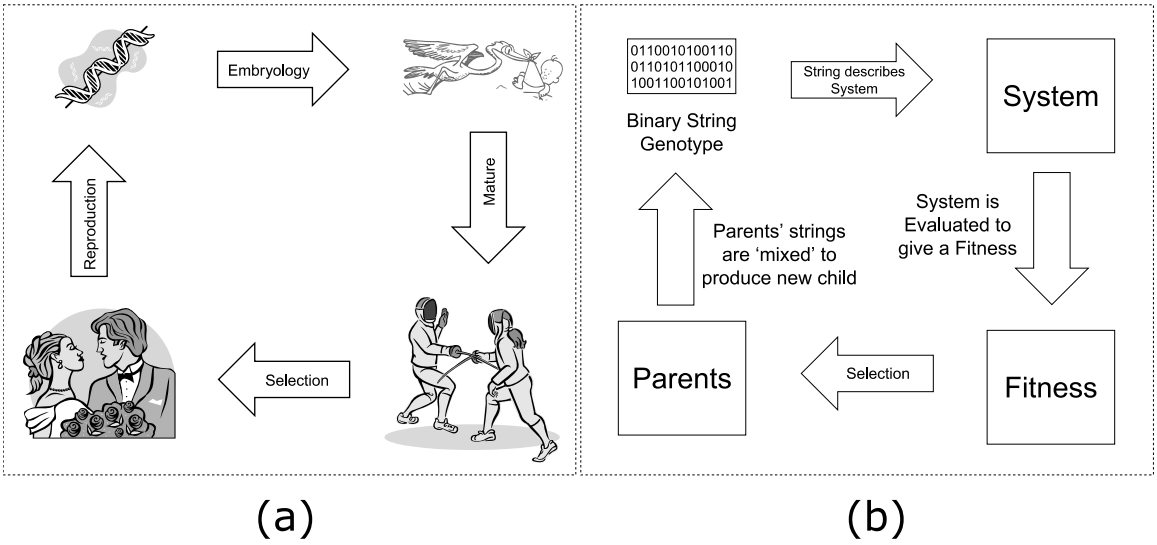


Figure 1.4: Evolution cycle between (a) biology and (b) electronics [1].

that re-running the search process results in finding different solutions. The subfunctions for an EA can be listed as follows:

- *Representation*: EA has to determine how to represent a hardware solution as a genome. In other words, a decoding function or criterion needs to be defined that allows mapping between two domains: the genotype (genome in the solution space) and phenotype (individual in the problem space). For example, for a digital circuit design, the genotype – a genome in the EA domain – can be represented by a string of bits, while the phenotype – an individual in electronics – might be configuration data for a circuit. Henceforth, genome, individual, and solution will used interchangeably in this work.

- *Population*: the collection of individuals in a generation is called population. It is generally fixed in size during evolution. The EA has to determine how to generate and manage populations. The two common approaches for managing population are

generational, where a population is erased and recreated after every generation, and overlapping, where a population is modified after every generation [32]. Population diversity is a critical aspect of a successful EA.

- *Evaluation function* (also called fitness function or objective function): the EA has to determine how to evaluate individuals. In other words, a fitness function needs to be defined. A fitness function is a gauge of how close an individual is to meeting the design specifications. In some applications, multiple fitness functions are defined. Fitness functions are discussed in section 3.3. There are three methods to evaluate the fitness in EHW: extrinsic [33], intrinsic [34], and mixtrinsic [35], which are discussed in the following section.

- *Genetic operators* (also called variation operators): the EA has to determine how genomes are raised and evolved in generations. These operators are selection, mutation and crossover, and they are discussed in section 3.2.

- *Termination condition*: the EA has to determine when to stop evolution, e.g., after evaluating a certain number of genomes or achieving a certain quality goal (i.e., fitness value).

Hardware Evolution Types

The type of evolution is determined by where individuals are evaluated, i.e., software and/or hardware models, and where the EA and fitness function are running. The three types are described as follows:

Extrinsic evolution is performed when an individual is modeled and evaluated on software. Subsequently, the solution with the best fitness is implemented in hardware. The simulation is an approximate modeling of hardware and, in some cases, may not behave

equally. In digital circuits, the software modeling can represent hardware with almost complete correctness [36, 37, 38, 39]. An example of this method is using a hardware description language (HDL) to simulate a circuit; then, based on the simulation results, the fittest circuit is replicated in hardware.

Intrinsic evolution is performed directly on hardware where a genome is modeled and evaluated on hardware. This method requires hardware with a fast reconfiguration time. Intrinsic evolution is fast, and since there is no software modeling, there is no hardware-to-software functional mismatching. There are four subcategories of intrinsic EHW, which are based on where the EA and fitness evaluation are performed. These are intrinsic, complete intrinsic, multi-chip intrinsic, and multi-board intrinsic, which are summarized in figure 1.5.

Mixtrinsic evolution is where individual solutions are modeled in a mixture of hardware and software models. This method solves the extrinsic evolution issue of mismatching by guaranteeing that a solution behaves well on hardware and during simulation. Complementary mixtrinsic and combined mixtrinsic are the two modes of operation for mixtrinsic evolution [35]. Complementary mixtrinsic evolution is where a genome is assigned randomly to a hardware model or software model. Combined mixtrinsic evolution is where a genome is modeled in software and hardware and an average fitness value is calculated when the two fitness values are mismatching.

1.3 Applications of EHW

The applications of EHW can be seen in many fields. This is because EHW allows online adaptation to environment changes, solves complex problems, and increases system reliability. One of the common uses of EHW is circuit synthesis: analog circuit design

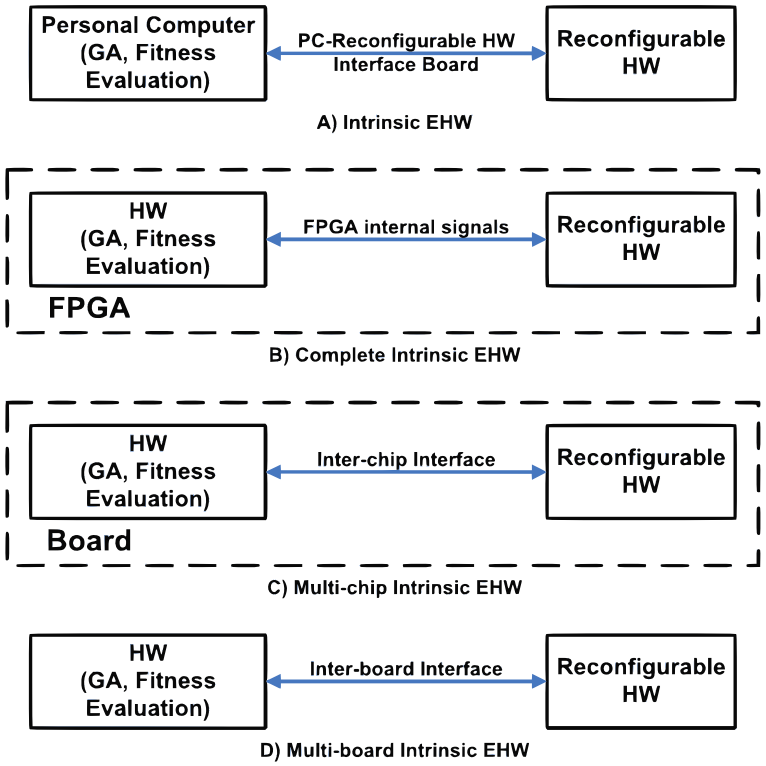


Figure 1.5: Intrinsic EHW types based on where the EA is running [2].

[40, 41, 42, 43, 44] and digital circuit design [45, 46, 47, 48, 49, 50, 51]. EHW is also used in image processing applications, such as developing adaptive filters [30, 52] and pattern recognition [53, 54, 21, 55]. In other cases, EHW is used to evolve systems with fault tolerance capabilities, such as fault-tolerant circuits [56, 57, 58], circuits with fault tolerance using natural redundancy [59], fault-tolerant image processors [60], and high-reliability space applications [61]. EHW has been used in solving some hard problems, such as the classical applications in [23] and NP-complete problems [62]. In the communication systems domain, EHW is used to adapt to different communication protocols [63] and autonomously optimize signal strength [64]. In computer hardware, EHWs have been used to evolve functional accelerators [65, 66]. Moreover, EHWs are utilized in neural networks

[67, 68, 15, 69, 70, 71], robotic and control systems [72, 73, 74, 75], data mining [76, 77], data compression [78, 79], data cryptography [80, 81, 82], medical applications [83], many analog applications as summarized on page 68 of [13], and many others.

1.4 Motivation and Research Objectives

The objectives and main areas proposed for investigation are summarized as follows:

1. Developing an autonomous system with accelerated evolution: The term accelerated evolution represents many subgoals, including developing a fast hardware core and an efficient evolutionary algorithm. In our work, we develop a set of EHWs where the hardware core and evolutionary algorithm are cooperating to improve evolution.

2. Achieving fast reconfiguration without sacrificing resources: Although this objective can be a subgoal of the previous one, fast reconfiguration is set as a goal by itself because reconfiguration with less overhead is a vital goal of EHW. Because there are advantages and disadvantages for both common types of reconfiguration schemes, a hybrid scheme is created that combines the merits of both schemes while still avoiding their drawbacks.

3. Improving reliability by fault detection and tolerance: In general, on an EHW system, self-adaptation is the main target, which is in contrast to EmHW, where self-healing is the main goal. The flexibility of the designed system allows for a simple yet efficient fault detection and tolerance mechanism.

4. Improving genetic algorithm to perform better genetic operations: The genetic operators of an EA are often independent of the underlying reconfigurable hardware

core. Although we believe it is a desirable abstraction, some modifications can still be made to make them perform better on systolic arrays.

1.5 Contributions

An efficient and complete intrinsic EHW platform is proposed. The system can be enclosed entirely on a commercial low-cost system-on-chip (SoC). The presented system features a novel FPGA-based reconfigurable hardware core (called HexArray). HexArray offers a high level of routing flexibility and combines the merits of the two common reconfiguration schemes.

In addition, the proposed system also features a novel genome-aware genetic algorithm (called GAGA), which is a context-aware genetic algorithm designed specifically for systolic arrays (such as HexArray). Moreover, some of the introduced genetic operations are applicable to a wide range of evolutionary algorithms and evolvable architectures. A collection of experiments shows that the proposed GAGA operators truly accelerate evolution. Additionally, the new architecture supports a simple but robust fault detection and tolerance mechanism. Moreover, a technique is proposed that allows the system to evaluate genomes while the arrays are being reconfigured.

In this work, we propose a novel EHW platform based on a new reconfigurable hardware core and evolutionary algorithm. Furthermore, additional features have been proposed that exploit the features of the new platform. The contributions in this dissertation are summarized as follows:

1. A novel processing element (called HexCell).

2. A novel systolic array of HexCells (called HexArray) featuring the following:

(a) A novel hybrid reconfiguration scheme.

(b) A novel fault detection and tolerance mechanism.

(c) A novel evolve-while-reconfigure technique.

3. A novel evolutionary algorithm (called GAGA) featuring the following:

(a) A novel genome-aware constrained selection (GAC selection).

(b) A novel genome-aware mutation (GAM).

(c) A novel genome-aware crossover (GAX) running in three different modes.

(d) A novel genome-aware pruner (GAP).

1.6 Dissertation Overview

Since an EHW is proposed in this dissertation, the first EHW component is discussed in Chapter 2. The discussion starts with the properties of reconfigurable hardware cores followed by the common architectures in the field. The next chapter, Chapter 3, discusses the other component of an EHW, evolutionary algorithms. This chapter explores the means of EAs, which are genetic operators and the fitness functions. Different common types of EAs are then briefly considered, with a special emphasis on genetic algorithms. Different classifications of EHWs are presented in Chapter 4, along with a variety of FPGA-based and custom-hardware implementations with a special focus on the systolic arrays.

The contributions of this work are presented in Chapter 5. The discussion starts with an early implementation of the proposed system in software – HexArray Simulator. The simulator served as a proof of concept, but discussing it permits a high-level understanding of the system without involving much complexity. In the next section, section 5.3, the hardware part of the system is presented in detail – HexCell and HexArray. The second

part (section 5.4) will be dedicated to presenting the GAGA and its utility functions and genome-aware genetic operators. Additional features enabled by the new system are provided in section 5.6. A comparison between HexArray and the state-of-the-art systolic array is presented at the end of the chapter.

Chapter 6 consists of a comprehensive set of experiments to test the new platform; the discussion in this part of the chapter is limited to the evolution speed (i.e., for a fixed number of genomes, what is the best fitness achieved). Section 6.3 discusses the implementation details, timing analysis, and resource utilization.

Finally, we conclude this dissertation by summarizing the contributions of this work. "What is the value of HexCell, HexArray, GAC Selection, GAM, and GAX?" will also be answered. Future work and suggestions are presented in section 7.1.

CHAPTER 2

RECONFIGURABLE HARDWARE CORE

2.1 Introduction

To develop an EHW system, a reconfigurable architecture and an evolutionary algorithm are needed. A reconfigurable architecture is a hardware system that can be programmed by the user or application. The hardware system is used to realize solutions suggested by the EA.

There are several technologies that enable programmability (and re-programmability). Early programmable architectures used programmable read-only memory (PROM). Since PROM was a one-time programmable memory, erasable PROM (EPROM) was introduced to allow for reprogramming several times. However, EPROM was erasable by long exposure to ultraviolet light, e.g., several minutes using an ultraviolet eraser machine. Consequently, electrically EPROM (EEPROM) was invented, which accelerated the erasing process and eliminated the need for an external device to erase the memory. Subsequently, flash memories were used. A flash device is a matrix of EEPROMs that are segmented into smaller blocks that can be independently erased. One aspect of all previous non-volatile devices is the limitation on how many times they can be reprogrammed. Another reconfigurable device is static RAMs (SRAMs), which are volatile devices used for fast configuration and unlimited reconfigurations. As mentioned previously in section 1.2.1, for an architecture to be considered a reconfigurable hardware core, it must support being

reconfigured multiple times.

2.2 Properties of Reconfigurable Hardware Cores

As shown in figure 2.1, there are several ways to characterize reconfigurable architectures, including architecture, interconnect, fabric structure and reconfiguration schemes. Although these terms are tightly connected, some key differences can still be outlined, as follows:

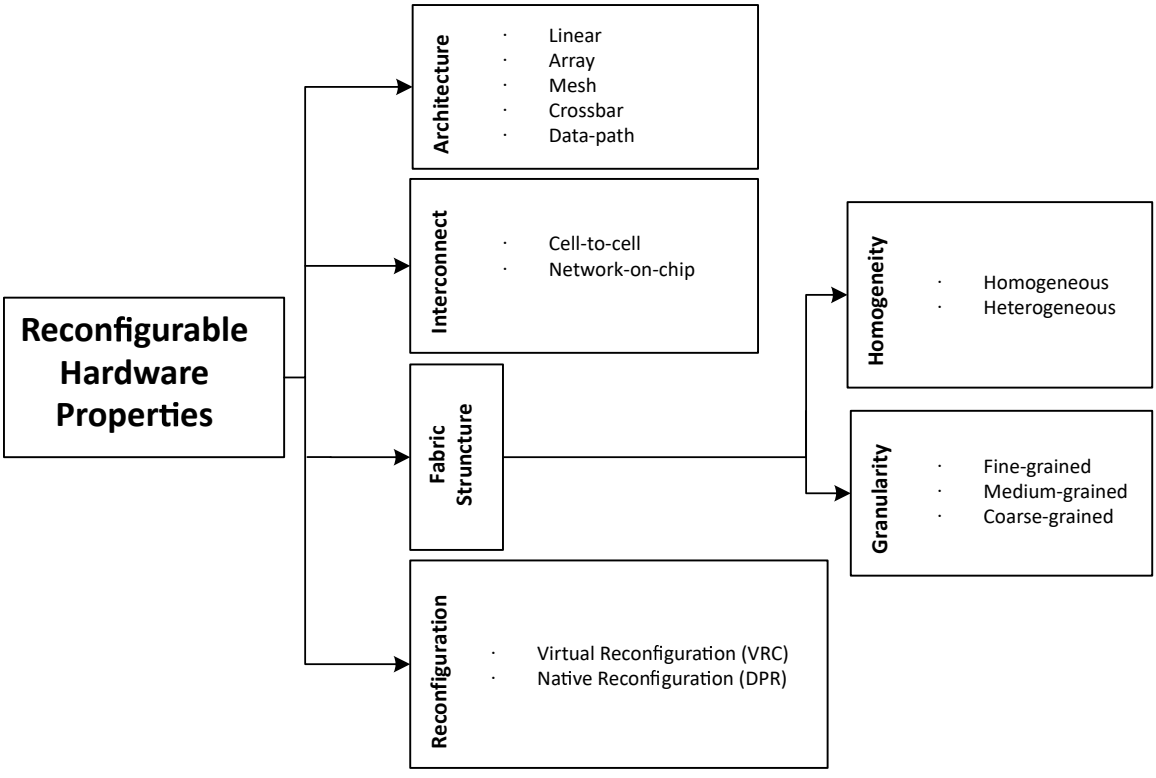


Figure 2.1: Hardware core properties in terms of structure and reconfiguration.

2.2.1 Architecture

Reconfigurable architectures generally have building blocks that are connected in a certain way – called the architecture of the platform. In other words, the architecture describes the connectivity scheme of processing elements (PEs) in a reconfigurable architecture. Common architectures are as follows:

1. Linear: PEs are connected linearly (without any dynamic routing). The connection, however, does not need to be to the nearest neighboring PEs.

2. Array: PEs are placed and connected in a regular manner.

3. Mesh: Array architectures can be further classified as a mesh architecture when the neighboring PEs are grouped in "super-blocks" to reduce the routing density. In this architecture, high-density routing is maintained intra-super-blocks while reduced inter-super-blocks.

4. Crossbar: Architectures that were classified as mesh can be classified as crossbar when extra (dynamic) routing resources are available between the super-blocks.

5. Datapath: An architecture is said to be a datapath when the routing of data is controlled at the bus-level rather than at the bit-level. This is typically for course-grained architectures such as x-bit processors, where the routing is controlled at the x-bit level.

2.2.2 Interconnect

The interconnect of an architecture describes the mechanism of the data flow. Depending on the system granularity, hardware interconnects can be as simple as cell-to-cell interconnects

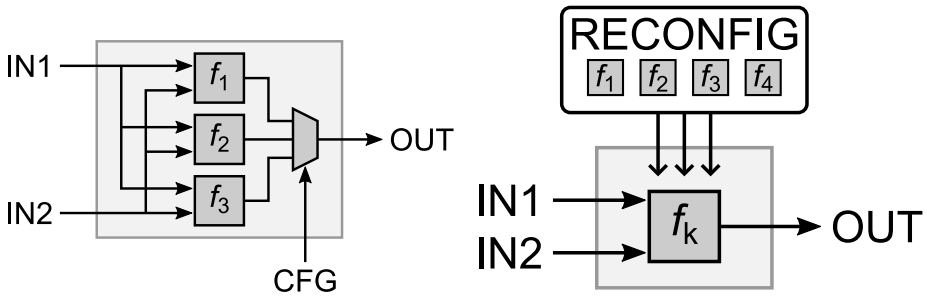
(for fine-grained logic) or as complex as a network-on-chip (for coarse-grained logic), where data are sent as network packets.

2.2.3 Fabric Structure

In terms of *homogeneity*, an architecture is classified as homogeneous when all the configurable blocks are identical in function and arranged in a regular manner, or it is classified as heterogeneous when specialized blocks exist. Conversely, the fabric structure can also be defined in terms of *granularity*: fine-grained (e.g., array of transistors or logic gates), medium-grained (e.g., array of basic processing units such as adder, subtractor, or multiplexer) or coarse-grained (e.g., array of DSP cores or processors).

2.2.4 Reconfiguration Schemes

In terms of reconfigurability, the hardware core can be reconfigured in one of two methods: virtual reconfiguration circuit (VRC) or native reconfiguration (often called dynamic partial reconfiguration, DPR), as shown in figure 2.2; additionally, a comparison is presented in table 2.1.



(a) VRC **(b)** DPR

Figure 2.2: VRC and DPR: the two reconfiguration schemes for EHW [3].

Virtual Reconfiguration Circuit (VRC)

Virtual reconfiguration circuit is a method where the evolutionary algorithm switches between a set of existing functions that are physically implemented in hardware. This method is fast since the delay only depends on the time consumed by switching between functions, but it is not space or power efficient. Moreover, in some applications, VRC may result in lowering the maximum operational frequency [84]. Most of the early EHW systems were VRC-based for three reasons: (1) VRC is a simple method to be implemented. (2) VRC works well for fine-grained functions. (3) Until recently, the technology did not support any other way of reconfiguration.

Dynamic Partial Reconfiguration (DPR)

Dynamic partial reconfiguration is the method of reconfiguring or reprogramming a dynamic region of an FPGA fabric using a bitstream from a library of pre-compiled functions. Because the dynamic region is relatively large, DPR is fairly slow for the required speed of most real-world applications. However, DPR is power and space efficient, and it may be the only practical choice for applications with coarse-grained functions. Initially, DPR was feasible using low-level bitstream manipulation methods [85, 56] enabled by open architectures, bitstream reverse engineering and/or some open-source application programming interfaces (APIs) such as TORC [86] and RapidSmith [87]. A low-level bitstream manipulation method can be unsafe and complicated, particularly on recent FPGAs [45]. Currently, major FPGA vendors support native run-time reconfiguration, but with some limitations, such as complex design flow and unsupported bitstream relocation. Despite these limitations, many successful EHW implementations have been proposed [23, 88, 3, 89, 90, 91, 60].

Table 2.1: Comparison of VRC and DPR.

Parameter	VRC	DPR
Reconfiguration speed	Fast ($\sim$GHz)	Slow, depending on the bitstream size ($<$10 KHz)
Fabric utilization	Inefficient	Efficient
Power consumption	All functions are ON.	One function is ON. Some transient power consumption on reconfiguration.
Max. operational frequency	Good (with limitation)	Best
Requirements	None	Reconfiguration device, port and memory for storing bitstreams
Complexity	Simple	Complicated but many efforts to streamline it
Best for	Applications with fine-grained functions	Applications with medium- to coarse-grained functions

2.3 Reconfigurable Architectures

Reconfigurable architectures can be classified into two categories: commercial and custom architectures, as shown in figure 2.3. Commercial programmable architectures are discussed first. Then, custom reconfigurable architectures are discussed, where some EHWs utilize FPGAs to realize the final system architecture.

2.3.1 Commercial Simple Programmable Logic Devices

After the birth of PROMs and because of the high demand for compact and flexible "glue logic" architectures, a new family of devices were created. Programmable logic devices (PLDs) come in many forms, as shown in figure 2.3, but they all serve one purpose. PLDs are programmed (and reprogrammed) by the user to realize a digital circuit.

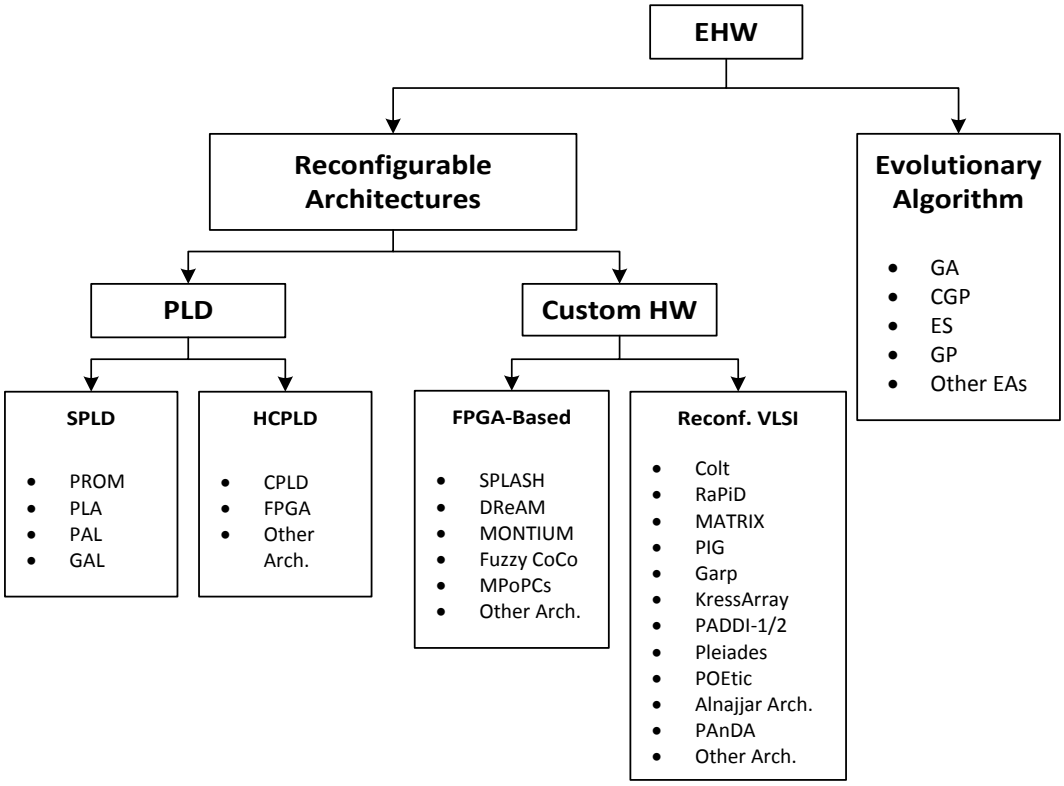


Figure 2.3: Classifications of reconfigurable architectures.

Simple Programmable Logic Devices

Simple programmable logic devices (SPLDs) are the simplest reconfigurable arrays with a relatively low amount of simple logic (< 1000 gates). These devices contain a set of fully connected macrocells, where each macrocell contains a mix of simple gates and flip-flops, which are sufficient to realize basic functions in the product-of-sums (or sum-of-products) canonical form. The SPLD consists of three main blocks: the input block, AND plane, and OR plane, as shown in figure 2.4.

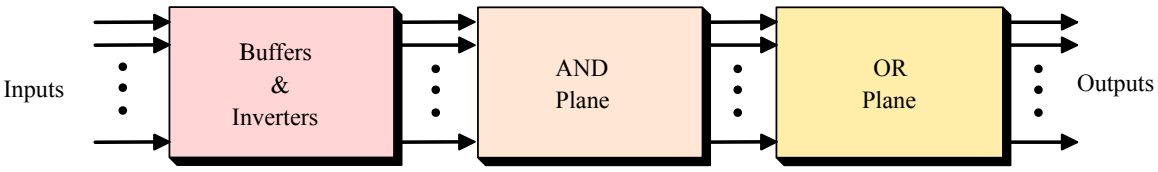


Figure 2.4: Simplified SPLD, adapted from [4].

Programmable Read-Only Memory (PROM)

The simplest form of programmable devices is PROM, which is a block of programmable memory that stores truth table data of the intended logical design. The input to the PROM works as an address to the memory, and the stored data serves as the output. An example of a PROM is shown in figure 2.5. For PROMs, the user can only program the content of the memory and is unable to change the input or output routing. Due to the discussed limitations, flexibility and scalability are the major drawbacks of PROMs.

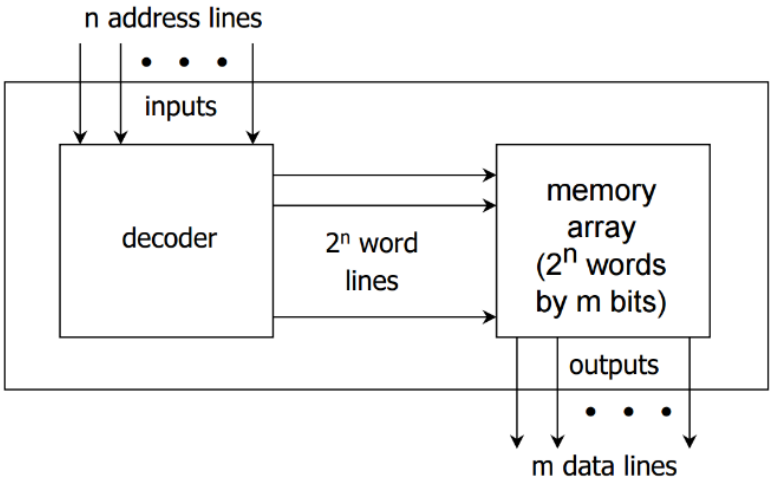


Figure 2.5: PROM: the simplest programmable architecture [5].

Programmable Logic Arrays (PLAs)

Programmable logic arrays are another type of SPLD. PLAs were introduced in the early 1970s by Texas Instruments based on read-only associative memory (ROAM) to solve the drawbacks from which PROMs suffered. The main feature of PLAs is that both of their planes (i.e., AND plane and OR plane) are programmable, as shown in figure 2.6. The architecture of the input interconnect allows for more flexibility on the input ports. Since the two programmable links were slow OR gates, PLAs were slower than PROMs and did not gain popularity.

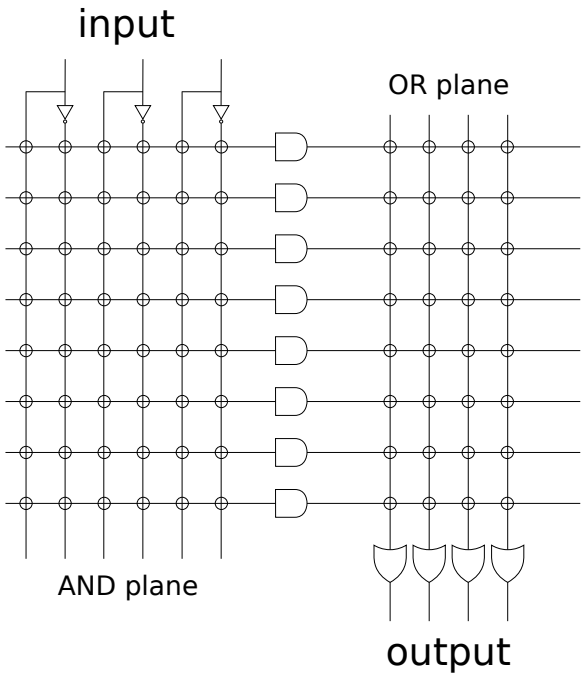


Figure 2.6: PLA has a programmable AND plane and a programmable OR plane.

Programmable Array Logic (PAL)

Programmable array logic was introduced in 1978 by Monolithic Memories Inc. [92]. In this architecture, the output interconnect was hardwired and the user could only program

the input connection matrix, as shown in figure 2.7. Since there is a single (AND gate) link in the PALs compared with two (OR gate) links in the PLAs, PALs were faster than PLAs. However, PALs were less reprogrammable than PLAs since they had AND-based programmable links. Although a hardwired output indicates a flexibility limitation on what logical equations the circuit can represent in comparison with the former PLA devices, the additional programmable loopback interconnect in PALs improved their flexibility because it allowed realization of multi-level canonical forms.

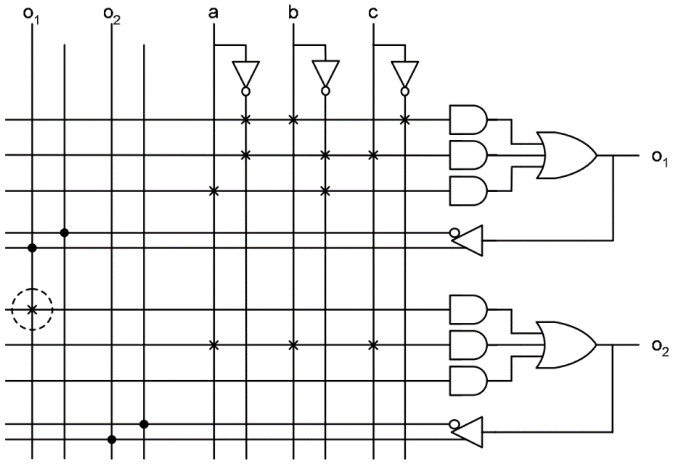


Figure 2.7: PAL architecture with loopback wiring to improve flexibility [6].

Generic Array Logic (GAL)

Generic array logic, introduced by Lattice Semiconductor in 1985, was the next generation of PALs. The main features showcased were the integration of CMOS technology and improved reconfigurability, where the device can be reconfigured many times using a programmer or in-circuit programming techniques.

2.3.2 Commercial High-Capacity Programmable Logic Devices

Another branch of PLDs consists of high-capacity programmable logic devices (HCPLDs). HCPLDs include complex programmable logic devices (CPLDs), FPGAs, and other commercial programmable architectures.

Complex Programmable Logic Devices (CPLDs)

A CPLD is a high-density programmable logic device that is more complex (larger) than PALs but less complex than FPGAs. In contrast to FPGAs, CPLDs have a non-volatile EEPROM-based configuration memory that makes them fast to boot but slow to be re-programmed. Due to their simpler architecture, CPLDs have low pin-to-pin delays. A simplified high-level block diagram for a CPLD is presented in figure 2.8. Macrocells are the building blocks of CPLDs, which are relatively larger than the building blocks of FPGAs. However, a CPLD typically has less than a few hundred macrocells versus more than ten thousand building blocks for FPGAs. Figure 2.9 shows MAX V, a CPLD manufactured

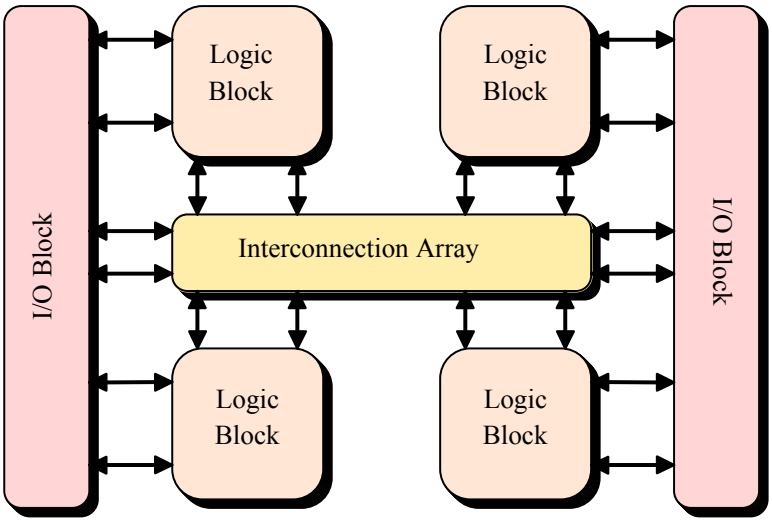


Figure 2.8: A simplified block diagram of CPLD architecture [4].

by Altera. This device (5M40Z specifically) has 24 logic array blocks (LABs) stacked in a 6x4 array with a MultiTrack interconnect in-between. Each LAB consists of 10 logic elements (LEs). The structure of an LE is shown in figure 2.10. The typical equivalent macrocells for 5M40Z is 32 macrocells, indicating that an LAB in this device is larger than a typical macrocell.

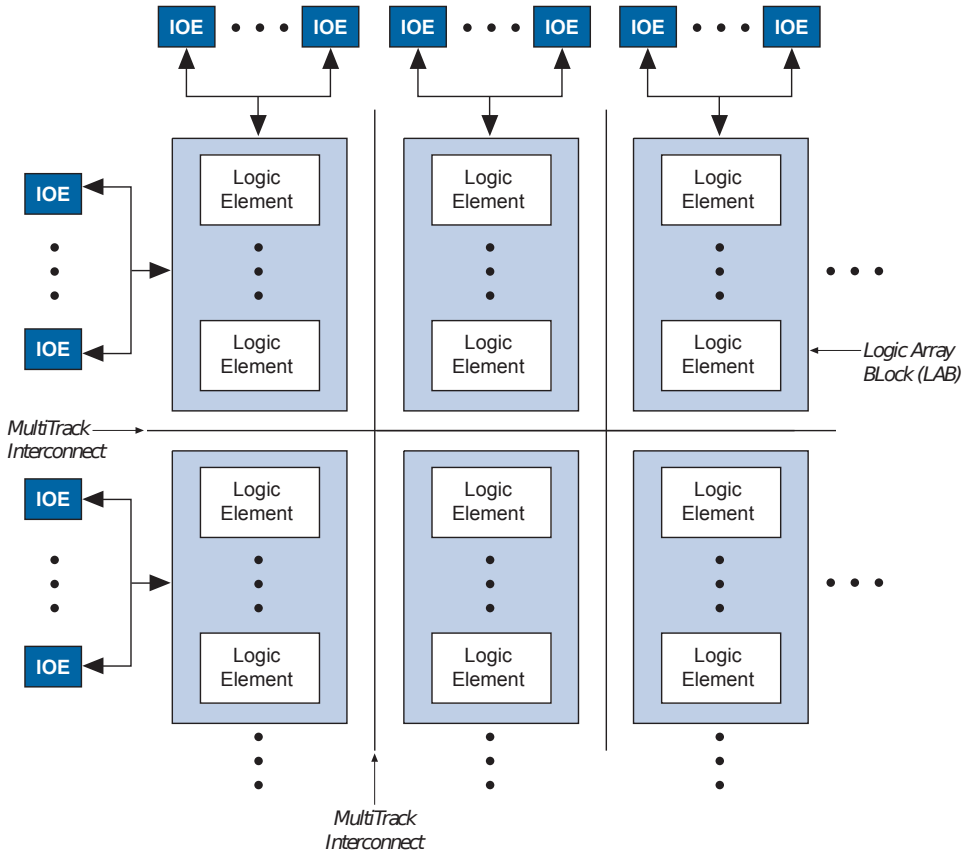


Figure 2.9: MAX V: a CPLD manufactured by Altera [7].

Field-Programmable Gate Arrays (FPGAs)

FPGAs are integrated circuits designed to be configured by customers in the field. An FPGA is an array of configurable logic blocks (CLBs), block RAM memory (BRAM), dig-

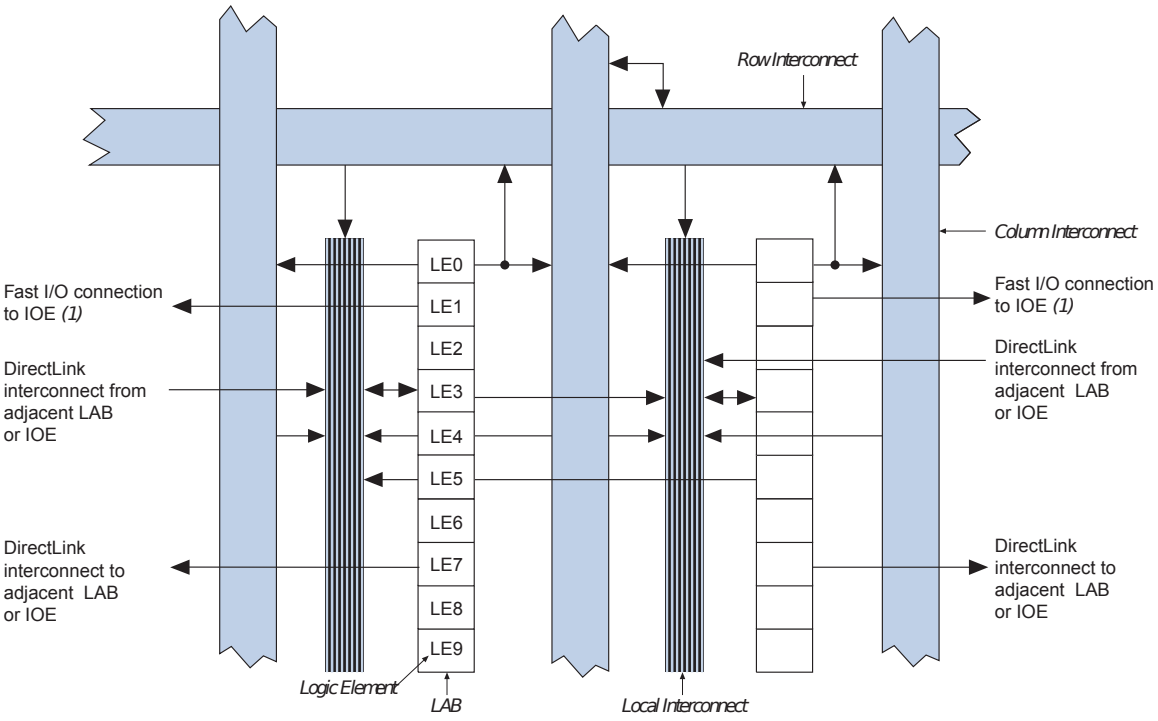


Figure 2.10: LABs are the building blocks of CPLDs. Each LAB has 10 LEs [7].

ital signal processor blocks (DSPs), and other hard-cores (occasionally hard-IP cores). The programmable blocks are arranged in columns with complex intermediate interconnects. The programmable array is surrounded by programmable input/output blocks (IOBs). A simplified FPGA layout is shown in figure 2.11.

CLBs are the main building blocks of FPGAs. In the case of the Xilinx 7 series, a CLB, as shown in figure 2.12, consists of two slices, where each slice has four 6-input look-up tables (LUTs), eight flip-flops, multiplexers, and arithmetic carry logic. The arrangement of programmable blocks for this FPGA is called the ASMBL (advanced silicon modular block) architecture, as shown in figure 2.13.

Most of the FPGAs' configuration memory is SRAM-based. The configuration of an FPGA is performed using a bitstream that is initially generated from a design modeled

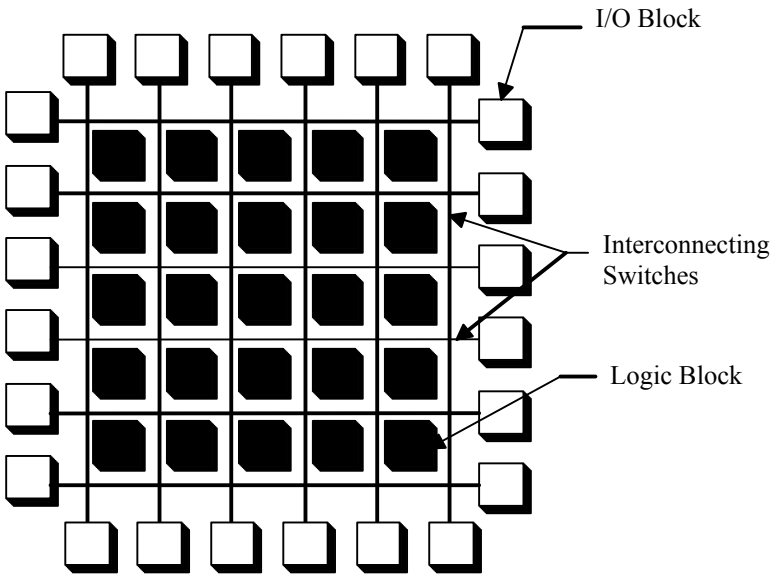


Figure 2.11: Simplified FPGA block diagram [4].

using an HDL, such as VHSIC (very high speed integrated circuit) hardware description language (VHDL) or Verilog using electronic design automation (EDA) tools provided by the FPGA vendor. This process includes many subprocesses, such as elaboration, synthesis, placement, routing, implementation, bitstream generation, and optionally simulation.

There are two types of bitstreams: *full-chip bitstream* and *partial bitstream*. The full-chip bitstream is for configuring the entire chip, where the device functionality is interrupted while it is being programmed. Conversely, partial bitstream is where only a subarray of the logic (called dynamic partition) is reprogrammed without interrupting the operation of the remaining part of the array. The size of the partial bitstream depends on the size of the dynamic partition, which determines the speed of reconfiguration. Xilinx 7 series devices (specifically Zynq-7000) support several ways to configure the programmable logic, such as using the processor configuration access port (PCAP), joint test action group (JTAG), or internal configuration access port (ICAP). PCAP uses the device configuration interface

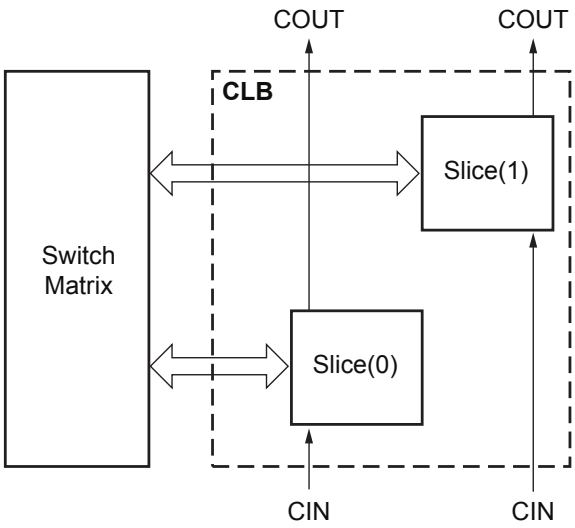


Figure 2.12: Two slices per CLB, Xilinx 7 series [8].

module (DevC), which has an embedded direct memory access (DMA) controller capable of initiating data transfers from the external memory to the fabric configuration memory. Therefore, PCAP does not need any hardware instantiations on the programmable logic. The maximum configuration speed achieved by PCAP is 128 MB/sec. ICAP, in contrast, requires instantiating a Hardware ICAP (HWICAP) module in the programmable logic. Although the theoretical maximum speed of ICAP is 400 MB/s, the maximum achievable speed is 67 MB/sec using conventional DMA-dependent transactions. Vipin et al. proposed an efficient management system, ZyCAP, that increases ICAP performance to 382 MB/sec [93].

Other Commercial Reconfigurable Platforms

There are many commercial reconfigurable architectures. One example is D-Fabrix manufactured by Panasonic, which is a low-power ASIC aimed at embedded multimedia applications [94]. The device is an array of homogeneous word-based processing elements based

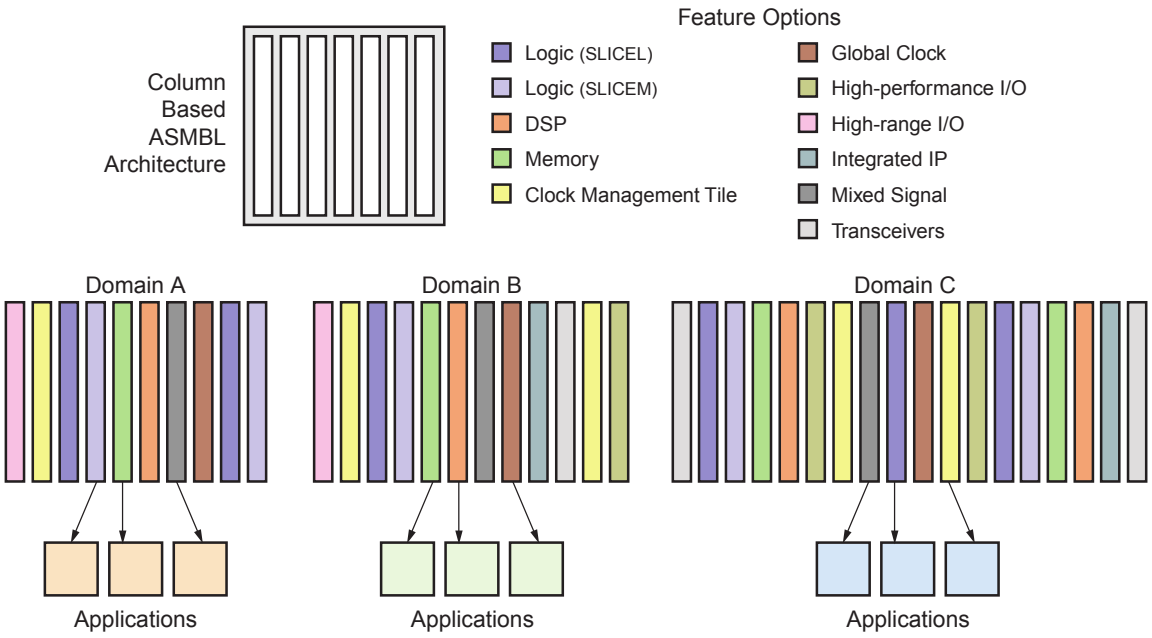


Figure 2.13: Variety of programmable logic blocks are arranged in a column-style, ASMBL architecture [8].

on the CHESS reconfigurable platform developed by Hewlett Packard Labs [95]. The device specifications are not disclosed. Another example of commercial reconfigurable architectures is the PACT XPP-III architecture used in low-power irregular control-flow-dominated streaming algorithms [13]. The XPP-III architecture is based on a hierarchical array of sequential coarse-grained processors optimized to run in different types of parallelism [96]. QuickSilver Adapt2400 [97], Coherent Logix HyperX [98], and Adapteva Parallella [99] are examples of other common commercial reconfigurable architectures.

2.3.3 Custom Architectures

Custom architectures include two branches. The first branch is for FPGA-based architectures, but here it does not mean using the FPGA fabric (e.g., LUTs or CLBs) as the building block for the reconfigurable core but rather using it to construct a higher-level

reconfigurable architecture. Thus, in this sense, the system can be realized in any platform, including VLSI. The second branch is custom-hardware architectures. These systems are fabricated on silicon.

FPGA-Based Reconfigurable Architectures

FPGAs are strong candidates as hosts for EHWs due to many reasons enabled by the advancements made in this technology. Some of these reasons are (1) high-speed performance, (2) high reliability (compared to non-commercial VLSI), (3) low cost, (4) high-speed reconfiguration and native support for dynamic partial reconfiguration [100], and including high-performing hard-IP cores (such as processors) [101].

In fact, considerable research has been performed that simply takes advantage of the device features to improve the performance and integration of EHW, such as the CoPR framework implemented on Zynq, which isolates the designer from the low-level architecture by using a high-level API [102]; implementing Linux accelerators on Zynq [103]; and ZyCAP, which increases the reconfiguration throughput [93]. In this section, a brief overview of some of the FPGA-based reconfigurable architectures is presented.

SPLASH, proposed by Gokhale et al., is one of the early FPGA-based platforms designed for DNA sequence matching [104].

DReAM is a dynamically reconfigurable hardware architecture for mobile communication systems proposed by Becker et al. [105]. The system is composed of a coarse-grained array of reconfigurable processing units and configuration controllers. The reconfigurable processing units are connected to the nearest neighbor, are capable of performing high-level arithmetic operations and include a complex correlation unit needed for communications applications. The configuration controllers can perform a local reconfiguration based on

local configuration memory without the need for external memory. DReAM was implemented on an FPGA and uses a CAD tool to create and customize bitstreams.

The *MONTIUM* architecture, proposed by Heysters et al., is an energy-efficient, flexible, coarse-grained array of tiles designed for high-performance applications [9]. As shown in figure 2.14, each tile has a set of ALUs connected to a direct datapath. Each ALU has a local memory used to increase parallel processing throughput. This system was successfully configured for fast Fourier transform (FFT) and finite impulse response (FIR) filtering.

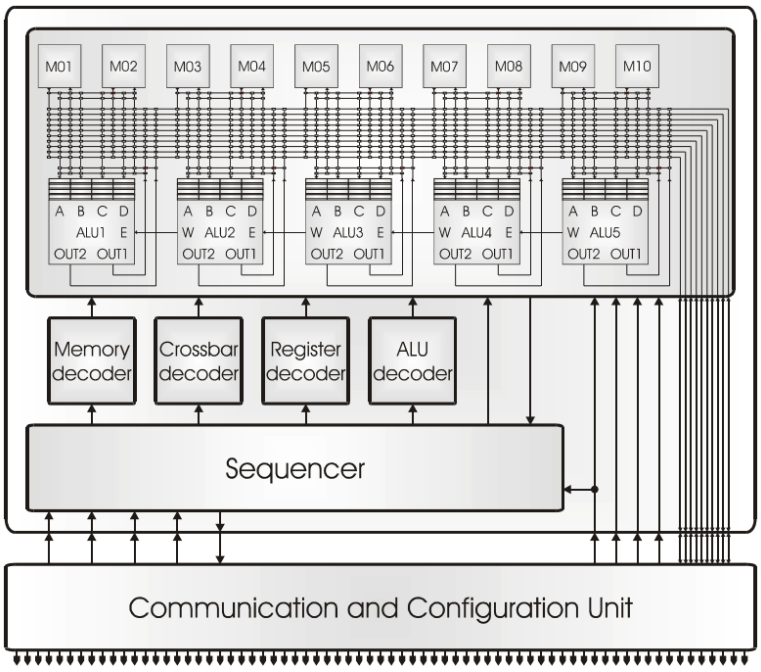


Figure 2.14: Block diagram of MONTIUM tile [9].

Fuzzy CoCo is a reconfigurable system based on fuzzy logic used for general computation applications, and it was proposed by Mermoud et al. [10]. FPGAs were used due to their flexibility for the testing of modular layers by dynamically using an adaptation mech-

anism to tune system parameters. The system has three layers of operation: fuzzification, rule-based inference, and defuzzification, as shown in figure 2.15.

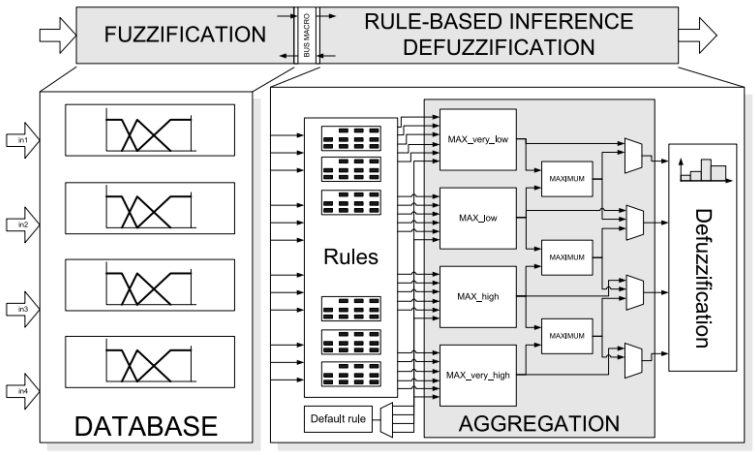


Figure 2.15: Reconfigurable architecture based on fuzzy logic, Fuzzy CoCo [10].

The *MPoPCs* architecture is an FPGA-based reconfigurable system with heterogeneous coarse-grained processing elements proposed by Wang et al. [106]. The system consists of IP-based processing systems running in MIMD (multiple instruction, multiple data) mode, customizable memory and an interconnection network. The system is designed for massively parallel operations, such as matrix operations. The scheduling process for run-time load-balancing is made possible by software techniques.

Custom-Hardware Reconfigurable Architectures

Over the past 30 years, many custom reconfigurable platforms have been proposed. In this section, several common platforms will be discussed.

The *Colt* architecture, presented by Bittner et al., consists of an array of functional units and data ports connected via a smart crossbar switch, as shown in figure 2.16 [11]. The system utilizes the wormhole run-time reconfiguration computing paradigm to allow

operation while reconfiguring; thus, the system targets online partial reconfiguration. A major disadvantage of this platform is that there is a lack of hardware mapping tools available to designers.

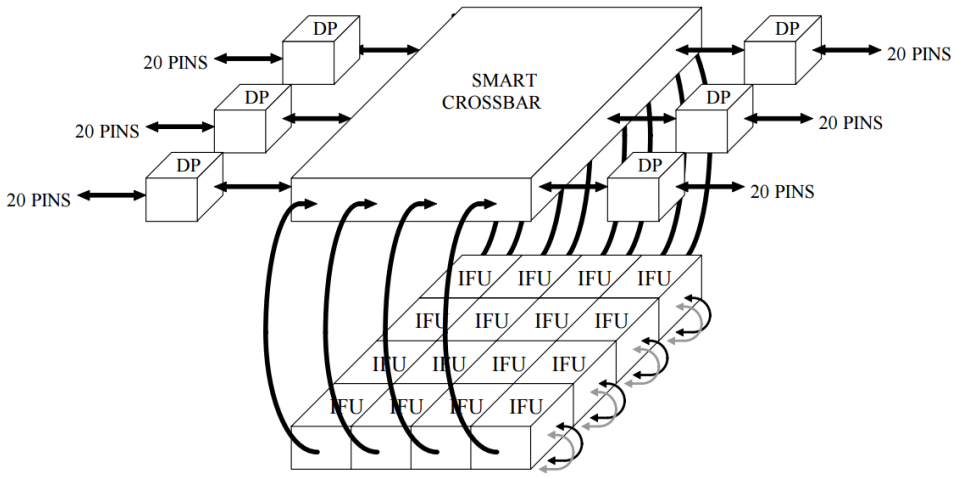


Figure 2.16: Colt reconfigurable architecture with 16 functional units, smart crossbar interconnect and 6 data ports [11].

The *RaPiD* system, proposed by Ebeling et al., is a resource-efficient coarse-grained FPGA-like architecture optimized for performance that supports deep application-specific pipelines [107]. The system is designed for computationally intensive applications and uses a mixture of static reconfiguration and dynamic control. Static reconfiguration is used to program the underlying pipeline datapath, and dynamic control is used to schedule the pipelined operations.

A common approach for developing reconfigurable platforms is to have a special configuration layer, similar to that in *MATRIX* as proposed by Mirsky et al. [108], which is a coarse-grained platform that enabled applications to control resources using a multi-level configuration scheme by configurable instruction distribution. MATRIX building blocks can be configured to serve as instructions store, memory elements or computational ele-

ments. The platform also features configurable datapaths, but their effect was not studied. A drawback of this system is the lack of mapping tools, as mapping is performed manually. PIG is a general-purpose massively parallel fine-grained reconfigurable system that contains a two-layered (data and configuration) 2D grid of cells [109]. These cells are capable of reconfiguring other cells to scale the array dynamically and autonomously.

Garp is another reconfigurable platform proposed by Hauser et al. [12], which is a fine-grained architecture capable of complex bit-oriented computations for image processing applications. The programmable grid works as a co-processor for an in-chip MIPS II processor. The programmable array organization is shown in figure 2.17, where an operation can be mapped to a burst of logical blocks.

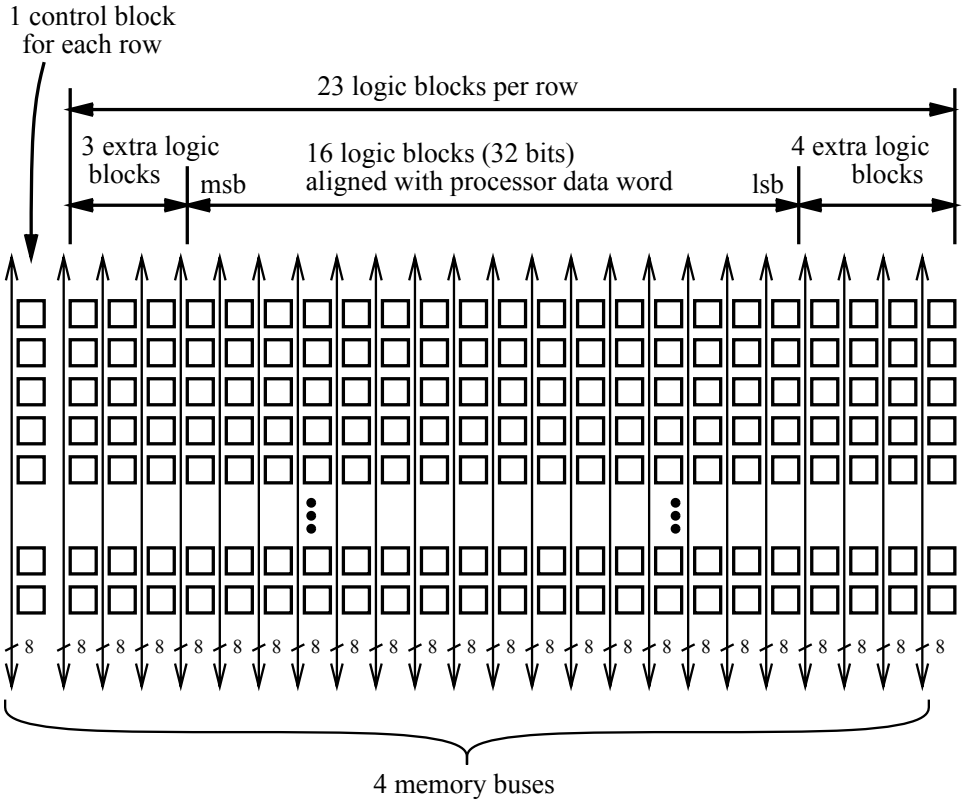


Figure 2.17: Garp reconfigurable architecture [12].

Other reconfigurable platforms use nontraditional architectures, such as that for *KressArray* proposed by Kress et al., which is a super systolic array with a nonlinear and wide reconfigurable datapath array (rDPA, shown in figure 2.18) to reduce communication resources [110]. Another feature in this architecture is the globally scheduled high-level serial bus. Several platforms were constructed based on this non-von-Neumann architecture, such as the KressArray Xplorer CAD [111] and MoM-PDA [112].

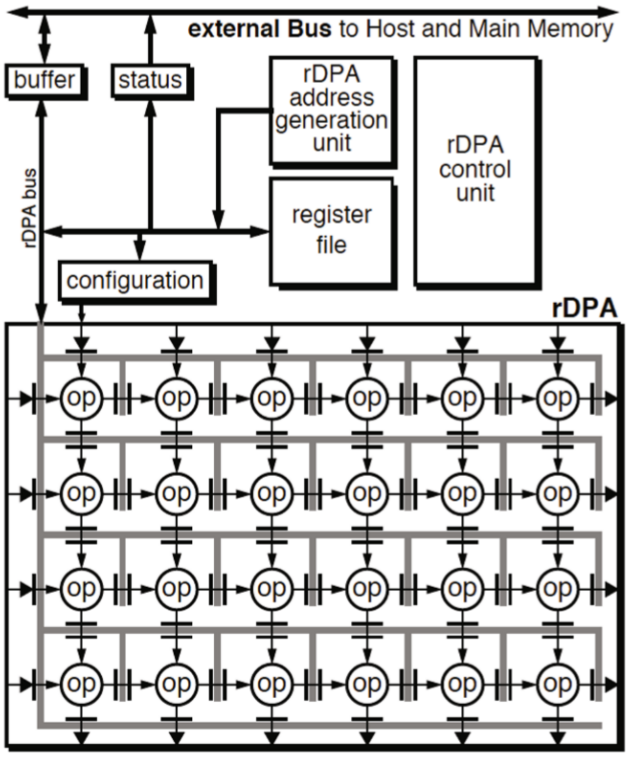


Figure 2.18: KressArray: a non-von-Neumann reconfigurable architecture [13].

PADDI is a family of three architectures. These are arrays of processing elements with a crossbar interconnect. PADDI-1 is an array of homogeneous fine-grained nano-processors reconfigured by software [113]. PADDI-2 presents a data-driven execution feature and improved processing elements built on an array that is still fine-grained and homogeneous

[65]. Finally, the *Pleiades* architecture, shown in figure 2.19, is a reusable architecture platform with a heterogeneous network of coarse-grained processing elements that can be programmed for a selected application domain [14].

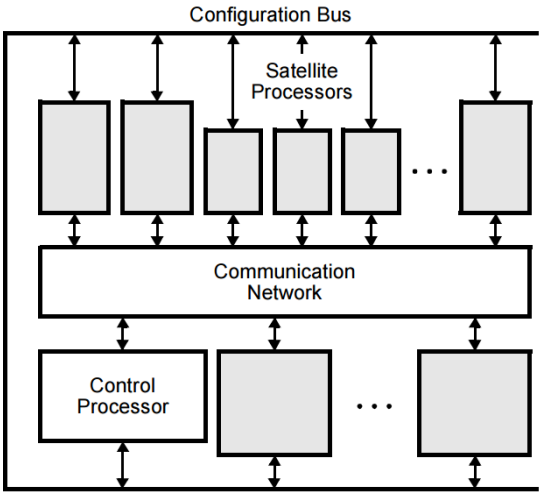


Figure 2.19: Pleiades: a heterogeneous coarse-grained reconfigurable platform [14].

Some EHW platforms are bio-inspired architectures, such as *POEtic* tissue, which contained three layers (phenotype, mapping, and genotype) to support three processes (evolution, development, and learning) [15], as shown in figure 2.20.

Alnajjar et al. proposed a coarse-grained dynamically reconfigurable architecture with flexible reliability [114]. The system consists of an array of clusters, where a cluster can select four operation modes with different levels of spatial redundancy. By utilizing redundancy, the system is designed to tolerate soft errors and device aging. The level of redundancy (i.e., error detection or error correction) can be selected by the user based on the application reliability constraints.

The *PAnDA* project, proposed by Walker et al., is a programmable analog and digital array with bio-inspired techniques that enable reconfiguration of the analog layer to over-

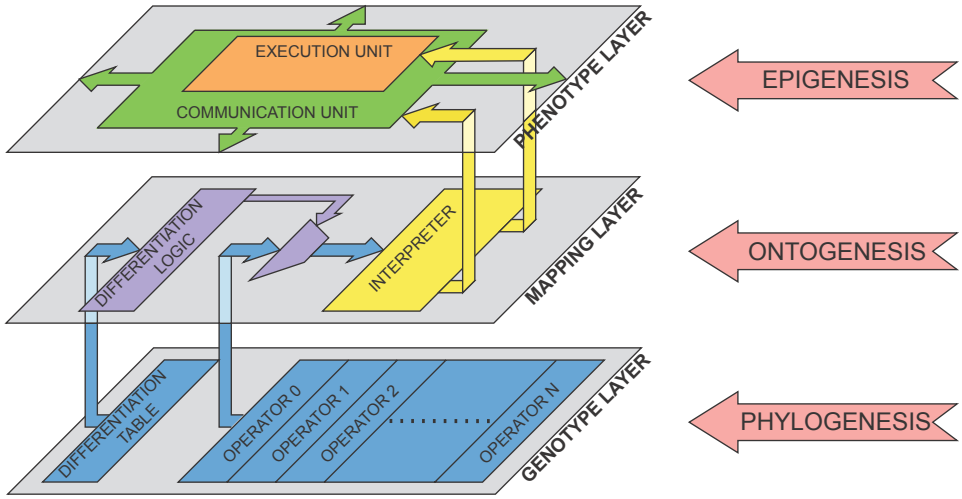


Figure 2.20: POEtic: a reconfigurable bio-inspired architecture [15].

come process variations [115].

2.4 Summary

The body of an EHW system is the reconfigurable hardware core. Early EHW implementations were PLD based, but their capabilities were limited. The next wave of systems were a mixture of custom VLSI architectures and FPGA-based architectures, where the fabric of the FPGA was used as the building block. The majority of these systems had a VRC-based reconfiguration scheme.

Subsequently, many novel systems were proposed with more emphasis on the DPR reconfiguration scheme, which was motivated by technology advancements. These systems were a mixture of custom VLSI architectures and FPGA-based architectures, where FPGAs were used to realize a higher-level reconfigurable system.

Currently, FPGA-based systems are the common trend of EHW systems. This is due to their advantages over custom VLSI, including low cost, short design time, fast run-time

reconfiguration, high reliability, and soft/hard-IP core processors. Additionally, FPGAs are considered to be the ideal platforms for hosting evolutionary algorithms while interfacing with the outside world.

CHAPTER 3

EVOLUTIONARY ALGORITHMS

3.1 Introduction

Through an iterative process, from generation to generation, the genetic operations are applied to a subset of selected solutions to produce increasingly fitter "offspring" solutions. The fittest of these individuals, or solutions, survive to the next generation. A general workflow of EAs is presented in figure 3.1. In brief, an EA requires recipes (which are the genetic operators discussed in section 3.2) to produce new solutions and a gauge (which are the fitness functions discussed in section 3.3) to measure the quality of the solutions.

3.2 Genetic Operators

Genetic operators are used to guide EAs toward generating fitter genomes. EAs depend on these operators for selecting solutions, maintaining genetic diversity, and recombining existing solutions. The main four genetic operators are selection, mutation, crossover, and elitism.

3.2.1 Selection

Selection is an operator for selecting the best individuals of a generation to pass to the next generation. The selection can be performed using roulette wheel selection [116], stochastic

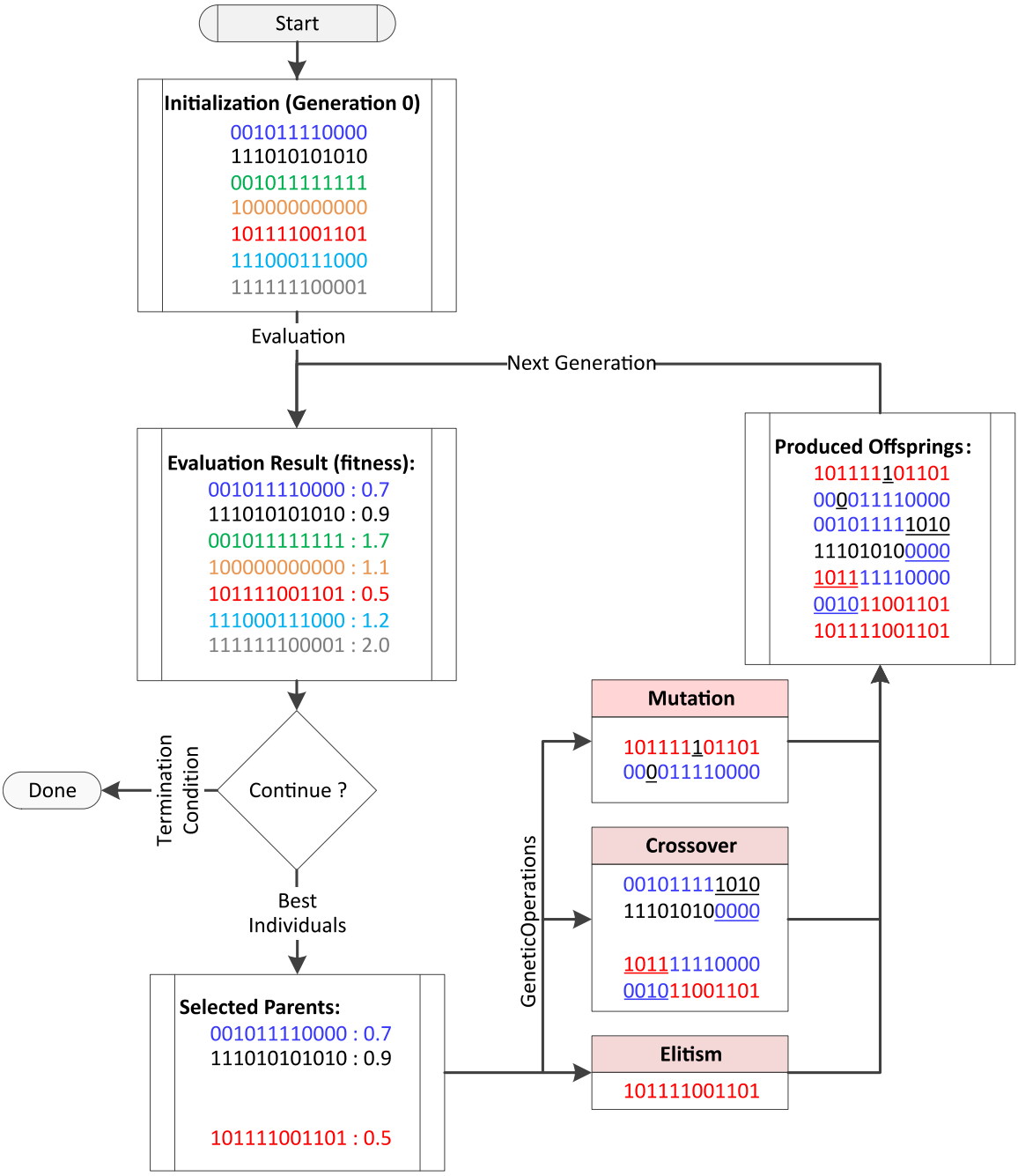


Figure 3.1: A general workflow for evolutionary algorithms. The closer the fitness is to zero, the better the solution is.

remainder [117], rank selection, tournament selection or Genitor selection [118].

3.2.2 Mutation

Mutation is an operator that changes one or more randomly chosen genes on a selected genome to produce a new offspring. In its simplest form, mutation is flipping a bit in a genome represented by a string of bits. Some of the mutation techniques are mutation with adaptive probability [119], mutation with optimal rates [120], and mutation with hill-climbing strategy [121].

3.2.3 Crossover

Crossover (also called recombination) is an operator that is analogous to natural reproduction, in which a new offspring is produced by recombining chromosomes from two or more parents [122]. Several recombination techniques exist, including single-point crossover, two-point crossover, three-parent crossover [123], uniform crossover [124], and adaptive crossover [125].

3.2.4 Elitism

Elitism is an operator that originates in response to the potential concern of losing good genomes in the evolution process. In other words, genetic operators may damage a selected genome and prevent it from raising to the next generation. Elitism guarantees that a genome will not be discarded except in a case where a better genome exists [126]. In some literature, elitism is considered to be a selection operator.

3.3 Fitness Functions

Evolutionary algorithms are used by scientists to solve non-trivial problems. This means that EA will hopefully provide the final design without human intervention. However, that does not mean that human designers do not become involved. An EA requires a fitness function (also called an objective function) [127], which is supposed to summarize the performance of an individual in a single figure of merit (score). It simply describes how close a solution is from achieving a defined goal. One of the top challenges in designing EAs is the design of a fitness function. Poorly designed fitness functions may cause the system to produce inappropriate solutions or not produce solutions.

For a successful EA, a successful fitness function must be designed. The outline for a successful fitness function is as follows:

1. Accurate but simple: The fitness function must describe a solution behavior without involving low-level specifications.

2. High-speed computation: The computations needed for calculating the fitness value must not slow the system. In fact, for an intrinsic evolution, the fitness function must be hardware implementable and as fast as the system running in hardware.

3. Comprehensive: The fitness function must capture all design objectives, such as reducing power consumption and resource utilization. A challenge that arises when combining multi-objective functions is the mechanism of assigning weight to them.

Because speed and accuracy are generally working against each other, a trade-off needs to be made by the fitness function designers or another approach should be used – fitness approximation. Since many of the evolutionary systems are not searching for optimal solutions and are running in a noisy environment, fitness approximation can be utilized. Fitness

approximation can be achieved by many techniques, such as assuming that individuals that behave similarly have similar fitness [128], using local approximation of difference evaluation functions [129], or fitness landscape approximation with a Fourier transform [130].

Fitness approximation cannot be used in applications where finding solutions with complete correctness is desired, e.g., designing a multiplier; bitwise fitness functions are often used in this case [131].

Finally, fitness values can be absolute or relative. Although relative values are desired, the implementation implicates challenges, such as an increase in the required computations and the prior knowledge of the fitness of a "perfect" solution, which will be used as a reference. The fitness used in the previous example in figure 3.1 was relative and normalized to 1, where 1 is given to solutions with no improvements and 0 for the best possible solution. In other words, the closer the fitness is to zero, the better the solution is.

3.4 Evolutionary Algorithm Types

Commonly used EAs are the genetic algorithm [132], evolutionary strategies [133], genetic programming [134], Cartesian genetic programming [16], differential evolution [135], neuroevolution [136], and learning classifier system [137]. This work focuses on the major EAs, which are the first four algorithms and are discussed in this section. The genetic algorithm is the most commonly used EA and the best fit for EHW [13], as discussed in section 3.5.

Evolutionary strategy (ES) was developed in parallel to genetic algorithms, but with more focus on optimization problems, specifically (floating-point) parameter optimization. Algorithm 1 provides a simplified $(1+\lambda)$ ES pseudo code, where 1 is the number of parent

genomes and λ is the number of the generated child genomes (offspring). Offspring are generated using mutation alone; in other words, ES does not use crossover operations. Mutating one parent to generate λ genomes in multiple generations can result in a hill-climbing search. However, this may reduce the search space and cause evolution to be stuck in a local minimum, which affects the quality of the generated results. Note that the mutation operator in this context is not simply by flipping random bits but uses further sophisticated mechanisms, such as adding randomly distributed numbers [138]. Another common version of this algorithm is the $(\mu + \lambda)$ ES, where μ parent genomes are used (rather than 1).

Algorithm 1: A simplified $(1 + \lambda)$ ES algorithm, assuming that smaller fitness is better.

```
1  g_parent = null
2  fitness(g_parent) = ∞
3  generation = 0
4  while not termination condition do
5  |   g_child = g_parent
6  |   for i ← 0 to 1 + λ do
7  |   |   if generation=0 then
8  |   |   |   generate g_i by random selection
9  |   |   else
10 |   |   |   generate g_i by mutation of g_parent
11 |   |   end
12 |   |   if fitness(g_i) ≤ fitness(g_child) then
13 |   |   |   g_child = g_i
14 |   |   end
15 |   end
16 |   g_parent = g_child
17 |   generation++
18 end
```

Genetic programming (GP) was introduced by John Koza in 1992 [134]. GP is a form of autonomous computer program evolution that is performed by genetically modifying a population of computer programs using natural bio-inspired concepts. The main characteristic of this algorithm is the representation of individuals, where the genotype is a parse tree and the phenotype is a computer program, as shown in figure 3.2. GP is a domain-independent method, but the functions (called functions set and terminals set) and fitness function are

domain dependent and need to be defined per the application. Similar to other algorithms, evolution using this algorithm is an iterative process that includes a variety of genetic operators, such as crossover, mutation, reproduction, gene duplication and gene deletion. GP uses some unique genetic operators that are not used in other algorithms; gene duplication is an illegitimate crossover process that results in *longer* genomes, whereas gene deletion is the complementary process that results in *shorter* genomes. Therefore, implementing this variable-length phenotype in hardware is a challenge. In general, GP exploits the flexibility of software, which cannot be offered by hardware. Another major disadvantage of GP is the scalability because the algorithm performs poorly on complex problems [139].

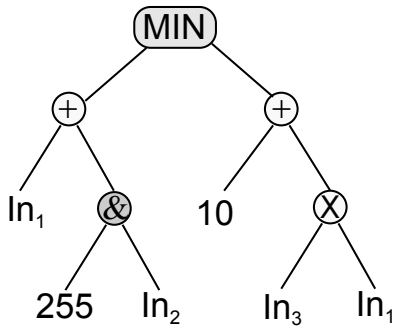


Figure 3.2: GP represents genomes as parse trees. Tree nodes are mapped to computer programs. The shown tree is equivalent to the program `MIN(`In_1`+(`In_2` & 255), 10+(`$In_3 \times In_1$`))`.

Cartesian genetic programming (CGP) was introduced by JF Miller to solve the scalability drawback of genetic programming [140]. CGP is simply a general form of genetic programming. It is called Cartesian because it represents programs in 2D grids of nodes, as shown in 3.3. Programs are described as directed acyclic graphs. In its simplest form, CGP uses a vector of integers to represent program primitives along with the routing interconnects. Although CGP was used efficiently in many computational application domains

such as circuit design, neural networks, mathematical equations, computer programs, and image processing, CGP has many drawbacks, as follows:

1. It requires a large logic space when implemented in hardware. In other words, CGP utilizes resources inefficiently.

2. Practical problems yield a large solution space, which results in a high demand for computational efforts and a low rate of convergence.

3. The reconfiguration scheme is limited to VRC.

New algorithms of CGP are self-modifying CGP (SMCGP) [141], modular CGP [142], and recurrent CGP [143].

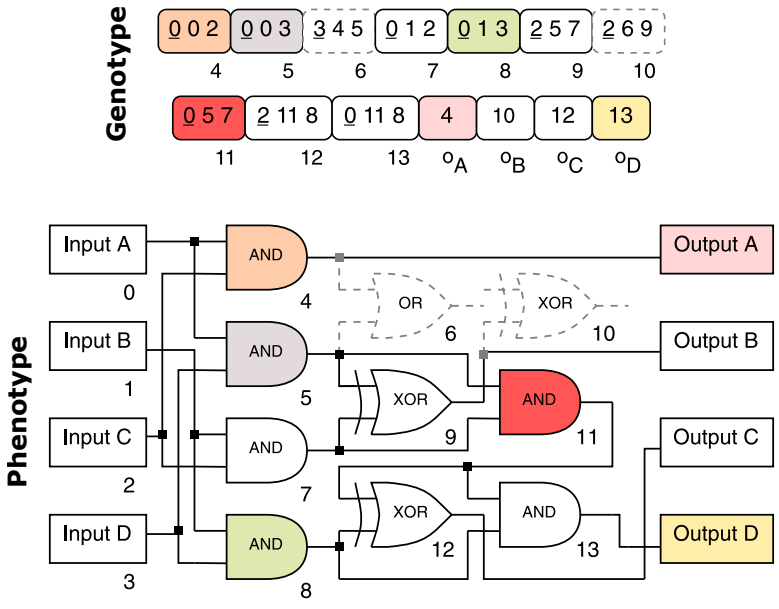


Figure 3.3: Example of two-bit multiplier circuit evolved using CGP by Miller et al. [16]. Each integer in the genotype defines a function selection or a routing option. Some chromosomes were left unused in this example.

3.5 Genetic Algorithm

John Holland invented the software implementation of genetic algorithms (GAs) in the early 1970s [132]. GA is an adaptive bio-inspired heuristic search algorithm that utilizes genetic operators to guide the search process. It can be described as an algorithm that allows the fittest among individuals to survive over consecutive generations. Algorithm 2 shows the simplest form of a GA (called canonical genetic algorithm), where the best individuals (solutions) are selected to rise to the next generation; in the next generation, offspring of those selected individuals will be generated through genetic operators. The genetic operators in the GA are selection, mutation, and crossover, as discussed in section 3.2. GAs use random search "intelligently" in evolution, which make them desirable options over other algorithms such as linear programming, depth-first, breath-first, and heuristic algorithms.

The size of the search space in the GA depends on the length of the genomes (number of bits, when genomes are represented in a string of bits). For an L-bit-long genome, the search space is an L-dimensional hypercube with a size of 2^L. In some literature [144], it was stated that a search space of 2^{400} would be "ridiculously large" assuming that there was one solution. However, this search space size is acceptable when there are plenty of "good" solutions randomly scattered.

The main drawbacks of GAs are slow convergence, no guarantee of finding global optima, and the need for fine tuning the evolution parameters. In the case where an EHW is utilizing GA, which is the most common use case [134, 13], convergence is accelerated because evolution is running in hardware. EHW is often used to find local optima and does not actually search for global optima; in fact, GA has no means to check whether a solution is a global optimal solution. Tuning the evolution operators can be performed experimen-

Algorithm 2: Pseudo code for the canonical genetic algorithm.

```
   Result: GA returns the fittest genome
1  g_r = g_m = g_c = null
2  generation = 0
3  while not termination condition do
4  |  if generation=0 then
5  |  |  for i ← 1 to population do
6  |  |  |  g_ri = generate_random()
7  |  |  end
8  |  else
9  |  |  for i ← 1 to (population×mutation_rate) do
10 |  |  |  g_p = random_select(G_parents)
11 |  |  |  g_mi = generate_mutation(g_p)
12 |  |  end
13 |  |  for i ← 1 to (population×crossover_rate) do
14 |  |  |  g_p1 = random_select(G_parents)
15 |  |  |  g_p2 = random_select(G_parents)
16 |  |  |  g_ci = generate_crossover(g_p1, g_p2)
17 |  |  end
18 |  end
19 |  G_children ⇐ [g_r1 ..., g_m1 ..., g_c1 ...]
20 |  G_parents ⇐ [ ]
21 |  for i ← 1 to parents do
22 |  |  g_s = get_fittest(G_children)
23 |  |  G_parents.add(g_s)
24 |  |  G_children.remove(g_s)
25 |  end
26 |  generation++
27 end
```

tally after selecting the application, function set and the fitness function. However, if the tuning did not occur, the risk is slowing the evolution, which is undesirable but not fatal.

The advantages of GAs are as follows:

1. GAs are robust search algorithms because they use probabilistic computations and naturally embody a high level of noise tolerance.

2. The mapping to phenotype is feasible since the genotype is represented in binary.

3. There are many (sub)optimal solutions scattered in the search space. A GA has no bias toward any subregion of the solution space.

4. By using mutation, evolution is guarded against becoming stuck in local optima.

5. GAs are domain independent. However, the function set and fitness function are domain dependent.

6. A GA is modular and inherently parallel, which makes it easily distributed.

There are many variants of GAs; some of the well-known algorithms are parallel GAs [145], adaptive GAs [146], GA with elitist selection [147], messy GA [148], learning gene linkage GA [149], and gene expression GA [150].

Focusing on GAs that are hardware capable, Higuchi et al. has many publications on implementing a GA on an FPGA/PLD to perform gate-level evolution [151] and function-level evolution [152]. Gallagher, Vigraham, and Kramer proposed a family of compact genetic algorithms that could be integrated in digital systems without a substantial increase in size and complexity [153]. Stomeo, Kalganova, and Lambert presented a scalable GA built on a PLA for the design of digital circuits that could evolve faster than a traditional GA [154]. Another enhanced GA was proposed in [155] to reduce the evolution time and the required computations. Li, Fialho, Kwong, and Zhang proposed an adaptive operator selection method in which the application rate of different genetic operators is determined in an online fashion based on their performance [156].

3.6 Summary

The brain of an EHW is the evolutionary algorithm. An EA requires two types of means, one to generate and select genomes, called genetic operators, and the other is to measure the "goodness" of a genome, called the fitness function. There are four common types of genetic operators. These are selection, mutation, crossover, and elitism. Selection is the

operator to select some of the best children of the current generation to be parents for the next generation. Mutation is the operator that maintains genetic diversity as it creates new chromosome combinations by randomly mutating current genes. Crossover is the operator that recombines genomes to produce hopefully better combinations. Elitism is a selection mechanism that allows elite genomes to survive in different generations.

The fitness function is simply a method to provide a score for any genome. This score alone is used to compare among genomes. Designing a fitness function is the greatest challenge in designing an efficient EA; it needs to be accurate but simple, capable of high speed, and comprehensive. There are four common types of EAs: ES, GP, CGP, and GA. All of them have limitations. However, if they were to be implemented in hardware, GA would be the best fit for many reasons, including (1) easy mapping to hardware, (2) being modular, (3) being a robust random guided search, (4) being domain independent, and (5) the use of mutation and crossover. There are many successful examples of implementations of GA on hardware.

CHAPTER 4

EVOLVABLE HARDWARE SYSTEMS

4.1 Introduction

Evolutionary hardware is defined as a set of hardware modules that have the ability to autonomously design and optimize a system using stochastic algorithms, specifically evolutionary algorithms. These algorithms are used for searching a given solution space for a set of inputs and internal parameters to achieve an optimal or suboptimal solution. The power of evolutionary algorithms comes from their ability to search the entire solution space, including the areas that would often be missed if humans were designing an algorithm. Thus, evolutionary algorithms need to have no bias or constraint that can prevent the system from exploring any subspace of the solution space [157].

In this chapter, EHW systems will be explored, starting with their classifications followed by the implementations. Systolic arrays are one of the implementations that will be discussed in more detail here because they are related to the proposed system.

4.2 Classifications of Evolvable Hardware Systems

EHW systems can be classified in many ways, including hardware platform, reconfiguration scheme, evolutionary algorithm, evolutionary level of abstraction, hardware evolution type, operation mode, and application area, as summarized in figure 4.1.

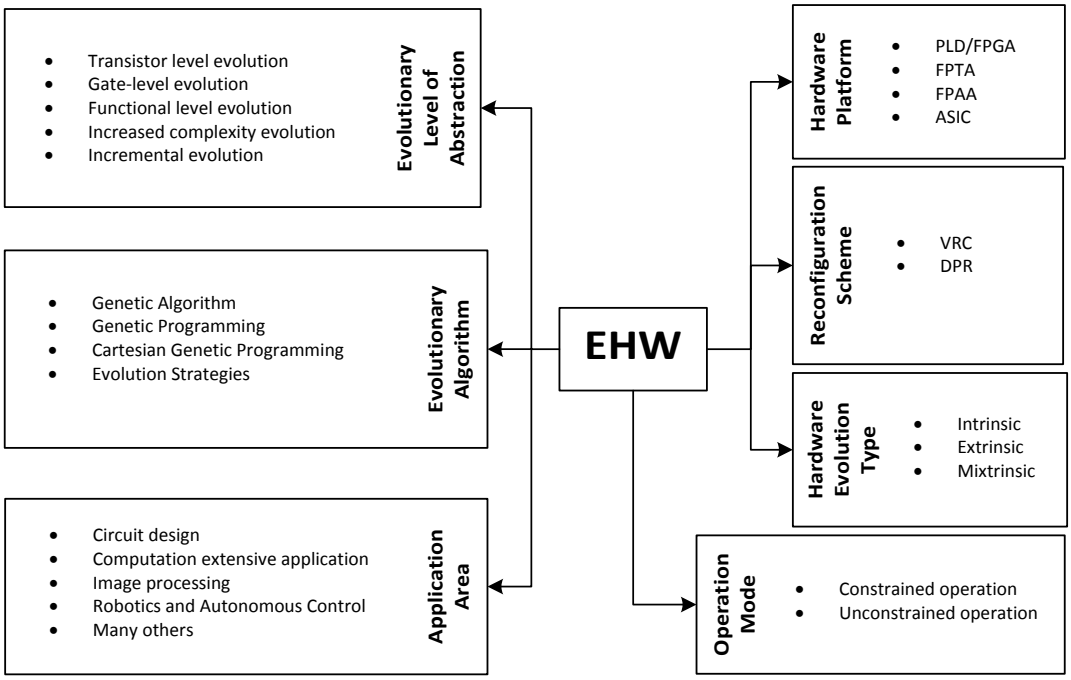


Figure 4.1: Classification schemes of EHW.

4.2.1 Hardware Platform

EHW systems can be classified based on the hardware platform used. For digital systems, FPGAs (a type of PLD) and ASICs (also known as custom hardware) are commonly used. FPTAs can be used for analog and digital systems. FPAAs are used for analog EHW systems. Based on the published research, ASIC-based EHW systems were the common trend for digital systems until the early 2000s, when FPGA-based EHW systems became the mainstream.

4.2.2 Reconfiguration Scheme

EHW is classified as a VRC-based system when the reconfiguration scheme, as discussed in section 2.2.4, in use is a VRC or DPR-based system when using DPR. Note that VRC and DPR are generally accepted terms that mean signal multiplexing and time multiplexing, respectively.

4.2.3 Evolutionary Algorithm

EHW systems can be classified based on the EA that is used. The common GAs for EHW systems are GA, ES, GP, and CGP. Although many systems use modified versions of the EA, they are still classified using the original EA. There is a strong correlation between the EA used and the application domain.

4.2.4 Evolutionary Level of Abstraction

Evolution is performed using different levels of abstraction, which describe the function set that is in use. Netlist-level evolution was the mainstream when problems were small (e.g., 1-bit adder) and architectures, specifically FPGAs, had open (and relatively simple) architectures. At this level, device-specific modules (e.g., CLBs) were used as the reconfigurable blocks, and genetic operations occurred on the configuration bits of LUTs. Subsequently, logic-level evolution became the trend for many reasons, including (1) the increased complexity of problems, (2) avoiding low-level device-specific details, and (3) because some manufacturers concealed the netlist-level details of their devices. AND, OR, XOR, NOT, and Multiplexer are examples of functions used at this level. Finally, the function level is the highest level of abstraction that emerged to bridge the gap between gate-level abstraction and coarse-grained systems. It brings the function sets closer to the

application domain and away from low-level device-specific details. Within the function-level abstraction, there is a wide spectrum of functions; for example, functions can be as simple as adders and as complex as FFT.

In contrast to the previous "fixed" levels of abstraction, some studies focused on supporting dynamic levels of abstraction. Increased complexity evolution is a novel mechanism in which the system is initially evolved by evolving smaller subsystems [158]. Incremental evolution is another novel mechanism that supports automatic incremental evolution in two directions: full system to subsystem and subsystem to full system [159].

4.2.5 Hardware Evolution Type

The type of hardware evolution is based on where genomes are evaluated and where the fitness function and EA are running, as discussed in section 1.2.2. The summary for these types is provided in table 4.1.

Table 4.1: Summary of hardware evolution types.

Hardware Evolution Type	Genome evaluation	EA and fitness function
Extrinsic	Software	Software
Intrinsic	Hardware	Software (PC)
Complete intrinsic	Chip A	Chip A
Multi-chip intrinsic	Chip A	Chip B
Multi-board intrinsic	Board A	Board B

4.2.6 Operation Mode

EHW systems can operate under two modes: constrained operation and unconstrained operation. Hardware that has a deterministic output is said to be running in constrained operation. An example of this is the evolution of a digital circuit, where the outcome

circuit is expected to function identically to the initial evolved design even on a different digital architecture (e.g., a circuit that was evolved on Xilinx FPGAs can be implemented on Altera's FPGAs). Unconstrained operation, in contrast, is where an evolved hardware system is free to use any parameter in the environment, including those parameters that are not typically considered in a conventional design flow, e.g., using the analog characteristics of a digital device or the temperature in circuit design.

4.2.7 Application Area

EHW systems can be classified based on the application area, as discussed in section 1.3. Some of the common areas are circuit synthesis, computation-extensive applications, image processing, robotic and autonomous control, satellite, communications, data compression, data encryption, data mining, and so forth.

4.3 Evolvable Hardware Implementations

Integrating an evolutionary algorithm with a reconfigurable architecture creates an EHW system. Looking at the past 50 years of EHW history, many milestones have been achieved. In 1963, the first publication in this field was by Larry Fogel. He used a hardware system driven by a primitive version of GA to accelerate the solving of wire routing problems on the Atlas Missile Guidance system [23]. Other works of the same nature followed over the next two decades, such as that by Alvin Owens and Michael Walsh [25]. In the early 1980s, EHW was used in solving gas pipeline routing problems [25]. In the late 1980s, the field of EHW witnessed incredible advances sparked by the widespread use of computers. In 1992, de Garis predicted that the commercial FPGAs at that time were capable of building a "Darwin Machine", a machine capable of evolution [28].

Followed by de Garis's prediction, in 1999, Adrian Thompson conducted an experiment to search for a solution for tone discrimination on FPGAs using EAs [157]. A tone discrimination circuit is not the typical application for FPGAs, specifically without using a clock or an external timing reference. FPGA-based EHW systems could find an unconventional solution using the underlying physics of the substrate, a solution that would not be considered by human intelligence itself. The significance of this work was (1) proving that an EHW can find solutions that circuit designers do not know about, (2) FPGAs are good hosts for EHW systems, and (3) FPGA-based EHW systems were the first online EHW (intrinsic), while all previous implementations of EHW were offline or by simulation (extrinsic) only. Additionally, this experiment was an example of evolution in an unconstrained operation where the evolved circuit used the underlying substrate, which is not expected in a digital design domain. This evolved circuit only worked on the specific type of FPGAs used in the evolution process and could not be replicated to other FPGAs with similar architectures. Moreover, the evolved circuit was temperature dependent and did not perform well using a temperature different than what was used in the evolution process.

Searching the literature from the past 30 years, many successful EHW implementations can be found due to the advances made in computing systems and FPGAs. Some of these implementations were novel reconfigurable architectures fabricated on custom-hardware with traditional EAs [11, 107, 110, 12, 109, 108]. Other implementations were EHWs with novel FPGA-based architectures with traditional EAs [160, 161, 162, 152, 163, 164]. Some of the EHW systems were using traditional reconfigurable hardware (e.g., PLD or FPGAs) with novel EAs [153, 151, 155, 152, 154]. There were no EHW implementations that were constructed with a novel reconfigurable hardware core and novel EA; our proposed system is the first that includes both.

4.3.1 Systolic Arrays

An architecture of interest is the systolic array, which is one of the common hardware architectures that provides a high-level of parallelism. The systolic array is an array of tightly coupled functional cells, called processing elements, with a linear data dependency. This architecture was proposed by Kung in 1978 [17]. As Kung and Leiserson [18] wrote, *"A systolic system is a network of processors which rhythmically compute and pass data through the system. Physiologists use the word "systole" to refer to the rhythmically recurrent contraction of the heart and arteries which pulses blood through the body. In a systolic computing system, the function of a processor is analogous to that of the heart. Every processor regularly pumps data in and out, each time performing some short computation, so that a regular flow of data is kept up in the network"*. The architecture was invented to be patterned efficiently in VLSI systems for computationally intensive applications (i.e., convolution computation).

Processing elements come in many shapes; in fact, in many of the classical papers [17, 18, 165, 166], the design of a processing element was to serve in a specific application. The most common designs are the "type R" and "type H" proposed by Kung et al., as shown in figure 4.2 and figure 4.3. The different shapes were designed to achieve certain functions where "type R" were used for two-operand operations and "type H" were used for three-operand operations.

Systolic arrays are well suited for EHW systems and have been used several times [167, 104, 168, 3, 169, 170]. A common paradigm for many systolic array implementations is to have functional cells in a rectangle shape with communication ports going in four directions (similar to type R).

Applications with clearly defined functional blocks that can be dynamically swapped

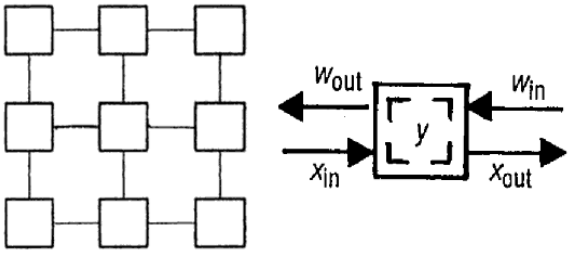


Figure 4.2: Type R systolic array proposed Kung et al. in 1978 [17].

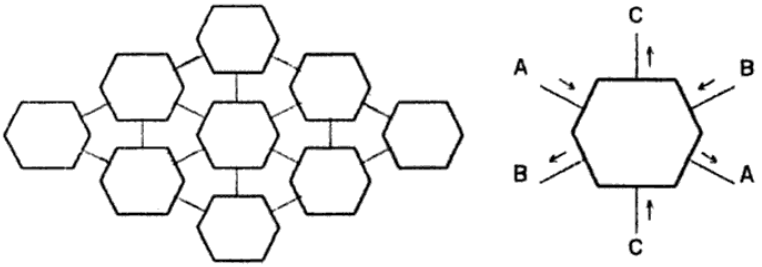


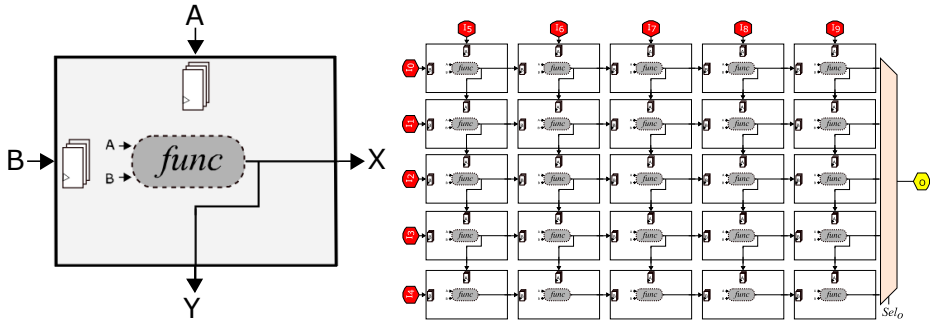
Figure 4.3: Type H systolic array proposed Kung et al. in 1979 [18].

during execution are well suited for systolic arrays, such as image processing, big data searches, data sorting, communication decoders, packet filtering, accelerated SQL modules, matrix operations, computation accelerators, and so forth [22]. In this work, image processing applications were chosen for their distinct phases of processing.

State-of-the-Art Systolic Array

Prof. Sekanina's research team at Brno University of Technology, Czech Republic, have conducted considerable research in using systolic arrays in image processing applications [84, 90, 169, 171, 172, 30]. The utilized system was a 2D array of medium-grained processing elements that were reconfigured using a DPR scheme, as shown in figure 4.4.

The system showed impressive results, although there were some major drawbacks, including the following:



(a) State-of-the-art PE **(b)** State-of-the-art systolic array

Figure 4.4: A 5×5 systolic array of state-of-the-art PEs, where the array uses a single output and PEs use DPR reconfiguration scheme.

1. Slow reconfiguration due to the use of DPR solely. In [30] [172] the team compared DPR against the "traditional" VRC, which resulted in the argument that using DPR alone was the best choice.

2. Parallelism was not exploited. Only a single output is evaluated per genome [90], while many can be evaluated in parallel.

3. The EA (ES/CGP hybrid) is not aware of the genome structure, thereby causing inefficient evolution operations. For example, mutation of bits representing a PE that is not in the datapath of the selected output is not worth evaluating.

4. The systolic array suffers from "narrow" data propagation. For example, based on figure 4.4, all cells in rows 2, 3, and 4 are not contributing to the output at row 1.

In a recent paper (in 2015), a VRC/DPR hybrid FPGA-based EHW system was proposed, which was the first attempt toward this goal [84]. This system, however, has many drawbacks. It involves high-complexity mechanisms as it deals with low-level bitstream modifications, which are device specific. We believe that our new VRC/DPR hybrid contribution is more general and robust.

4.4 Summary

Pairing a reconfigurable hardware core with an evolutionary algorithm creates an EHW system. There are several ways to classify an EHW system, in which some of them are somewhat connected, such as (1) using CGP/GP (evolutionary algorithm) for netlist-level evolution (evolutionary level of abstraction), (2) using extrinsic evolution (hardware evolution type) for circuit synthesis (application area), or (3) using DPR (reconfiguration scheme) when FPGAs (hardware platform) are used.

There were many successful implementations of EHW systems. However, the majority of the systems were constructed after improving either the reconfigurable hardware core *or* the evolutionary algorithm. A systolic array is one of the commonly used architectures for EHW systems. It is defined as a pipelined grid of processing elements with a linear data dependency. It was invented, along with many processing element architectures, in the late 1970s to allow a modular design for VLSI. Although the state-of-the-art systolic array proposed by Sekanina et al. [30] has impressive performance, it still has some drawbacks, including (1) slow reconfiguration, (2) lack of parallelism, (3) inefficient genetic operators, and (4) limited data propagation.

CHAPTER 5

HEXARRAY PLATFORM DESIGN

5.1 Introduction

This dissertation designs an EHW using a new reconfigurable hardware core and a context-aware GA. The design considerations of the proposed platform were two-fold:

- The reconfigurable hardware core should have a high level of parallelism to improve performance.
- The reconfigurable hardware core should have flexible routing to improve system reliability.
- The reconfigurable hardware core should have a reconfiguration scheme that combines the merits of the virtual and native reconfigurations while avoiding their drawbacks: wasting resources and slow reconfiguration.
- The evolutionary algorithm should be guided to make "smarter" decisions to accelerate evolution.

To meet these considerations, a novel reconfigurable hardware core is designed. The system is a systolic array called HexArray, which features a high level of parallelism. HexArray is constructed using a novel PE called HexCells. HexCells feature flexible ports that, when tiled in a HexArray, allow a "virtual" DPR and other additional features, such

as routing around faulty cells (discussed in subsection 5.6.1) or cells under reconfiguration (discussed in subsection 5.6.2).

Additionally, a genome-aware genetic algorithm (GAGA) is designed. The GAGA is designed to perform genetic operations based on understanding the genome structure and cell dependency; in addition, for randomly generating (selecting) genomes, GAGA follows a statistical model and "common sense" to reduce redundant evaluations.

Early in the project life, a software model (a simulator) of the system was implemented to prove the concept. This simulator is functionally identical to the hardware system (although the performance difference is 1:1000). The discussion of the simulator will serve as a high-level workflow of the system and will be discussed first in section 5.2.

Finally, a comparison between HexArray and the state-of-the-art systolic array will be presented in section 5.7.

5.2 HexArray Simulator

A simulator for the HexArray platform was initially developed to validate the effectiveness of the proposed platform and to tune some parameters. The text-based simulator was written in the Python programming language and was tested on many data sets.

After setting the evolution parameters, the simulator input is a training image, e.g., noisy image. Starting with this input, the simulator will evaluate a large set of genomes – genomes in this case are image filters – and measure their fitness using a fitness function. In our case, the fitness function requires a reference image to be able to assess how close or far a genome is from achieving the target. Here, the target is to find an image filter that transforms the training image to bring it closer to the reference image, as shown in figure 5.1. The fittest genome will be selected as the best solution at the end of the evolution

process. The user adjusts the simulator parameters to control the evolution process. The parameters are shown in table 5.1.

(a) Training image **(b)** Evolved image **(c)** Reference image

Figure 5.1: (a) A training image with 20% salt & pepper noise. (b) Image produced by algorithm with a 63% noise reduction. (c) Reference image used for the fitness calculations.

Once the parameters are set and evolution is started, the system generates a genome and applies it to the array cells and input ports. A small sliding window on the training image pixels will be used iteratively to generate filtered pixels. The array has multiple outputs, and each output has a fitness unit. After the last pixel of the image, the system reads the fitness values. Fitter genomes are selected as parents for the next generation. The system uses these parents to generate better offspring. The number of genomes required for generating a good filter is problem dependent, but it is generally a large number (e.g., $> 10^N$, where $N \geqslant 4$).

During the simulator evaluations, it was observed that there were many solutions with no observed improvement. Consequently, a new parameter, $fitness\_threshold$, was added to the algorithm to only report genomes with appreciable improvements. Another observation was the effect of the $gen\_operators\_mode$ parameter, where A_GENOME appeared to be the best option. This is expected for two reasons:

Table 5.1: Simulator parameters to control the evolution process.

Parameter	Description	Example
$\mathbb{R}\times\mathbb{C}$	HexArray size	8×8
window_size	Image data window size	5×5
reference_img	Reference image file name	Lena
training_img	Training image file name	Lena_sp10
generation	number of generations	10
population	number of individuals per generation	1000
parents (μ)	number of parents to be selected	100
m_rate	mutation rate	0.3
c_rate	crossover rate	0.4
M_{GAM}	number of bits to be mutated	1
fitness_threshold	A genome is good when its fitness is less than the fitness_threshold of the best fitness of the previous generation.	0.99
gen_operators_mode	To which part of the genome genetic operations can be applied; the options are A_GENOME for Array-Genome or ALL for Array-Genome and Input-Genome	A_GENOME

- In ALL mode, the search space significantly increases, which slows evolution (for example, for an 8×8 HexArray, the size of A_GENOME is 2^{640}, while the size of A_GENOME+I_GENOME is 2^{795}).

- Genetic operations are not effective when applied to the input genome.

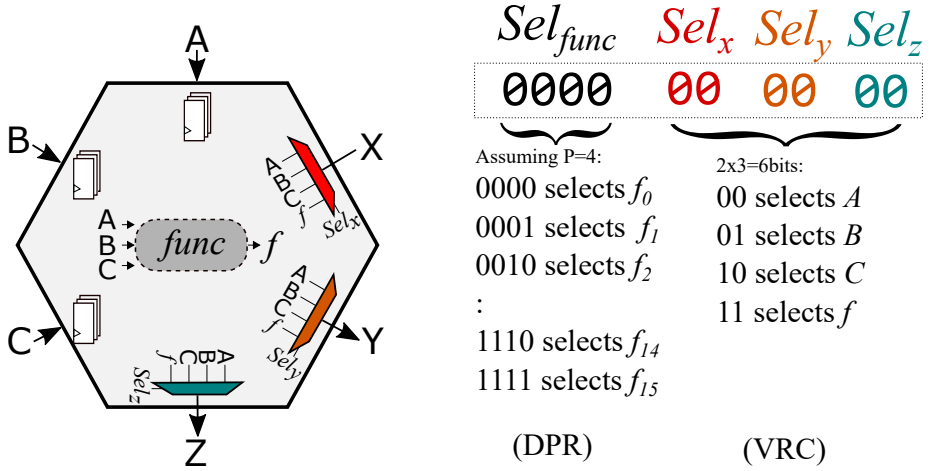
An intrinsic issue of simulators and software models in general is poor performance. This simulator is no exception. Moreover, it was observed that parameters such as array size have a direct impact on speed. The simulator took 3.5 seconds to evaluate a single genome on a 3× 3 array, whereas 14.4 seconds elapsed when evaluating a genome on an 8×8 array. For an evolution process of 10 generations with populations of 1000 on an 8×8 array, the simulator took more than 40 hours. However, implementing the system

in hardware achieved an approximately 2500× speed up, and running the same evolution example mentioned previously on the hardware module took less than one minute.

5.3 Proposed Reconfigurable Hardware Core

The discussion in this section will begin with the design of HexCell. HexCell will be discussed from two perspectives: cell structure and cell chromosome representation. The exploration of HexArray will follow, including the array structure with its auxiliary modules and the array genome representation.

5.3.1 A Novel Processing Element – HexCell



(a) HexCell architecture **(b)** HexCell chromosome

Figure 5.2: The HexCell structure and representation: the HexCell's functional unit is on a dynamic partition while the remaining logic is static. The HexCell chromosome has four genes, where three genes implicate a VRC and one implicates a DPR.

HexCell Structure

HexCell is a virtual hexagonal-shaped PE designed to be tile-able in systolic arrays, as shown in figure 5.2.a. The cell's main components are *input ports*, *input buffers*, *functional unit*, and *output ports*. HexCell is different than the type H PE, proposed by Kung et al. [18], figure 4.3, where the later contains a memory and executes a fixed function.

1. Input Ports: The cell has three inputs, north (N), north west (NW), and south west (SW), called A, B, and C, respectively. Each input has a data bus and a ready signal. The ready signal works as a "WRITE ENABLE" for the input buffers.

2. Input Buffers: Every cell input must have a data buffer (FIFO) because the pipelined data are coming at different latencies on different input ports. The depth of an input buffer depends on the cell location, where the maximum depth is dependent on the size of the array. For a cell located at row x and column y in an $\mathbb{R}$ rows $\times$ $\mathbb{C}$ HexCell array, where $x \in \{1,...,\mathbb{R}\}$ and $y \in \{1,...,\mathbb{C}\}$, the depth for input buffers is:

$$Depth_A = \begin{cases} y + \lfloor \frac{y}{2} \rfloor & \text{if } x = 1 \\ 1 & \text{for all other} \end{cases} \tag{5.1}$$

$$Depth_B = \begin{cases} x & \text{if } y = 1 \\ y + \lfloor \frac{y}{2} \rfloor & \text{if } x = 1 \ \& \ y \in \{1,...,\mathbb{C}\}_{odd} \\ 2 & \text{for all other} \end{cases} \tag{5.2}$$

$$Depth_C = \begin{cases} x & \text{if } y = 1 \\ \mathbb{R} + \frac{3}{2}y - 1 & \text{if } x = 1 \ \& \ y \in \{2, ..., \mathbb{C}\}_{even} \\ 1 & \text{for all other} \end{cases} \tag{5.3}$$

and the maximum depth is:

$$Depth_{Max} = \mathbb{R} + \mathbb{C} + \left\lfloor \frac{\mathbb{C}}{2} \right\rfloor - 1. \tag{5.4}$$

3. Functional Unit: The core of the cell is the functional unit, which is a reconfigurable partition that can be reconfigured at run time to one of many functions using DPR. Designing the library of functions is performed by the user and is based on the desired application. The functions can operate on all three inputs or just a few of them; the operation starts when all dependent inputs are ready. Whether the functions are simple or complex will decide the granularity of the EHW system. It is recommended, but not required, to select $2^{\mathbb{P}}$ functions, e.g., 4, 8, 16 and so on functions, as shown by:

$$Sel_f \in \left\{ f_0, f_1, f_2, ... f_{2^{\mathbb{P}}-1} \right\}. \tag{5.5}$$

For real-world applications where resource utilization is optimized, it is important to identify the granularity of the used functions; thus, an FPGA with a compatible dynamic partition size is selected. Our chosen application has a mixture of fine-/medium-grained functions, but an FPGA with coarse-grained dynamic partitions has been used due to availability, which is suitable for our purpose as a proof of concept. A library for the used functions in the proposed system is provided in table 5.2

with index, name, description and dependency vector given for each function. The dependency vector is needed for the genome-aware operators, where an input with a dependency value of 1 means that the function is dependent on this input and that the operation will not start until the data ready signal for that input is asserted.

Table 5.2: Function set for the selected image processing application.

Function Index	Function Name	Function Description	Dependency A, B, C
f_0	Average	`AVG:=(A+B+C)/3`	1, 1, 1
f_1	Conditional	`MUX:=C[7]?A:B`	1, 1, 1
f_2	Greater	`GRT:=(A>B)?0x00:0xFF`	1, 1, 0
f_3	Full High	`FLH:=0xFF`	0, 0, 0
f_4	Bitwise OR	`OR_:=A|B|C`	1, 1, 1
f_5	Bitwise AND	`AND:=A&B&C`	1, 1, 1
f_6	Bitwise NOT	`NOT:=~A`	1, 0, 0
f_7	Bitwise XNR	`XNR:=~(A^B^C)`	1, 1, 1
f_8	Shift Left	`SHL:=C<<B[7]`	0, 1, 1
f_9	Shift Right	`SHR:=C>>B[7]`	0, 1, 1
f_{10}	Maximum	`MAX:=max(A,B,C)`	1, 1, 1
f_{11}	Minimum	`MIN:=min(A,B,C)`	1, 1, 1
f_{12}	Low Pass	`LPS:=A&0x0F`	1, 0, 0
f_{13}	High Pass	`HPS:=A&0xF0`	1, 0, 0
f_{14}	Different	`DIF:=(B[7]^A[7])?0xFF:0x00`	1, 1, 0
f_{15}	Intensify	`TNS:=A[7]?A<<2:A>>2`	1, 0, 0

4. Output Ports: HexCell contains three output ports: north east (NE), south east (SE), and south (S), called X, Y, and Z, respectively. Each output can be independently sourced from any of the input ports or the output of the functional unit. By having this output control, the VRC reconfiguration scheme is achieved since changing an output port selection of a cell will affect all dependent cells along the datapath. A cell output can be selected to be A, B, C, or f, where f can be $f_0, f_1, f_2, ...f_{2^{\mathbb{P}}-1}$, resulting in $3+2^{\mathbb{P}}$ possibilities. In other words, $Sel_x, Sel_y, Sel_z \in \{00, 01, 10, 11\}$, resulting in $X, Y, Z \in \{A, B, C, f\}$.

HexCell Chromosome Representation

HexCell is represented by a chromosome of a $(\mathbb{P}+6)$-long binary string of genes, which contains two substrings as shown in figure 5.2.b. One is a fixed-length substring with 6 bits (three genes) to control the selections of three output ports using a VRC scheme. The other substring is the function selection, which implicates a DPR scheme. The chromosome is structured as

$$Chromosome_{HexCell} = \langle Sel_z,\ Sel_y,\ Sel_x,\ Sel_f \rangle . \tag{5.6}$$

5.3.2 A Novel Systolic Array – HexArray

HexArray is the reconfigurable hardware core of the proposed EHW system. HexArray is discussed here from two perspectives: structure and representation. The hardware entities of HexArray are the data window controller, genome register, array input controllers, systolic array, and array output controllers. HexArray is represented by a genome, which is the complete set of configurations to replicate the current state of the system. At the end of this chapter, a comparison between HexArray and the state-of-the-art systolic array is performed, and a summary of the properties of HexArray is outlined in table 5.3.

HexArray Structure

1. The data window controller is responsible for providing a sliding data window that includes a window of 5×5 pixels from the training image, one pixel from the reference image and the pixel coordinates (X, Y), as shown in figure 5.3. Note that the reference pixel is not used as an input pixel for the array and is used by the fitness function. This module has a “Start” signal that propagates to the array input

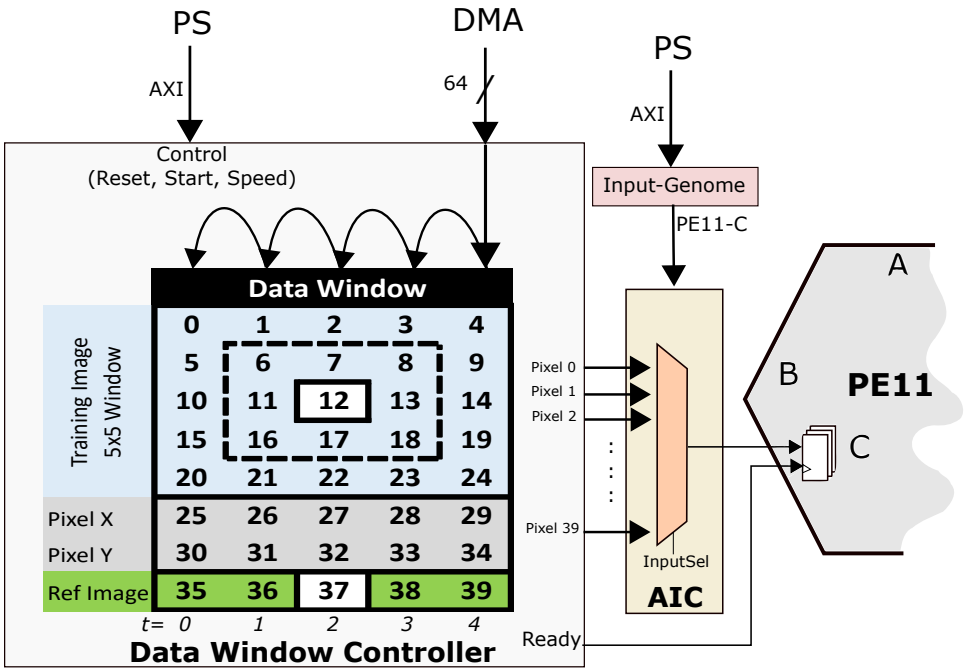


Figure 5.3: Data window controller formats the input data stream received from the DMA as a sliding data window accessible by the array input controllers, which are controlled by the `i_GENOME` and fed into the array cells.

controllers, which start or pause the execution for the entire array. The controller contains a DMA with a 64-bit bus.

2. The genome register contains an input genome (`I_GENOME`) and an array genome (`A_GENOME`) that can be written by the PS using the AXI interface [101]. `I_GENOME` defines the pixel selection from the data window provided by the data window controller for all array input controllers. `A_GENOME`, in contrast, has two types of data. One type of data is for controlling cells' output multiplexers. These data have a fixed length of $6 \times \mathbb{R} \times \mathbb{C}$, and any change to these data will take effect on the hardware instantaneously, as the change implicates a VRC scheme. The second type of `A_GENOME` data is for selecting the cells' functions. These data have a length of

$\mathbb{P} \times \mathbb{R} \times \mathbb{C}$, and any change to these data means that cells' functional units need to be natively reconfigured, implicating a DPR reconfiguration. This process of reconfiguration is maintained by the PS using the PCAP [101].

3. The array input controller (AIC) selects a pixel from the sliding data window and feeds it into an array input port based on its desired selection signal defined in `I_GENOME`, as shown in figure 5.3.

4. The systolic array is a homogeneous array of HexCells patterned in a 2D symmetrical mesh, where each cell is neighboring 6 cells, except those on the array boundary, which may be connected to AICs or array output controllers (AOCs), as shown in figure 5.4. Each cell receives data from neighboring cells or AICs in the N/NW/SW directions through the ports A, B, and C, respectively, and is sending data to other cells or AOCs on the NE/SE/S directions via the ports X, Y, and Z, respectively.

5. The array output controller is responsible for calculating the fitness value for each array output. The selected fitness function is the mean absolute error (MAE), but it can be any user-defined function. The function, which requires a reference data, is defined as follows:

$$MAE = \frac{1}{WL} \sum_{m=1}^{m=W} \sum_{n=1}^{n=L} |Out(m,n) - Ref(m,n)| . \tag{5.7}$$

Upon processing the "expected count" of pixels, the AOC will store the calculated fitness value in an FIFO. The expected count is typically the total number of pixels in the used image, which is programmed by the PS using the AXI interface, as shown in figure 5.5.

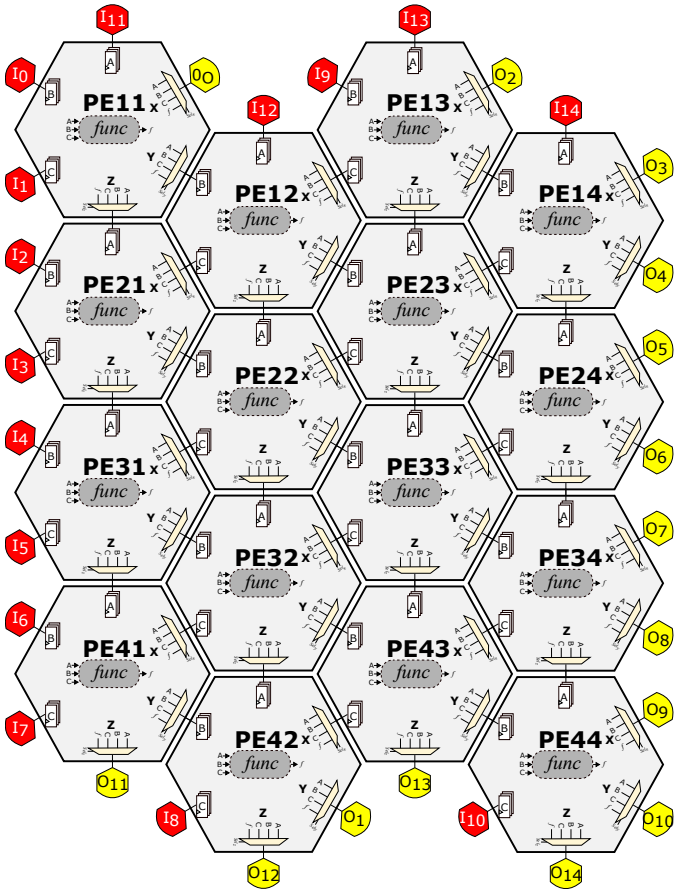


Figure 5.4: 4×4 HexArray with AICs (shown in red) and AOCs (shown in yellow).

HexArray Genome Representation

The genomes of HexArray take two forms. When a genome is not evaluated, it takes the form $\texttt{GENOME}_{HexArray}$ = ⟨ I_GENOME, A_GENOME ⟩. However, when a genome is evaluated and an output is selected, it takes the form $\texttt{GENOME}_{HexArray}$ = ⟨ I_GENOME, A_GENOME, Active-Output ⟩.

- The input genome (I_GENOME) is a string of bits that defines the selected pixel for each AIC. In our implementation, each AIC can point to one of 35 pixels (25 training

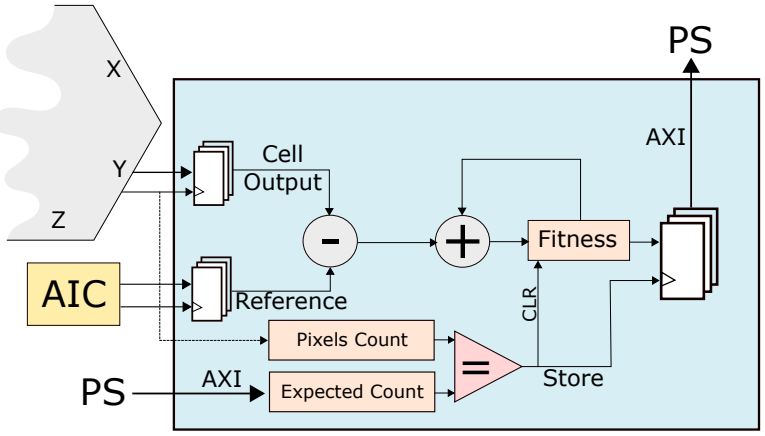


Figure 5.5: Array output controller module which accumulates the absolute difference between the evolved pixel and the reference pixel for "Expected Count" pixels.

image pixels and 10 pixels for X/Y coordinates); thus, 6 bits are needed, with the allowed range being 0 to 34.

- The array genome (`A_GENOME`) is a string of bits that defines the configuration of all HexCells in the systolic array. Each cell is represented by $\mathbb{P}+2+2+2$ bits for Sel_{func}, Sel_x, Sel_y, and Sel_z, respectively.

- `Active-Output` is an integer number that identifies which array output port is used to obtain the desired result of a chosen genome. This is needed because of the parallelism in HexArray, where there will be different output data coming from different output ports and the EA or the user needs to know the specific output.

5.4 Proposed Genome-Aware Genetic Algorithm (GAGA)

The GAGA is a high-level algorithm, and the chosen implementation is written in C/C++ and runs on the hardcore processing system on the Xilinx Zynq-7000 All-Programmable SoC [101]. The program has a class-based structure, which makes it modular and easy

to understand. The algorithm includes a library of utility functions that are necessary for the content-aware processing of a genome. These functions allow the GAGA to identify the cells' hierarchical dependencies and perform smarter genetic operations. To understand how the genome-aware genetic operators work, the main code structures need to be outlined first. Two main classes (`GENOME` and `DV`) along with some utility functions and code structures will be discussed in the following section. Most of the discussion is performed using code listings.

5.4.1 Algorithm Utility Functions

All utility functions work under global rules (`GLOBAL_RULES`), which allow control over what cells can or cannot be used or what values ports can take. The global rules are automatically generated by combining the temporary rules (`TEMPORARY_RULES`) and the permanent rules (`PERMANENT_RULES`), where the GAGA controls both rules. The temporary rules are temporary and can be changed throughout the evolution process; an example using these rules is supporting the virtual resizing of the array in evolution, where the array starts with a small array size, in which the `TEMPORARY_RULES` ban the usage of cells' functional units outside the desired array size. When larger arrays are needed, usage is allowed for the next row and column of cells to expand the array size. The permanent rules, in contrast, are fixed. They are designed to permanently ban the use of a certain cell (e.g., a faulty cell). To understand them better, consider the example in listing 5.1.

```
1
2  TEMPORARY_RULES = {PE11_X: [A, B],  # X selects A or B (not C or f)
3                     PE11_Y: [C, f],  # not B or A
4                     PE11_Z: [f],     # Fixed to f
5                     PE22_F: [0, 1, 2, 3, 4]} # f0 to f4 are allowed
6
7  PERMANENT_RULES = { PE11_X: [A, B, C], # Assuming that PE11 is
8                      PE11_Y: [A, B, C], # faulty cell and we need
9                      PE11_Z: [A, B, C]} # to avoid using it
10
```

```
# The algorithm generates GLOBAL_RULES, automatically, anytime
# a change to the TEMPORARY or PERMANENT rules occurs.
GLOBAL_RULES = combine(TEMPORARY_RULES,  PERMANENT_RULES)

# combine function results in a rule that satisfies both.
# If constraints cannot be resolved, function goes with
# PERMANENT_RULES

# The result will be:
# GLOBAL_RULES ={ PE11_X: [A, B],
#                 PE11_Y: [C],
#                 PE11_Z: [A, B, C],
#                 PE22_F: [0, 1, 2, 3, 4]}
#
```

Listing 5.1: GAGA utilizes temporary, permanent, and global rules.

A function that is related to `GLOBAL_RULES` is *get_rand_chromosome*, which is described by listing 5.2. The `List` is a code structure that is frequently utilized in the proposed algorithm. `List` can contain any (same) type of objects. The functions associated with this class are *add*, *remove* and *random_select*, as discussed in listing 5.3.

```

# get_rand_chromosome returns a random chromosome value
# that does not violate the GLOBAL_RULES.
# Assuming:
#    GLOBAL_RULES ={ PE12_X: [A, B],
#                    PE12_Y: [C],
#                    PE12_Z: [A, B, C],
#                    PE12_F: [0, 1, 2, 3, 4]}

val=get_rand_chromosome(1, 2)       # (1, 2) means PE12
# val can be  <Z=A, Y=C, X=A, F=0>
#     but not <Z=A, Y=C, X=A, F=5>
#          or <Z=A, Y=C, X=C, F=0>
#          or <Z=A, Y=A, X=A, F=0>
#          or <Z=f, Y=C, X=A, F=0>
```

Listing 5.2: Example for the function to generate a random chromosome. Note that generated chromosomes are `GLOBAL_RULES`-compliant.

```
G=[]        # G is an empty list

# "add" function: adds to the List
G.add(g1, fitness)  # G is now a list of genomes with g1 in the list
G.add(g2, fitness)
G.add(g3, fitness)
```

```
G.add(g4, fitness)
G.add(g5, fitness)
#  G=[g1, g2, g3, g4, g5]

# "remove" function: removes a genome from the List
G.remove(g2)
#  G=[g1, g3, g4, g5]

# "random_select" function: unbiased random selection
g = G.random_select()
# random_select selects one object from the list randomly.
# Each object has 25% chance to be selected.
```

Listing 5.3: The `List` code structure.

Two concepts need to be described to understand how the `GENOME` class is working. The first concept is the *dependency vector* (`DV`), which is a vector of bits where each bit represents whether a gene or chromosome is "active", meaning that a change to it can affect the `Active-Output` (O). A bit is set to 1 if the gene or chromosome is active; otherwise, it is set to 0. Three levels of dependency can be used, as shown in figure 5.6. They are described as follows:

1. *Static chromosome-level dependency* is where knowing the `Active-Output` is all that is needed to determine what are the active chromosomes (i.e., cells). In this option, a bit in the `DV` represents a chromosome, which means that a genome has a $(\mathbb{R} \times \mathbb{C})$-bit `DV`. This is the simplest implementation.

2. *Dynamic chromosome-level dependency* is where chromosomes and the `Active-Output` of a genome are needed to allow the algorithm to recursively traverse cells in the `Active-Output` datapath using the functions' dependency defined in table 5.2 along with taking the predefined inter-cell routing into consideration. In this option, a bit in the `DV` represents a chromosome, which means that a genome has a $(\mathbb{R} \times \mathbb{C})$-bit `DV`.

3. *Dynamic gene-level dependency* is similar to dynamic chromosome-level dependency except that a bit in the `DV` is now representing a gene (not a chromosome).

One cell will have 4 genes ($\mathbb{P}$-bit for Sel_f, 2-bit for Sel_x, 2-bit for Sel_y, and 2-bit for Sel_z), which means that a genome has a $(\mathbb{R} \times \mathbb{C} \times \mathbb{P})$-bit `DV`. This is the most sophisticated implementation.

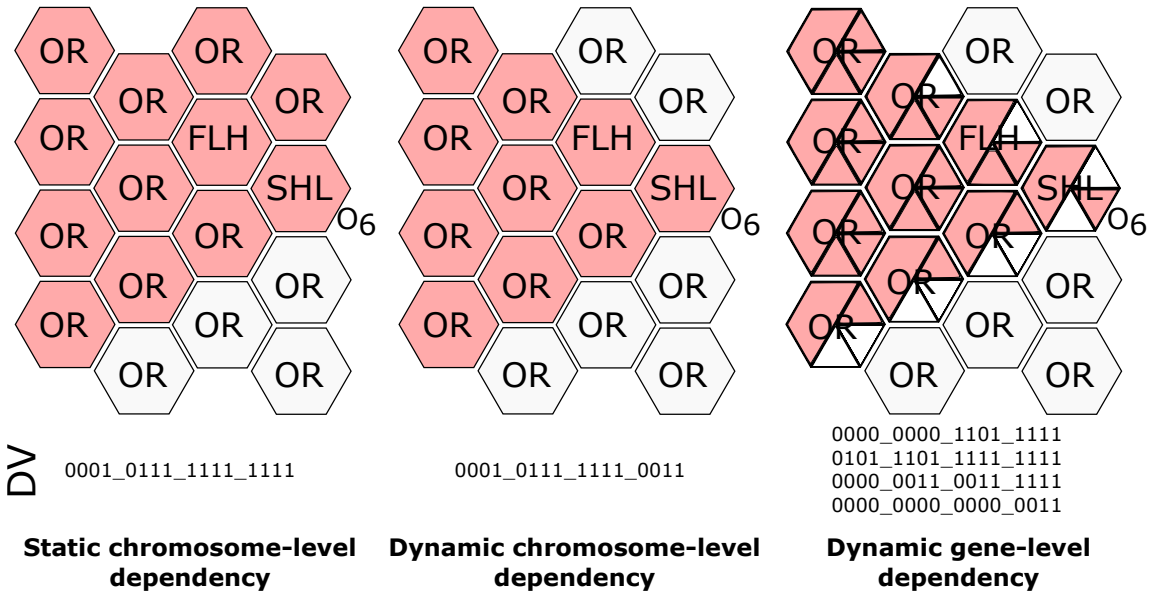


Figure 5.6: HexArray can have different levels of dependencies where (1) the static chromosome-level dependency is unaware of the cells' functional units dependencies, (2) the dynamic chromosome-level dependency is aware of the cells' functional units dependencies, and (3) the dynamic gene-level dependency is aware of the cells' functional units dependencies and the cells' output ports selection.

For our GAGA implementation, dynamic chromosome-level dependency was used because it can be performed without involving much complexity.

The second concept is the "boundbox", which is defined as the smallest rectangular boundary box around all active cells of a selected genome, as shown in figure 5.7. The boundbox is represented by $\langle (x_1, y_1)_{upper-left}, (x_2, y_2)_{lower-right} \rangle$. The "size" function of a boundbox returns $\langle width,\ length \rangle$.

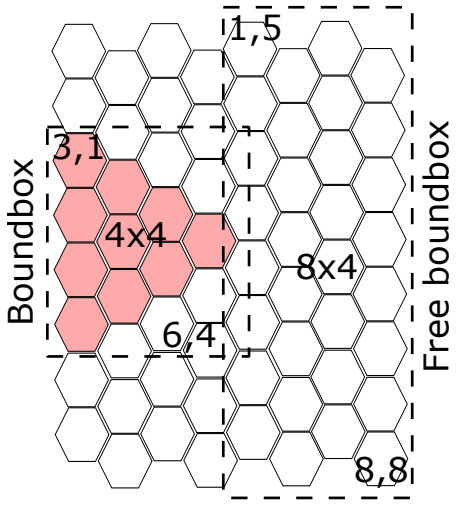


Figure 5.7: Boundbox and free boundbox for a genome of HexArray.

The main code structure used in the GAGA is the `GENOME` class. This class describes genomes and all their aspects, including the `DV` and boundbox. The `GENOME` class constructor and basic functions are explained by the example in listing 5.4.

```
# Assuming 2x2 HexArray with P=4
# genome =<PE22, PE21, PE12, PE11>
# PE = <Sel_z, Sel_y, Sel_x, Sel_f>
# PE is a 2+2+2+4= 10-bit number

# Constructor with no attributes:
G1 = GENOME()     # Initialize an empty genome where chromosomes
                  # are set to 0s and DV is set to 0

G1.get_genome()
# prints: 0x0000000000

G1.set_genome(0x0000000005)
# G1 is now 0x0000000005

# Constructor with genome and Active-Output attributes:
g = 11.10.01.0000_11.10.01.0000_11.10.01.0000_11.10.01.0000
o = 0
G2 = GENOME(g, o)  # Declares a genome (G2) where all functions
                   # are Sel_f=0 (AVG) and Sel_x=B, Sel_y=C
                   # Sel_z=f

G2.get_genome()
```

```
# prints: 0xE4390E4390

# Mutate the lower bit of G2
G2.set_genome( G2.get_genome() ^ 0x1 )
# G2 is now  0xE4390E4391

# Chromosome-level functions
row=1
column=1

chromosome_value = get_chromosome(row, col)
#chromosome_value= 0x391   **Note: chromosome is a 10-bit number

new_chromosome_value=0xF

# Update the chromosome of PE11 to 0xF (00.00.00.1111)
G2.update_chromosome(new_chromosome_value, row, column)
# G2 is now 0xE4390E400F
```

Listing 5.4: Example for how to declare genome objects and use some functions such as get/set a genome/chromosome.

A declaration of a genome object will automatically instantiate an internal object that describes the dependency vector. The DV object is utilized by many functions. An example to explain some of these functions is provided in listing 5.5.

```
g = 11.10.01.0000_11.10.01.0000_11.10.01.0000_11.10.01.0000
o = 0        # The Sel_o=0 means PE11 is the only cell
             # in the Active-Output datapath
G=GENOME(g, o) # Create a genome object

dv=G.DV.get_value() # Return dependency vector value.
# dv=0001           # dv is (RxC-bit value).

                  # (row, col)
u11 = G.is_used_cell( 1,   1 ) # Returns 1
u12 = G.is_used_cell( 1,   2 ) # Returns 0
u21 = G.is_used_cell( 2,   1 ) # Returns 0
u22 = G.is_used_cell( 2,   2 ) # Returns 0

num_used_cells = G.num_used_cells()
# num_used_cells = 1


# randomize_unused_cells:
# -----------------------
#      Assigns a random value for any cell (chromosome) that is
# not in the Active-Output datapath. Note that generated
```

```
# chromosomes do not violate the GLOBAL_RULES.
#

# G  = 11.10.01.0000_11.10.01.0000_11.10.01.0000_11.10.01.0000
# DV = <---- 0 ---->|<---- 0 ---->|<---- 0 ---->|<---- 1 ---->
G.randomize_unused_cells()
# G  = 10.11.10.1001_01.10.11.0110_10.00.11.1010_11.10.01.0000
#    <-Randomized->|<-Randomized->|<-Randomized->|<-  same  ->


ExcludeList = [0,1,2,3]
idx = G.DV.get_rand_index(ExcludeList)
# Returns a random index for bits in cells in use excluding
# bit indices in the ExcludeList.
# idx can be any of 4, 5, 6, 7, 8, 9 and cannot be any of 0, 1, 2, 3
```

Listing 5.5: Explaining some basic functions to obtain the DV value, check if a cell is active, obtain the number of active cells, and obtain an active cell randomly.

The functions discussed thus far are not sufficient for the genome-aware operations. Listing 5.6 shows an example of handling the boundbox. The merge and shift functions will be defined and explained in listing 5.7.

```
G1 = GENOME(g1, o1)
dv1=G1.DV.get_value()     # --> 0000_0000_0001_0111

bb=G1.DV.get_boundbox()   # Returns the boundbox for the genome.
                          # The return object is BB-type.
                          # bb=<(1,1), (2,3)>

fbb=G1.DV.get_free_boundbox() # Returns the largest boundbox around
                              # the free cells. Returned object is
                              # BB-type. fbb=<(3,1), (4,4)>

# Assuming g2.dv2 = 1111_1111_0000_0000
G1.merge_genome(g2, o2)
dv1=G1.DV.get_value()          # dv1 = 1111_1111_0001_0111

bb2=G1.DV.get_boundbox()       # bb2=<(1,1), (4,4)>
fbb2=G1.DV.get_free_boundbox() # fbb2=<(1,4), (2,4)>
```

Listing 5.6: Boundbox versus free Boundbox.

```
# Assuming g1.dv = 0000_0000_1111_1111
# Assuming g2.dv = 0000_0000_0000_1111

G1=GENOME(g1, o1)
G2=GENOME(g2, o2)
```

```
6
7  # Check for intersection by doing bit-wise and (&)
8  intersect=G1.DV.get_value() & G2.DV.get_value()
9  # if intersect = 0, they do not intersect
10 # if intersect = 1, they do intersect
11
12 # g1 and g2 cannot be merged because active cells intersect.
13 fbb1=G1.DV.get_free_boundbox()  # fbb1=<(3,1), (4,4)>
14 fbb1_size=fbb.size()           # Returns <width, length>
15 # fbb1_size: is the largest unused boundbox is 2x4
16
17 bb2=G2.DV.get_boundbox()
18 bb2_size=bb2.size()            # = 1x4
19 # bb2_size: is the boundbox needed for G2.
20
21
22 # Since G1 has unused boundbox of size 2x4 and G2 needs 1x4, merging
23 # G1 and G2 can be done with the shift function
24
25 G2.shift(fbb1)  # Shift G2 around to fit in the free boundbox
26                 # ** Note that this DV is shifted as well
27
28 g2_shifted  = G2.get_value()
29 dv2_shifted = G2.DV.get_value()
30
31 #       g2.dv = 0000_0000_0000_1111  (before shifting)
32 #        fbb1 = 1111_1111_0000_0000  i.e., <(3,1), (4,4)>
33 #               --------------------
34 # dv2_shifted = 0000_1111_0000_0000  (after shifting)
35
36 # Combines g2 to G1 based on the used cells defined by dv2_shifted
37 G1.merge_with_dv(g2_shifted,  dv2_shifted)
38 # g1.dv       = 0000_0000_1111_1111 (before merging)
39 # dv2_shifted = 0000_1111_0000_0000
40 #               --------------------
41 # G1.dv       = 0000_1111_1111_1111 (after merging)
42
43 # Another way of doing this is by combining g1 to G2.
```

Listing 5.7: The process of merging two genomes that do not align.

Finally, the last utility function discussed here is select_parent, which is used to select a parent genome from a list of parents. The function initially filters the given list based on an optional criterion provided by the user. The criterion can be set to select genomes that have at least min_size or at most max_size of active cells or active cells fit into a boundbox

or can fit after shift operation. Some examples are given below. Second, the function utilizes a biased selection based on the roulette wheel method [116], which is performed where the probability of a genome to be selected is proportional to its fitness. In fact, implementing a selection probability that scales with fitness involved some complexity; thus, we constructed a simple selection function that scales with the available number of parents and is biased toward the better ones. The function is defined as follows:

$$index = \left\lfloor (rand() \times \sqrt{\mu}\,)^2 \right\rfloor, \tag{5.8}$$

where $rand()$ is a random floating point number that is normally distributed and smaller than 1 and μ is the number of available parents to choose from. The function returns a random index of the parent genome in a list of μ parents, where the genome with the best fitness is at index 0 and the worst one at index $\mu - 1$; an example is shown in figure 5.8. The method of selecting a parent is described in listing 5.8.

```
# Genomes List
GS=[(g1,o1), (g2,o2), (g3,o3), (g4,o4), (g5,o5)]

# Format:
#   g, o = select_parent(<Genomes List>, <filter>)

# filter is optional and can be inhibited
g, o = select_parent(GS)      # Returns any of the genomes in the
                              # list according to probabilities

# filter can be min_size
g, o = select_parent(GS, min_size=6) # Returns any genome that
                                     # has 6 or more active cells.
                                     # Function returns null if
                                     # no genome was found.

# filter can be max_size
g, o = select_parent(GS, max_size=6) # Returns any genome that
                                     # has 6 or less active cells.

# filter can be boundbox
g, o = select_parent(GS, bb=<(3,1), (4,4)>)
```

```
                        # Returns any of the genomes that fit in the
                        # given bb without shifting

# filter can be boundbox size
g, o = select_parent(GS, bb_size=1x4) # Returns any genome that
                                      # has a width <= 1  AND
                                      # a length <= 4
```

Listing 5.8: The *select_parent* function filters a list and performs a biased selection based on fitness.

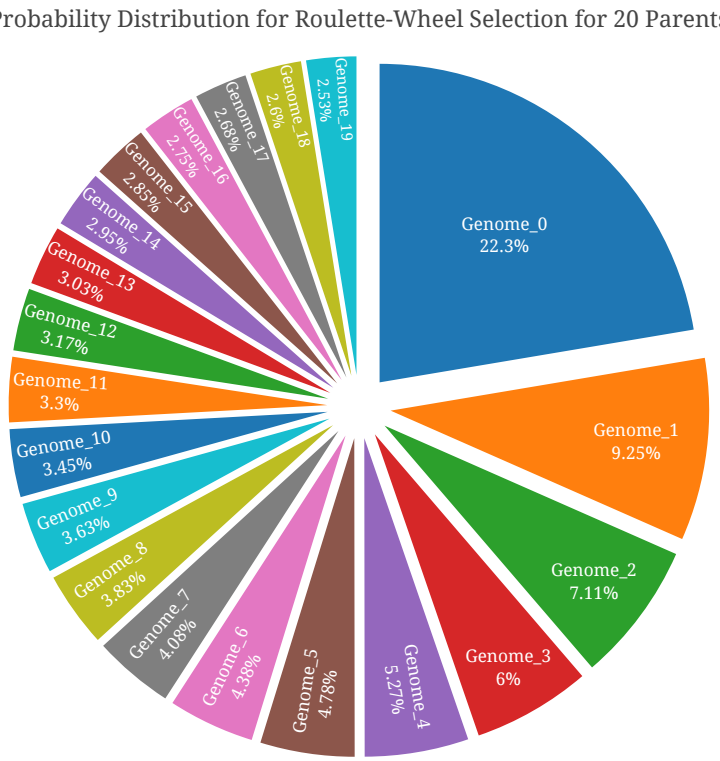


Figure 5.8: The probability distribution for selecting a genome out of 20 parents. Because Genome_0 is the parent with the best fitness, it has the highest chance (22.3%) of being selected. Genome_19 is the one with the worst fitness (compared to others); thus, it has the lowest chance (2.5%).

In the following subsections, the GAGA operators will be presented, which are genome-aware constrained selection, genome-aware mutation, and genome-aware crossover, in addition to the new GA operator, genome-aware pruner. The target of these improvements

is to accelerate evolution and increase parallelism without limiting the GA from exploring the entirety of the search space.

5.4.2 Genome-Aware Constrained (GAC) Selection

GAC is a set of rules (using `TEMPORARY_RULES`) that are assigned by the user or GAGA to accomplish the following goals:

1. Reduce redundant evaluations: For example, the Sel_x, Sel_y, and Sel_z for $PE_{4,4}$, as shown in figure 5.4, should have different selections. In other words, forcing the signals Sel_x to select A, Sel_y to select B, and Sel_z to select f eliminates redundant evaluations and reduces the search space by 6 bits.

2. Reduce the genome string length (practically reduce search space): For example, the two bits needed for Sel_x of $PE_{1,1}$ should be fixed to f as other selections will route an input pixel, not an interesting output. This constraint reduces the search space by 2 bits. However, Sel_x of $PE_{1,3}$ can be constrained to be C or f, which further reduces the search space by 1 bit.

3. Improve probabilities of routing the less-fortunate ports of the array based on statistical analysis, as shown in figure 5.9: For example, O_5 and O_6, figure 5.4, are forced to not select A as A has a good chance of being routed through (O_3, O_4) and (O_9, O_{10}, O_{14}).

4. Support virtual resizing of the systolic array: For example, forcing the cells of the last row and last column to bypass the outputs of the cells of the second-to-last row and second-to-last column would virtually make the array one row/column smaller.

The pseudo code for GAC selection is given in algorithm 3.

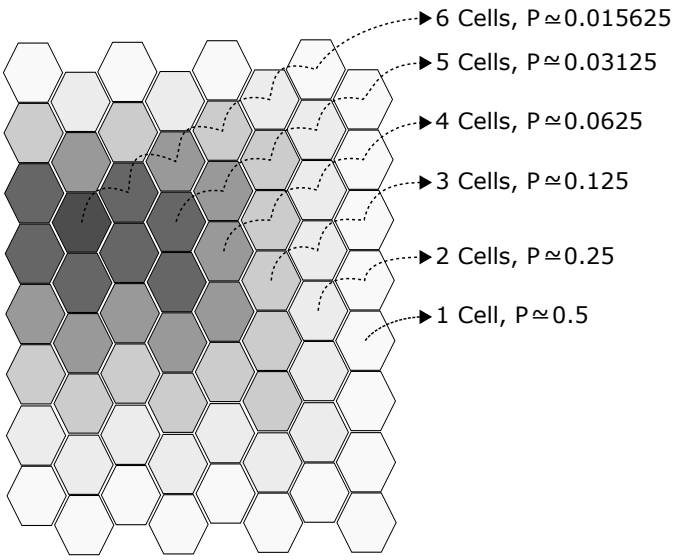


Figure 5.9: Estimated probability for routing certain functions to the closest array outputs. The darker the cell is, the lower the chance is for f to reach an output. For example, the probability value of 0.5 was obtained from the probability of f being routed through X or Y $(0.25 + 0.25)$.

5.4.3 Genome-Aware Mutation (GAM)

GAM is an effective mechanism for performing bit mutation. For a selected genome with a specific `Active-Output`, the GAGA recursively traverses the `Active-Output` datapath and identifies the dependent cells and performs mutation only on these cells. Mutation can be for M_{GAM} bits, defined by the user. All cells that do not affect the `Active-Output` are changed randomly to increase the evolution efficiency. The GAM method is presented in algorithm 4.

5.4.4 Genome-Aware Crossover (GAX)

GAX is a genetic operator that generates an offspring by combining two or more selected genomes. Three methods of combining are defined: cascade, interleave, and parallel,

Algorithm 3: Pseudo code for genome-aware constrained selection – GAC selection.

```
Result: GAGA returns a constrained genome (g_GAC)
g_GAC=GENOME()
for row ← 1 to ℝ do
    for col ← 1 to ℂ do
        chromosome = get_rand_chromosome(row, col)
        g_GAC.update_chromosome(chromosome, row, col)
    end
end
return g_GAC
```

Algorithm 4: Pseudo code for genome-aware mutation – GAM changes M_{GAM} bits in the `Active-Output` datapath and randomizes other bits.

```
Result: GAGA returns a mutated genome (g_GAM)
g_s, o_s = select_parent(G_parents)
g_GAM =GENOME(g_s, o_s)
g_GAM.randomize_unused_cells()
excludingList= [ ]
for i ← 1 to M_GAM do
    bit_index= g_GAM.DV.get_rand_index(excludingList)
    excludingList.add( bit_index)
    g_GAM = g_GAM ⊕ (1 ≪bit_index)
end
return g_GAM
```

which are summarized in figure 5.10. In all modes, any cell that is not active in any of the combined genomes receives a random chromosome value.

Cascade mode

In cascade mode the hypothesis is that cascading two (or more) good genomes can yield an additive improvement. The method is defined as the output of one genome is fed as an input into another genome. This method clearly requires relatively short genomes[1]. Algorithm 5 explains the procedure for GAX-Cascade.

[1] Short genome means a genome with a short active datapath.

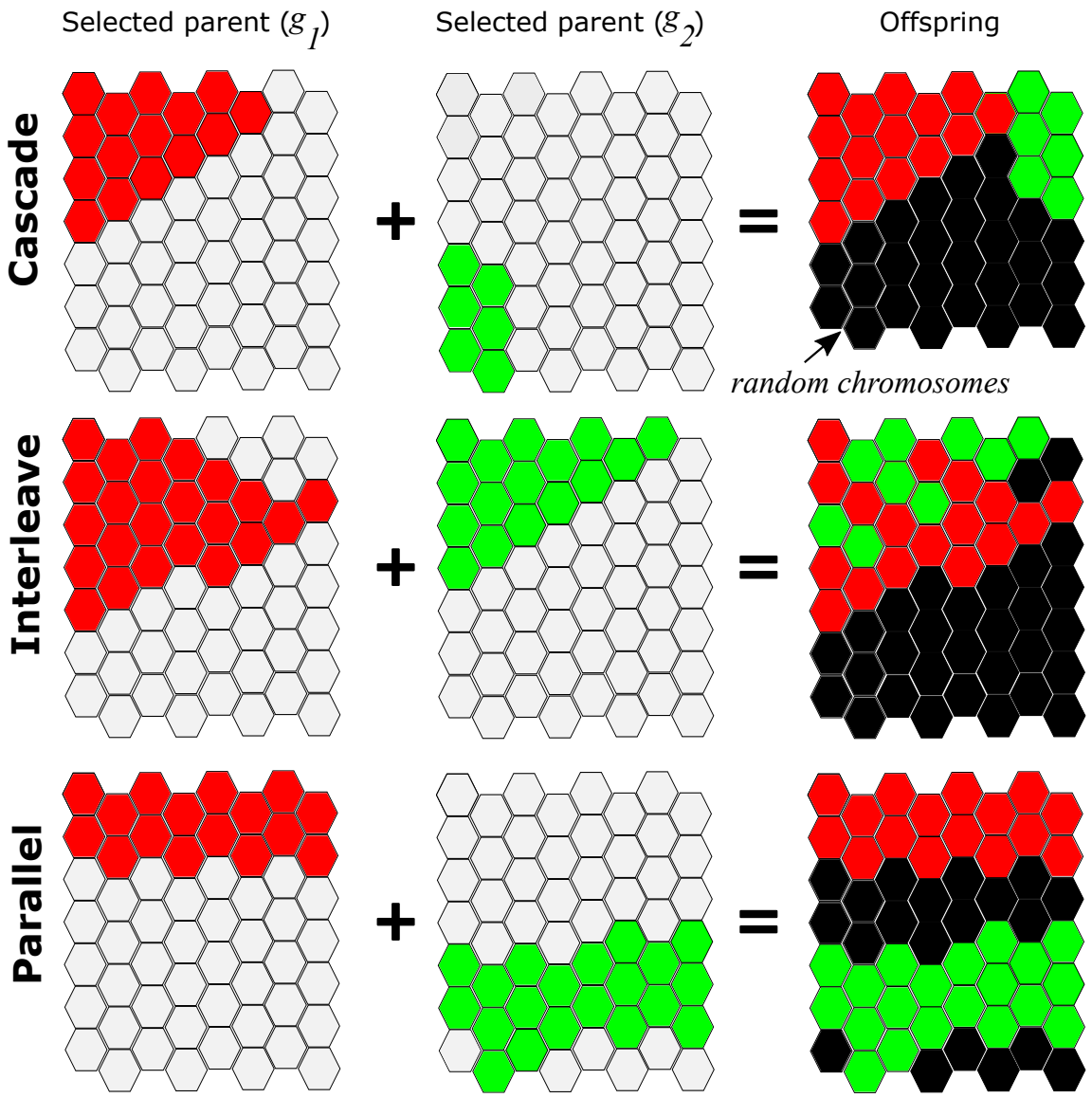


Figure 5.10: GAX modes. (Top) An offspring is generated by cascading genomes, where one feeds into the other. (Middle) An offspring is generated by interleaving genomes at the cell level. (Bottom) An offspring is generated by combining genomes in parallel and inserting some cells in-between with randomly selected functions.

Algorithm 5: Pseudo code for genome-aware crossover running in cascade mode – GAX-Cascade.

Input: SHORT_GENOME := $\frac{\mathbb{R}\times\mathbb{C}}{4}$
Output: G_{GAX}: a recombined genome

1 $SG_{parents}$=$filter\_parents(G_{parents},\ max\_size$ =SHORT_GENOME)
2 $g_s,\ o_s = select\_parent(SG_{parents})$
3 G_{GAX} =GENOME($g_s,\ o_s$)
4 $G_{GAX}.randomize\_unused\_cells()$
5 BB_{free}=$G_{GAX}.DV.get\_free\_boundbox()$
6 $G_{candidates} = filter\_genomes(SG_{parents}$, bb=$BB_{free})$
7 **if** $(G_{candidates} = [\,])$ **then**
8 | **return** G_{GAX} ▷ No genome fits in the remaining BB
9 **end**
10 $g_i,\ o_i = select\_parent(G_{candidates})$
11 G_i =GENOME($g_i,\ o_i$)
12 $G_i.shift(BB_{free})$
13 $g_{shifted}$ =$G_i.get\_value()$
14 $dv_{shifted}$ =$G_i.DV.get\_value()$
15 $G_{GAX}.merge\_with\_dv(g_{shifted}, dv_{shifted})$
16 **go to** 5 ▷ Try more cascading

Interleave mode

In interleave mode the theory is that merging chromosomes of good genomes can result in an improved hybrid genome. The method is a simple interleaving of cell-level chromosomes of two or more selected genomes. To allow effective cell interleaving, it is recommended to use genomes with more than $\frac{\mathbb{R}\times\mathbb{C}}{4}$ cells being active. Algorithm 6 shows the strategy for GAX-Interleave.

Parallel mode

Finally, in parallel mode the operator combines the genomes in parallel while leaving some cells in-between. These cells, which have random chromosomes, will have a chance to incorporate intermediate cell outputs of the merged genomes to create better offspring. The

Algorithm 6: Pseudo code for genome-aware crossover running in interleave mode – GAX-Interleave.

Input: LONG_GENOME := $\frac{\mathbb{R}\times\mathbb{C}}{4}$
THRESHOLD := $\mathbb{R}\times\mathbb{C}$
Output: G_{GAX}: a recombined genome

```
1  LG_parents=filter_parents(G_parents, min_size =LONG_GENOME)
2  k = 0
3  used_cells = 0
4  while (used_cells ⩽ THRESHOLD) do
5  |  k++
6  |  g_k, o_k = select_parent(LG_parents)
7  |  G_k =GENOME(g_k, o_k)
8  |  used_cells +=G_k.num_used_cells
9  end
10 G_GAX=GENOME()
11 G_GAX.randomize_unused_cells()
12 for row ← 1 to ℝ do
13 |  for col ← 1 to ℂ do
14 |  |  Candidate_Chromosomes = [ ]
15 |  |  for i ← 1 to k do
16 |  |  |  if G_i.is_used_cell(row, col) then
17 |  |  |  |  c=G_i.get_chromosome(row, col)
18 |  |  |  |  Candidate_Chromosomes.add(c)
19 |  |  |  end
20 |  |  end
21 |  |  if (Candidate_Chromosomes ≠ [ ]) then
22 |  |  |  c_sel = random_select(Candidate_Chromosomes)
23 |  |  |  G_GAX.update_chromosome(c_sel, row, col)
24 |  |  end
25 |  end
26 end
27 return G_GAX
```

recommended genome size for this method is to not be too short or too long. Algorithm 7 provides the methodology for GAX-Parallel.

Algorithm 7: Pseudo code for genome-aware crossover running in parallel mode – GAX-Parallel.

Input: LONG_GENOME := $\frac{\mathbb{R}\times\mathbb{C}}{4}$
THRESHOLD := $\frac{3\times\mathbb{R}\times\mathbb{C}}{4}$
Output: G_{GAX}: a recombined genome

```
1  LG_parents=filter_parents(G_parents, min_size =LONG_GENOME)
2  g_s, o_s = select_parent(LG_parents, max_size =THRESHOLD)
3  G_GAX=GENOME(g_s, o_s)
4  G_GAX.randomize_unused_cells()
5  while (G_GAX.num_used_cells < THRESHOLD) do
6  |  BB_free=G_GAX.DV.get_free_boundbox()
7  |  G_i, O_i=select_parent(LG_parents, bb=BB_free)
8  |  G_GAX.merge_genome(G_i, O_i)
9  end
10 return G_GAX
```

5.4.5 Genome-Aware Pruner (GAP)

GAP is a new operator that can alter the genome structure without affecting its functionality. It is defined as a mechanism of generating different genomes with identical functionality; note that for a selected genome, only one output is active and altering other outputs has no effect. GAP is unique to HexArray as it is an artifact of HexArray being an array of identical cells and having flexible routing that can provide more than one route to a targeted function.

GAP can be used as a step in other genetic operations, such as shifting genomes for GAX, where in this case the shifting does not affect the genome output, in contrast to what was proposed earlier where cells are shifted while array inputs are not. Another use of

GAP is to reduce power consumption by shutting off[2] non-used functional units. Finally, the operator can be used to condense genomes by eliminating chromosomes of bypassed cells.

5.5 Overall System Workflow

In this section, the overall workflow of evolution in HexArray will be presented first, followed by the full sequence of evaluating a genome. Algorithm 8 presents the GAGA flow control at a higher level; all used functions and genetic operations were described earlier. GAGA supports elitism, in addition to other common genetic operations. Elitism is supported by simply allowing the all-time-best genomes to compete with children of the current generation to create new parents for the next generation. Since GAX operations have restrictions on genomes' length, they can potentially fail to generate offspring. In these cases, GAGA uses GAC selection to generate random genomes instead. All subprocesses of the GAGA algorithm are running on the PS as software, except the "evaluate" function.

"Evaluate" is the function responsible for evaluating genomes, that is, *"returning a fitness value for a given genome"*. The function is executed in hardware, and the execution process is identical for all genomes regardless of how they are generated or their content. In this sense, studying the workflow of a single genome is adequate for understanding the overall system behavior.

To describe the process of evaluating a genome, the system initial status and specifications need to be outlined first. A high-level anatomy of the HexArray platform is shown in figure 5.11, where a 4×4 HexArray is implemented in hardware, with 15 AICs

[2]Shutting off a functional unit is done by sending a blank partial bitstream to clear the dynamic partition content.

Algorithm 8: Pseudo code for genome-aware genetic algorithm (GAGA).

Result: GAGA returns the fittest genome

$g_{GAC} = g_{GAM} = g_{GAX\_C} = g_{GAX\_I} = g_{GAX\_P} = null$
$G_{Elite} = [\,]$
$generation = 0$
while *not termination condition* **do**
  if *generation=0* **then**
    for $i \leftarrow 1$ ***to*** *population* **do**
      $g_{GAC_i} = call\ GAC\_selection$ ▷ as described in algorithm 3
      $F_{GAC_i} = Evaluate(g_{GAC_i})$
    end
  else
    for $i \leftarrow 1$ ***to*** *(population*$\times m\_rate$*)* **do**
      $g_{GAM_i} = call\ GAM$ ▷ as described in algorithm 4
      $F_{GAM_i} = Evaluate(g_{GAM_i})$
    end
    for $i \leftarrow 1$ ***to*** *(population*$\times c\_rate/3$*)* **do**
      $g_{GAX\_C_i} = call\ GAX_{Cascade}$ ▷ as described in algorithm 5
      $F_{GAX\_C_i} = Evaluate(g_{GAX\_C_i})$
      $g_{GAX\_I_i} = call\ GAX_{Interleave}$ ▷ as described in algorithm 6
      $F_{GAX\_I_i} = Evaluate(g_{GAX\_I_i})$
      $g_{GAX\_P_i} = call\ GAX_{Parallel}$ ▷ as described in algorithm 7
      $F_{GAX\_P_i} = Evaluate(g_{GAX\_P_i})$
    end
    for $i \leftarrow 1$ ***to*** *(population*$\times(1-(m\_rate + c\_rate))$*)* **do**
      $g_{GAC_i} = call\ GAC\_selection$
      $F_{GAC_i} = Evaluate(g_{GAC_i})$
    end
  end
  $G_{children} \Leftarrow G_{Elite} + [g_{GAC_i} \cdots,\ g_{GAM_i} \cdots,\ g_{GAX\_C_i} \cdots,\ g_{GAX\_I_i} \cdots,\ g_{GAX\_P_i} \cdots\ ]$
  $G_{parents} \Leftarrow [\,]$
  for $i \leftarrow 1$ ***to*** *parents* **do**
    $g_s = get\_fittest(G_{children})$
    $G_{parents}.add(g_s)$
    $G_{children}.remove(g_s)$
  end
  $G_{Elite} \Leftarrow G_{parents}$
  $generation++$
end

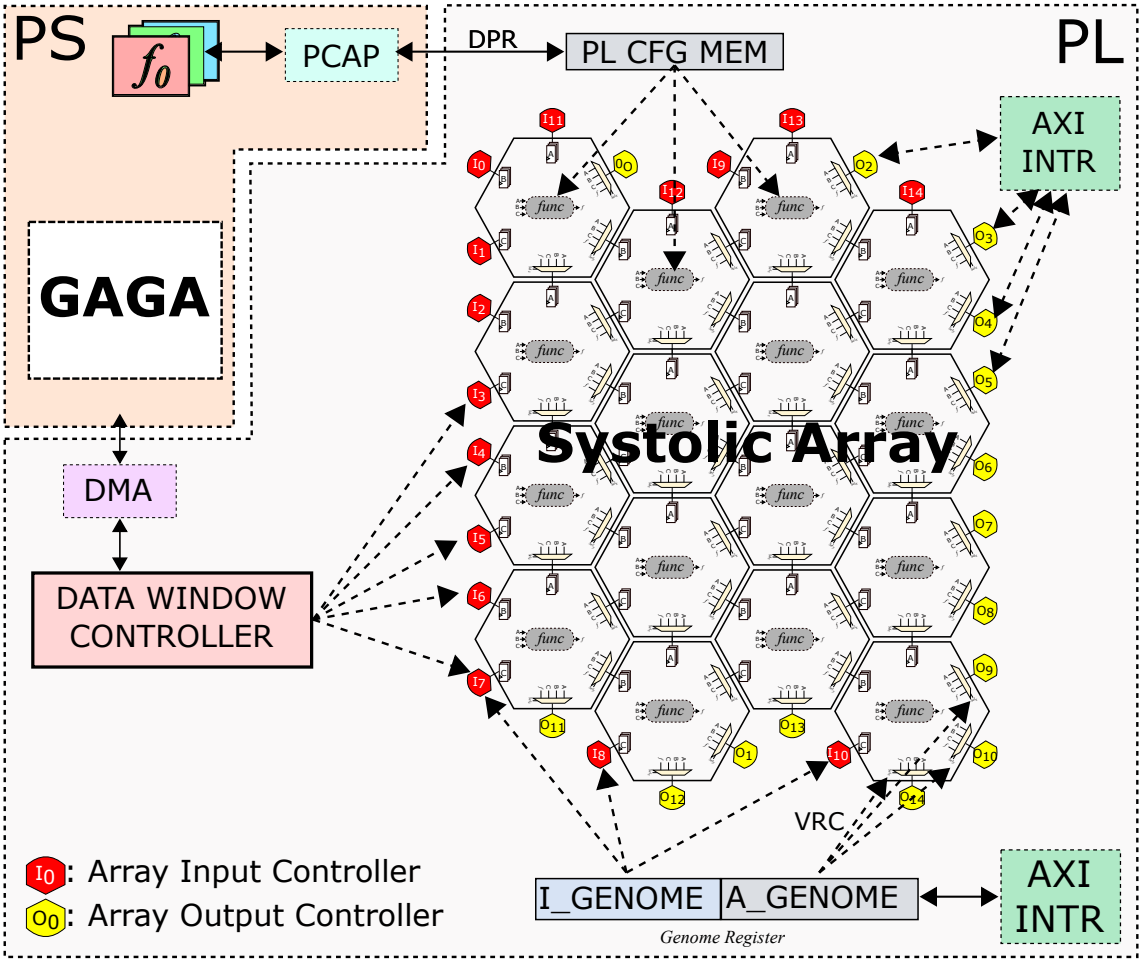


Figure 5.11: HexArray platform with HexArray and GAGA. HexArray, array input controllers, array output controllers, data widow controller, and genome register are implemented on the FPGA programmable logic, while the GAGA is implemented on the processor.

and 15 AOCs. All functional units of the HexCells (16 of them) are initially blank (not programmed to any function) or programmed for a previous genome. Partial bitstreams are loaded into memory; assuming that $\mathbb{P} = 4$, there will be 16 of them in addition to the blank bitstream. Buffers and counters of all units are assumed to be cleared initially. The GAGA is running on the PS. It is generating the next genome in parallel with evaluating the previous one. Training and reference images are selected by the user. PS creates and loads in memory a stream of pre-formatted data, including a sliding 8-byte vertical window of data from both images and the middle pixel coordinates. In other words, for the pixel at (x,y), the vertical window is $\langle$ Train$_{(x-2,y)}$, Train$_{(x-1,y)}$, Train$_{(x,y)}$, Train$_{(x+1,y)}$, Train$_{(x+2,y)}$, Reference$_{(x,y)}$, x, and y $\rangle$, as shown in figure 5.3. PS sends an AXI transaction to all AOCs to set the image size (e.g., for a 256×256 image, expected count=65,536). PS sends an AXI transaction to the data window controller to inform it of the image size and enable it to get ready to pass data to the array. However, data are not sent to this module yet.

Now, assume that $g_i = \langle$I_GENOME, A_GENOME$\rangle$ need to be evaluated. PS initializes the DMA for future transactions. PS determines the difference between the current state of the array functional units and what is defined in A_GENOME. For those different functional units, PS sends DPR transactions through PCAP to reprogram them. Note that while a PCAP transaction is executing, the PS is free. PS sends an AXI transaction to set the genome register using I_GENOME and A_GENOME. The array routing is now completed as AICs and HexCell output ports are configured by A_GENOME. Upon the completion of the DPR, PS requests the DMA to send the stream of pre-formatted data to the data window controller.

The data window controller stacks the 8-byte vertical data window into shift registers to create an 8x5-byte data window. For every pixel in the training image, this unit sends a ready signal to all AICs every time that a data window is formatted. AICs of boundary HexCells select a pixel from the data window based on the predefined I_GENOME and

inform the functional cell that data are available. When all data needed by a functional cell are ready, the execution is started, and the resulting data are sent to the next HexCell downstream. The data propagation is shown in figure 5.12.

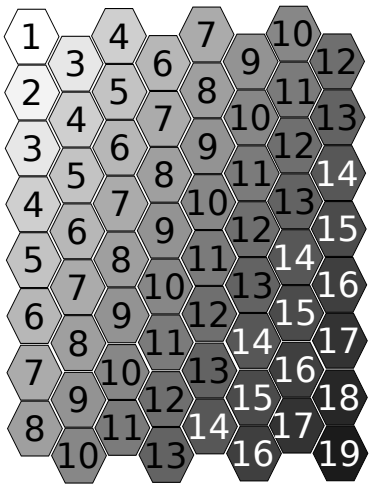


Figure 5.12: Data propagation in HexArray, where a pixel is processed by $PE_{1,1}$ at time 1 and by $PE_{8,8}$ at time 19.

An AOC, connected to every array output, is calculating the fitness by accumulating the absolute difference between the generated pixel and the reference pixel, which is provided by the data window controller as part of the data window. As the AOC executes the evolved pixels, it also counts them. After the last pixel in the image, the AOC counter will match the "expected count", resulting in pushing the fitness value into an FIFO and triggering the PS. On a trigger, the PS issues an AXI transaction to read fitness values. At this point, the evaluation is completed and the GAGA can perform evolution.

5.6 Additional Features

5.6.1 A Novel Fault Detection and Tolerance Mechanism

The routing flexibility and parallelism features in HexArray are exploited to enable a fault detection and tolerance mechanism. The targeted fault type is physical damage to the fabric of a dynamic partition. In other words, the fault is a persistent failure in the HexCell functional unit and not in the static circuit. This is a reasonable assumption because the static circuit of a HexCell is typically considerably smaller than the dynamic functional unit.

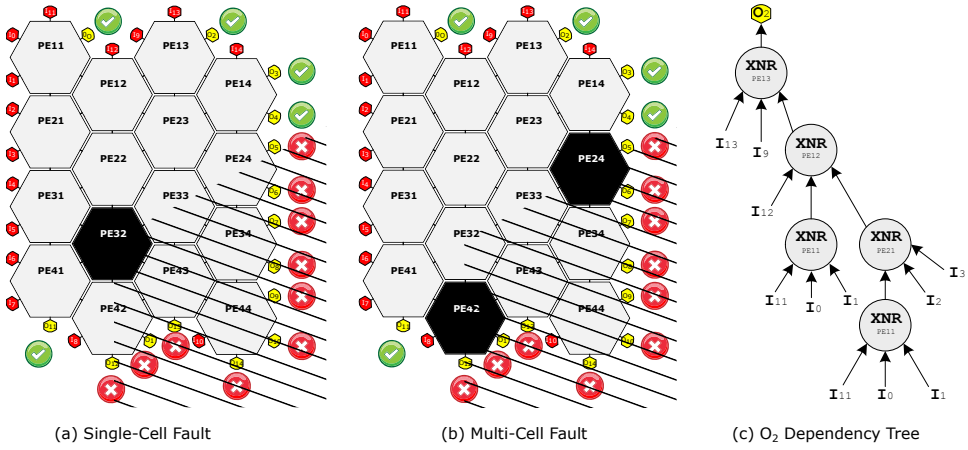


Figure 5.13: (a) Single-cell fault results in unexpected outputs. (b) Multi-cell fault results in unexpected outputs. (c) Example of an output dependency tree, where any fault in $PE_{1,1}$, $PE_{1,2}$, $PE_{2,1}$, and $PE_{1,3}$ will affect the output.

The proposed fault detection mechanism is simply to verify that the system generates the expected data using a predefined genome. This genome is designed such that its outputs are functionally dependent on all functional units in a way that any change to the behavior of a (defective) cell will affect all outputs downstream and yield unexpected results on these outputs, as shown in figure 5.13.a. It is critical to engineer this genome such that

any deviation by the functions in the datapath is reflected in the output. An example of the dependency tree for O_2 is shown in figure 5.13.c; the suggested function is `XNR` here. This mechanism works perfectly if there is one faulty cell. However, depending on this technique alone is not sufficient because the output of the array with two faulty cells may look like that resulting from one faulty cell, as shown in figure 5.13.b.

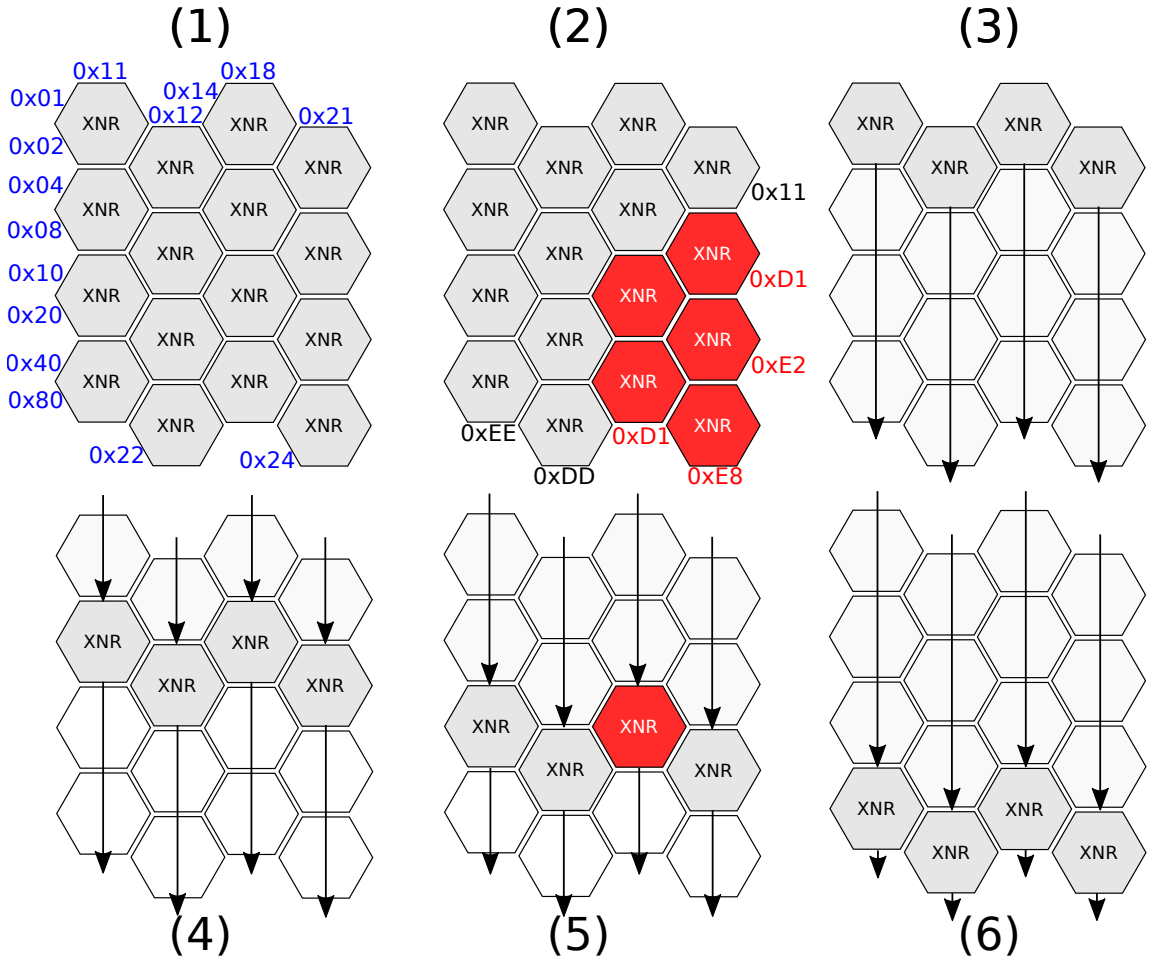


Figure 5.14: Example for fault detection mechanism using row by row testing where the array output of a predefined genome is checked against a pre-calculated output. If the outputs are matching, then the circuit is fault-free. If the outputs are not matching, then the circuit has one or more faulty cells. A row by row test is needed to determine which cells are faulty.

To expand the fault detection mechanism to detect multi-cell faults, an extensive cell by cell testing needs to be performed; an example for that is presented in figure 5.14 with 6 steps. A suitable test case to achieve this concept is that in which all cells have the same chromosomes equal to $\langle Sel_z = f$, $Sel_y = f$, $Sel_x = f$, $Sel_{func} =$XNR $\rangle$), and the input data of every cell port is unique, as shown in steps (1) and (2), respectively. The PS expects specific data on every output port. If one or more outputs are not matching, then a fault is detected. For example, at step (2), five cells are suspected. To determine which one (or more) is faulty, a row by row testing (i.e., one row is active and all others are bypassed) will be performed, as shown in steps (3), (4), (5), and (6). Since at step (5) the third output is failing, $PE_{3,3}$ is the faulty cell.

The fault tolerance mechanism also exploits the routing flexibility feature of HexArray to route around the faulty cell. The algorithm updates its PERMANENT_RULES to enforce this routing. In the previous example, to recover from that fault, the following rules are added to PERMANENT_RULES: $\langle Sel_z = [A,B,C]$, $Sel_y = [A,B,C]$, and $Sel_x = [A,B,C] \rangle_{PE_{3,3}}$. In the evolution process, the selected genomes that use the faulty cell will be re-structured by the GAGA automatically as the utility functions work with the GLOBAL_RULES in place. Note that in some cases, the defect does not affect all input combinations of a functional unit, for example, a function that is partially working. In this case, the amount of test cases needs to provide a full coverage of all bit permutations. This is clearly a trade-off between speed and reliability that the user/application can define. An overall flowchart of the fault detection and tolerance mechanism is presented in figure 5.15.

5.6.2 A Novel Evolve-while-Reconfigure Mechanism

Another feature that is gained by the flexibility offered by HexArray is evolving while some cells are under DPR. When the DPR process is active, the array is not ready to be used,

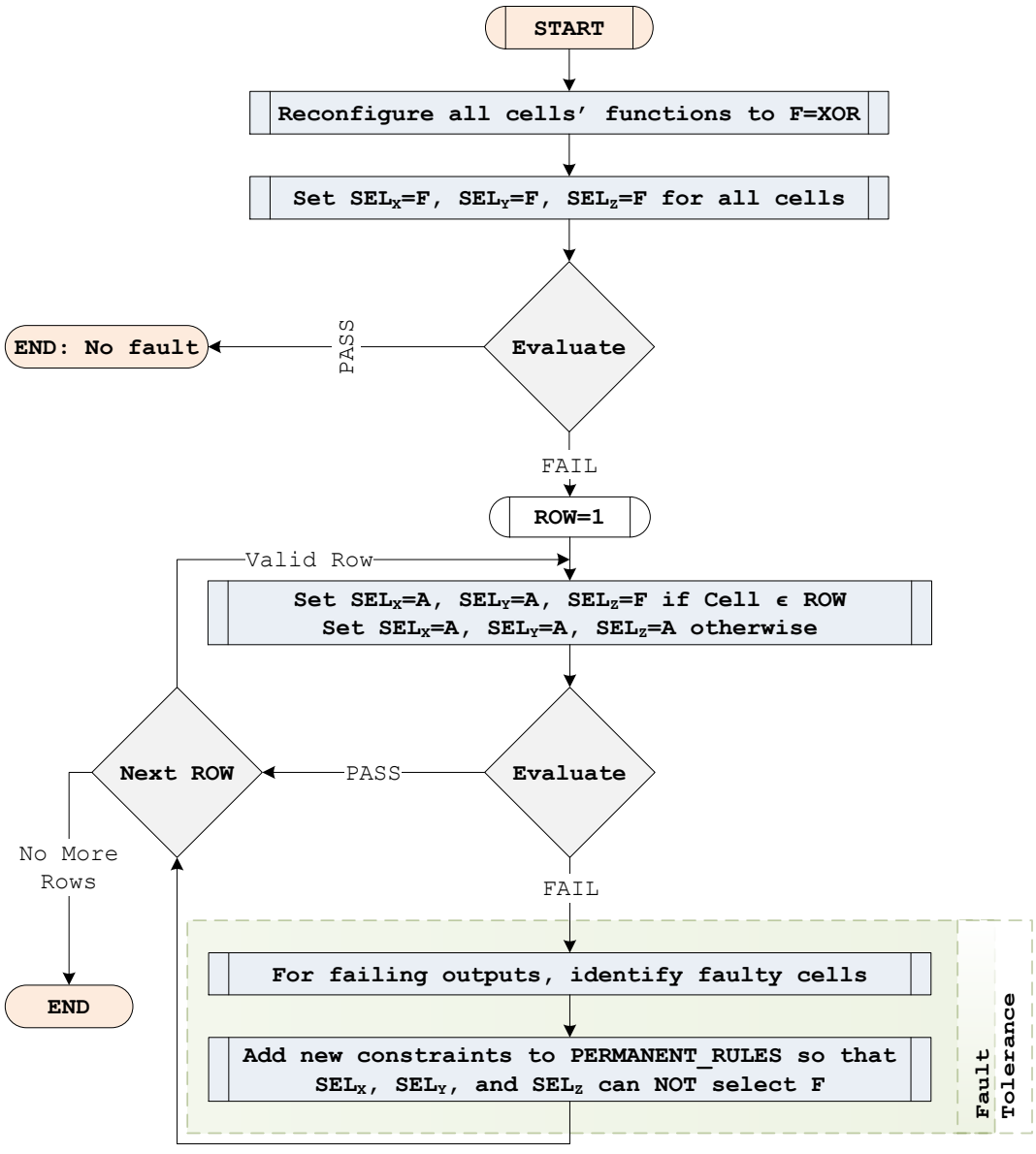


Figure 5.15: Flowchart for the proposed fault detection and tolerance mechanism.

or at least the cells that are being programmed cannot be used. Therefore, if these cells are bypassed, the array can be used. The concept of evolve-while-reconfigure is that rather than leaving the array idle while reconfiguration is in process, a genome (different than the target one) can be evaluated.

Because the time for evaluating a genome is longer than the time for reconfiguring a cell (e.g., 1 to 10), GAGA sets up the `TEMPORARY_RULES` to bypass a group of cells under reconfiguration, as shown in figure 5.16. Note that not all cells need to be reconfigured because some functions do not change between two genomes and others are not used (the case where none of the cell's ports are routing f). For an 8×8 HexArray implemented on Zynq assuming a 1 to 10 time ratio between a single cell DPR and a genome evaluation, GAGA can theoretically evaluate 7 genomes before evaluating the targeted one. Note that no time penalty is attached to this mechanism.

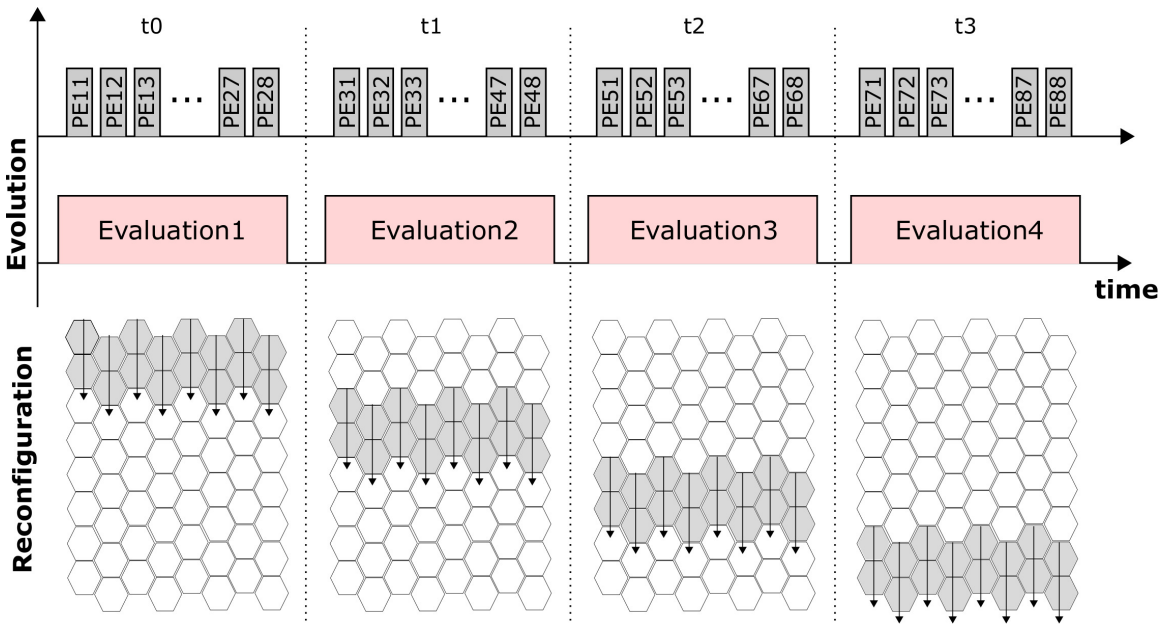


Figure 5.16: Running on "evolve-while-reconfigure" mode, where the evaluation occurs while some of the cells are being programmed (shown in dark gray).

5.7 HexArray Versus State-of-the-Art Systolic Array

HexArrays have many advantages over the traditional Cartesian systolic arrays (presented in [30] and other publications [173, 3]), such as having a hybrid reconfiguration scheme, more inputs/outputs, higher parallelism, higher throughput, and can evolve to higher-order functions (i.e., has the potential to solve more complex problems). Cells of the Cartesian systolic arrays are called RectCells for short, and table 5.3 summarizes the properties of HexArray in comparison with Cartesian array with RectCells.

Table 5.3: HexArray in comparison to Cartesian Array with RectCell.

Property	Cartesian Array	HexArray
Probability of DPR	100%	$100 \times \frac{\mathbb{P}}{\mathbb{P}+6}\%$
Probability of VRC	0%	$100 \times \frac{6}{\mathbb{P}+6}\%$
Input-ports count	$(\mathbb{R}+\mathbb{C})$	$2 \times (\mathbb{R}+\mathbb{C})-1$
Output-ports count	1	$2 \times (\mathbb{R}+\mathbb{C})-1$
Neighboring Cells	4	6
Data Propagation	E, S	NE, SE, S
Shortest Data Path	$\mathbb{C}$	1
Longest Data Path	$(\mathbb{R}+\mathbb{C})-1$	$(\mathbb{R}+\mathbb{C})-1+\lfloor\frac{\mathbb{C}}{2}\rfloor$
Degree of Polynomial	$\frac{1}{3}e^{1.35\mathbb{R}}$	$\frac{1}{6}e^{1.95\mathbb{R}}$
Array Throughput	f_{system}	$f_{system} \times [(\mathbb{R}+\mathbb{C})-\frac{1}{2}]$
Cell Latency(cycles)	1	2

5.7.1 Degree of Polynomial

The algebraic term "degree of a polynomial" can be described as the highest order of a function, and it can represent, in our scope, the maximum number of functions and combinations that an array can yield. To simplify the following calculations, symmetric arrays can be assumed (where $\mathbb{R} = \mathbb{C} = \mathbb{L}$). How the degree of the polynomial grows in both arrays is shown in figure 5.17.

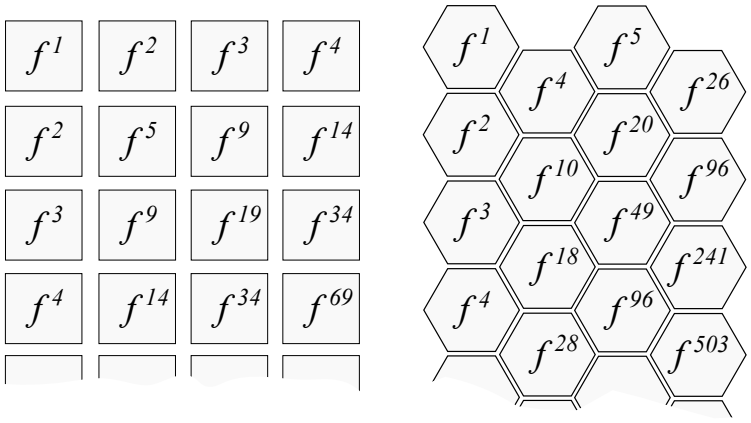


Figure 5.17: Degree of polynomial of HexArray is higher than Cartesian arrays.

The exponential fit of the degree of polynomial for both arrays is plotted in figure 5.18. When solving these two equations, the results show that a 10×10 RectCell array is functionally equivalent to a 7×7 HexCell array. This result is somewhat expected because of the three-operator functions and the added output multiplexers.

5.8 Summary

The contributions of this dissertation are presented in this chapter. The discussion of the HexArray simulator, which served initially as a proof of concept, serves as a high-level description of the HexArray platform. The simulator, which is a software model, is functionally equivalent to the proposed hardware model. However, as evolution is a slow process, the software model can not achieve acceptable performance and a hardware module is needed. The hardware implementation can boost the speed by more than $1000\times$. The proposed system includes a novel reconfigurable hardware core and an enhanced genetic algorithm.

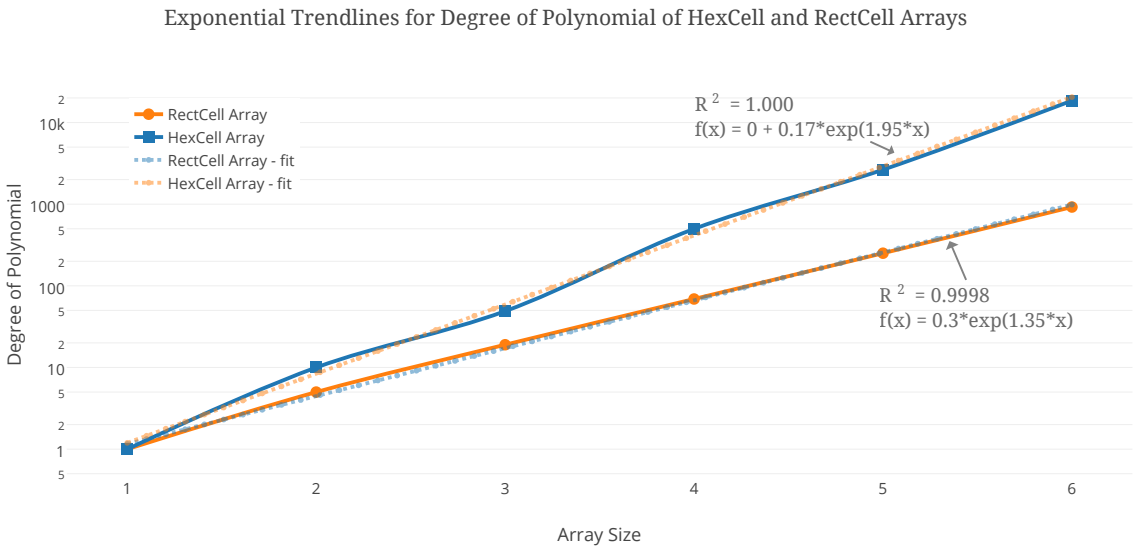


Figure 5.18: Fitting the degree of polynomial of HexArray and state-of-the-art systolic array.

The reconfigurable hardware core is a systolic array, called HexArray, which is constructed using a new processing element called HexCell. HexCell is a virtual hexagonal cell with three buffered input ports, three output ports, and a functional unit. The output ports are data selectors that route one of the input ports or the output of the functional unit. The functional unit is built on a dynamic partition that can be programmed to realize any function from a predefined function set. The hardware representation of the cell is called the phenotype of the cell, while the logical representation of it is the genotype, a string of bits in this case. The genotype of HexCell is a $(\mathbb{P}+6)$-bit chromosome with four genes, three 2-bit genes (for output port selection signals) and a $\mathbb{P}$-bit gene (for selecting the function used in the functional unit). The output port genes reflect a VRC process, while the functional unit gene reflects a DPR process. An $\mathbb{R}\times\mathbb{C}$ HexArray, which is a collection of cells (chromosomes), is represented logically by an $(\mathbb{R}\times\mathbb{C}\times(\mathbb{P}+6))$-bit genome. In addition to the systolic array itself, HexArray requires other modules; these are

(1) data window controller responsible for "systoling" input data into the array, (2) genome register responsible for storing the current genome data, (3) AICs responsible for selecting a specific pixel in a window of pixels around the target pixel, and (4) AOCs for "sinking" the generated data and measuring the fitness.

The enhanced genetic algorithm is a genome-aware genetic algorithm. GAGA has a library of low-level functions used to achieve higher-level operations. The algorithm performs context-aware genetic operations; these are GAC selection, GAM, and GAX. GAC selection is an operator that replaces the "totally-random" genome generation in the canonical GAs with a mechanism that constrains the randomness in a way that does not limit the ability of the GA from searching the entire solution space. GAM is a mechanism to limit mutation to the active chromosomes of a genome. GAX is a chromosome-level crossover that has three modes of operation: (1) cascade, which allows genome outputs to feed into another genome input; (2) interleave, which allows a random swapping at the chromosome level; and (3) parallel, which merges genomes in a way that permits in-between cells to interact with them. Elitism is achieved by the parent selection process as it allows *all* genomes to compete for the parents' "seats" in the next generation. GAP is a new operator that can condense genomes to save power by turning off functional units that are not in use.

Additional features are enabled by the new architecture; these are the fault detection/-tolerance mechanism and evolve-while-reconfigure mechanism. Fault detection is achieved by testing the array using a known test set (genome/output); if the produced data are not expected, then one or more faulty cells exist and a row by row testing needs to be performed. This is achieved by utilizing the flexible routing of HexArray to bypass other rows. Fault tolerance is simply achieved by adding a constraint to ban the use of the faulty cells; GAGA's functions work under these rules.

The concept of evolve-while-reconfigure is that a genome that has some of its HexCells bypassed is still worth evaluating. Thus, cells that are under DPR are bypassed, and the resulting genome is evaluated concurrently.

The last part of this chapter provides a comparison between HexArray and the state-of-the-art systolic array. HexArray has many advantages, including having (1) more inputs and outputs, (2) a hybrid reconfiguration scheme, (3) wider data propagation, (4) variable datapath (e.g., variable search space size), (5) higher throughput, and (6) higher degree of polynomial (e.g., has the potential to solve harder problems).

CHAPTER 6

EVALUATIONS AND IMPLEMENTATION ANALYSIS

6.1 Introduction

In this chapter, a set of experiments have been designed and conducted to validate certain assumptions about the proposed platform. These experiments were designed to achieve the following:

1. Demonstrate the efficiency of the new HexArray hardware architecture.

2. Demonstrate the effectiveness of the genome-aware genetic operators.

3. Demonstrate the adaptive behavior of the system.

4. Demonstrate the performance of the overall system.

5. Demonstrate the ability of the system to evolve autonomously.

The discussions in the next section are limited to the acceleration of the evolution convergence; that is, for a fixed number of genomes, what is the best fitness obtained? The implementation, time analysis, and resource utilization of the system will be discussed in section 6.3.

6.2 Evolution Speed Evaluations

For the evaluations, training and reference images with a variety of noise types and feature extraction types were used. Appendix A includes all image groups used in the experiments. All images are in gray-scale format and 256×256 pixels in size. The "Group Name" is used to uniquely address certain training sets.

In most of the following experiments, unless otherwise stated, an 8×8 HexArray was used, which was implemented in hardware. The array had 31 outputs, where each was fed into an AOC implemented in hardware to calculate the fitness value online. A normalized MAE fitness function was used, which is described as follows:

$$MAE_{Norm} = \left(\frac{1}{WL} \sum_{m=1}^{m=W} \sum_{n=1}^{n=L} |Out(m,n) - Ref(m,n)| \right) \times \frac{1}{MAE_{Init}}, \tag{6.1}$$

where W and L are the image width and length, respectively; $Ref(m,n)$ is the reference pixel; $Out(m,n)$ is the evolved output pixel; and MAE_{Init} is the initial fitness value (mean absolute error for the training image). MAE_{Norm} equals 1.000 for an evolved image with no improvement (e.g., evolved image = training image), and it equals 0.000 for an evolved image that matches the reference image. Clearly, the smaller the fitness value is, the better the output is. For all collected data points, three significant figures were considered. Unless otherwise stated, most of the experiments used 5 image groups, as shown in table 6.1.

Table 6.1: A collection of image groups used in the experiments.

Group Name	Training Image	Reference Image
S&P 25%	25% impulsive noise	Original image (without noise)
S&P 10%	10% impulsive noise	Original image
EdgeDetect	Original image	Laplacian of Gaussian filter
Thresholding	Original image	Simple thresholding filter
Gaussian	Gaussian noise	Original image

Seven experiments were conducted and will be described in the following subsections. Each experiment was repeated multiple times to increase confidence and allow statistical analysis. For each experiment, we collected the best (minimum) solution of each run (iteration) and plotted them using statistical box plots [174] to visualize the data distribution. For comparisons between two or more data groups, the median fitness and the best fitness were mainly used.

6.2.1 Experiment 1: HexArray Outperforms State-of-the-Art Systolic Array

Evolution was expected to be accelerated using HexArray. This expectation was due to three reasons (features) in the new architecture: *routing flexibility* offered by the HexCells, *improved parallelism* and *"wider" data propagation* offered by the HexArray architecture. This experiment was designed to explore the acceleration achieved by HexArray over the state-of-the-art systolic array (the traditional Cartesian array of RectCells) proposed by Sekanina et al. [30]. To understand and assess the effect of each of HexArray's features mentioned earlier, the experiment was extended by improving the routing flexibility and parallelism of the Cartesian arrays. The wide data propagation feature, however, is a unique feature of HexArray and could not be modeled on Cartesian arrays.

Hypotheses

1. HexArray can evolve faster than a traditional Cartesian array.

2. HexArray can evolve faster than a Cartesian array with modified RectCells (with flexible output ports).

3. HexArray can evolve faster than a traditional Cartesian array with improved parallelism.

4. HexArray can evolve faster than a Cartesian array with modified RectCells (with flexible output ports) and improved parallelism.

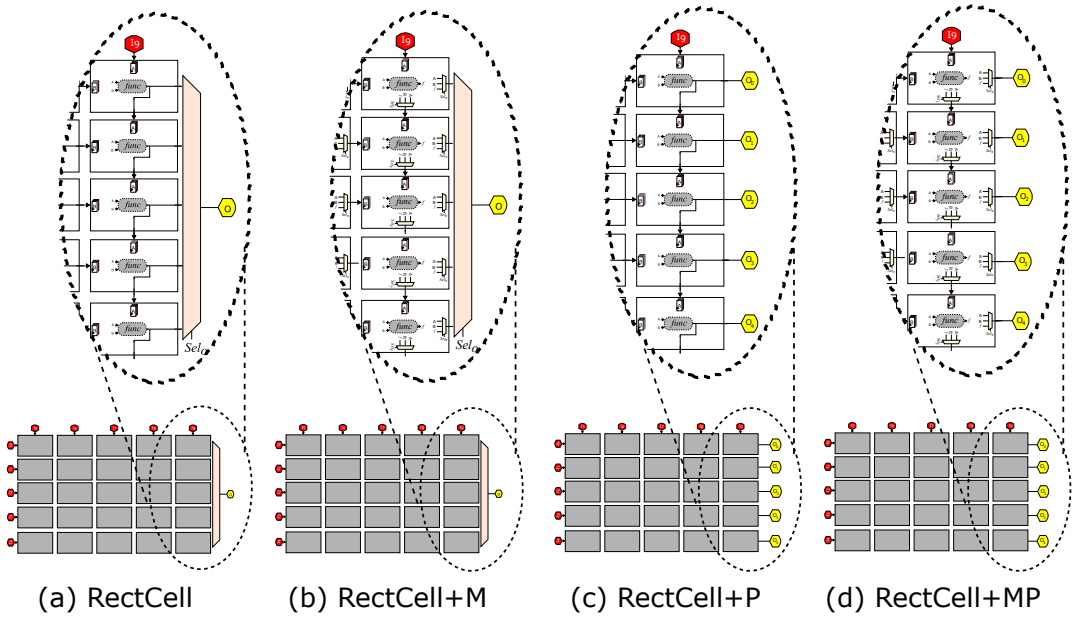


Figure 6.1: (a) A Cartesian array constructed using classical RectCells. One output is evaluated per genome (based on Sel_O), and a cell functional output is routed to the E and S ports. (b) A Cartesian array constructed using "modified" RectCells, where an output multiplexer has been added to every cell output to select from the N port, W port or the functional unit output. (c) and (d) are similar to (a) and (b), respectively, but with evaluating five outputs per one genome.

Experimental Setup

The experiment was conducted using 5 image sets selected from the library of data sets in table 6.1. For each image group, using a 3×3 window size, we evaluated 1000 randomly generated genomes and selected the best fitness obtained. We repeated the process 100 times.

The experiment was conducted on the following hardware architectures:

1. 5×5 Cartesian array with traditional RectCells (figure 6.1.a, called **RectCell**).
2. 5×5 Cartesian array with modified RectCells (figure 6.1.b, called **RectCell+M**).
3. 5×5 Cartesian array with traditional RectCells and improved parallelism (figure 6.1.c, called **RectCell+P**).
4. 5×5 Cartesian array with modified RectCells and improved parallelism (figure 6.1.d, called **RectCell+MP**).
5. 5×5 HexArray.

Results

For each image group and each hardware architecture, the best fitness of the generated solutions (filters) for each iteration was collected, resulting in 100 data points per image group per hardware architecture. For these data points, considering their normalized fitness values described earlier, the calculated median and best fitness for all combinations of image groups and hardware architectures are summarized in table 6.2. The box plot for the collected data is shown in figure 6.2.

Discussion

As shown in figure 6.2, the median fitness for the solutions generated by HexArray were always better than those generated by the RectCell array. The best solutions generated by HexArray were also better than those generated by RectCell in most image groups. For the "Thresholding" image group, the best fitness obtained by HexArray was 0.046 higher (worse) than RectCell's best fitness. This was a small difference that can be ignored, particularly since the data distribution indicated that RectCell's best fitness was an outlier.

Table 6.2: Summary of the best and median fitness values collected for evaluation of Cartesian arrays based on traditional RectCells and modified RectCells and HexArray.

	RectCell	**RectCell+P**	**RectCell+M**	**RectCell+MP**	**HexArray**
	Median Fitness				
S&P 25%	0.768	0.678	0.744	0.678	0.669
S&P 10%	0.853	0.69	0.818	0.698	0.643
EdgeDetect	0.536	0.514	0.545	0.538	0.525
Thresholding	0.556	0.377	0.619	0.575	0.535
Gaussian	0.994	0.953	0.979	0.934	0.809
	Best (min) Fitness				
S&P 25%	0.613	0.615	0.583	0.481	0.467
S&P 10%	0.514	0.566	0.582	0.546	0.459
EdgeDetect	0.495	0.486	0.492	0.481	0.181
Thresholding	0.208	0.133	0.387	0.217	0.254
Gaussian	0.919	0.819	0.786	0.767	0.747

Another method to validate that it was an outlier is to consider the second-best solution in both arrays and verify that RectCell is still outperforming HexArray. The second-best solutions were similar for both arrays, which indicates that we can ignore this case.

A 5×5 RectCell array can have 2^{100} genomes (25 cells with 4-bit chromosomes) while 2^{250} (25 cells with 10-bit chromosomes) for HexArray, which is a considerable size difference between the two search spaces that could slow evolution. However, the evolution was faster using HexArray. This result may appear unexpected, but it can be explained as follows. (1) Although the search space was increased, the number of good solutions was also increased. (2) HexArray has multiple outputs (19 outputs for 5×5 HexArray), where each output has a search space with a specific size (as shown in figure 6.3). In general, the HexArray architecture outperforms the state-of-the-art systolic array.

To explore the effect of the *routing flexibility* on RectCell and determine whether it is the one feature that could bridge the gap, we conducted the experiment using the RectCell+M array, as shown in figure 6.2.b. For 60% of the image groups, RectCell+M reported better

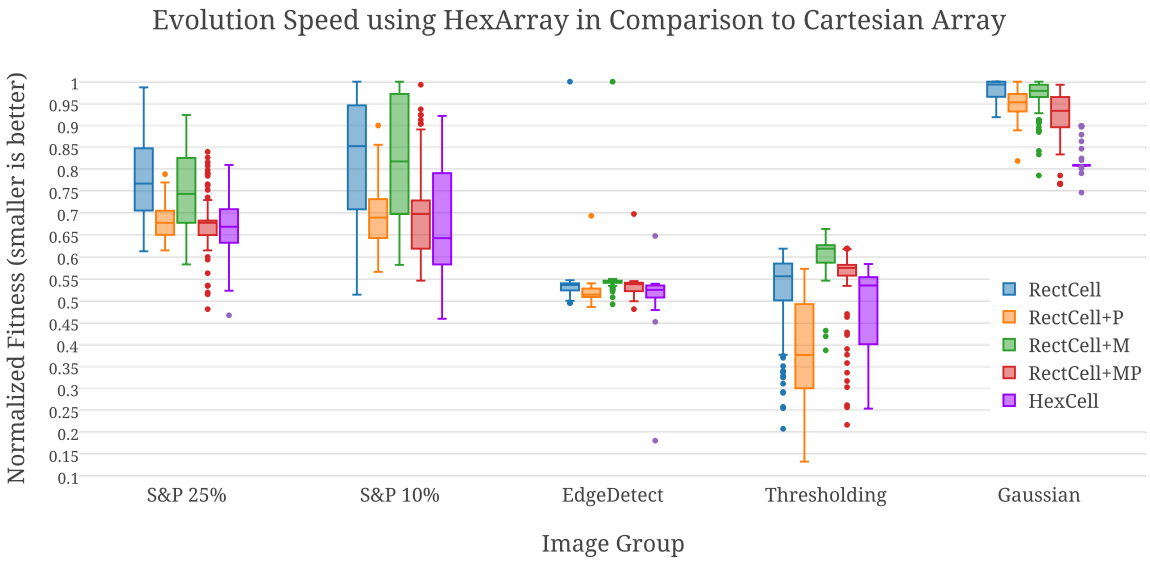


Figure 6.2: Evolution is accelerated by the HexArray architecture in comparison to Cartesian arrays with traditional RectCells and with modified RectCells. Adding parallelism to the Cartesian arrays appears to improve the quality of generated solutions more than adding the output multiplexers.

best and median fitnesses than RectCell, which indicated that the improvement was not significant. This might be due to the lack of parallelism and/or the reduced probability of routing the cells' functional outputs since the output port could select data from A, B, or f, which means that each has a $\frac{1}{3}$ chance of being selected. Therefore, the probability of bypassing the cell's functional output is $\frac{2}{3}$, which resulted in reducing the complexity of the generated solutions. For example, consider the case where the output selection of a genome is 0 (e.g., $Sel_O = 0$, as shown in figure 6.2.b); then, the probability of routing the "raw" input data (I_9) directly to the output will be $\frac{1}{3}$, which is high for a not interesting solution. Additionally, for the same example with $Sel_O = 0$, if the problem required a solution with 5 functional units (i.e., the degree of polynomial=5), then the probability of satisfying that would be very low ($\frac{1}{3^5}$= 0.004). In general, adding routing flexibility to RectCells contributed to a moderate acceleration of the evolution process.

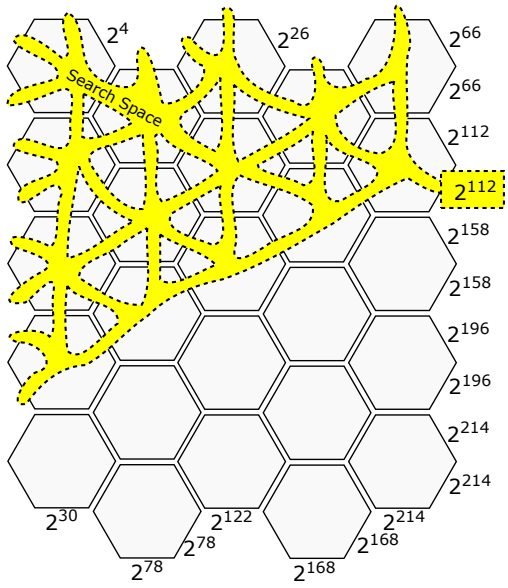


Figure 6.3: HexArray has multiple (different sizes) search spaces. The highlighted output (O_7) has a search space size of 2^{112}.

To evaluate the effect of improving *parallelism*, we tested the evolution process using RectCell+P and RectCell+MP arrays. This modification resulted in better solutions for almost all the cases. This result was expected, because five outputs per genome were evaluated rather than one. In general, parallelism can consistently improve the evolved solutions.

HexArray reported improved best solutions in 85% of the cases and improved medians in 90% of the cases compared with all four Cartesian array architectures. The cases where HexArray did not outperform others were mostly from one image group (Thresholding), which might indicate unique circumstances that are problem specific that made RectCell evolve better. Overall, the combination of the improved parallelism, routing flexibility, and data propagation features of the HexArray architecture contributed to accelerating the evolution and generated better quality filters in comparison with Cartesian arrays.

6.2.2 Experiment 2: GAC Selection Accelerates Evolution

In the previous chapter, GAC selection techniques were proposed to generate better genomes that can accelerate evolution. This experiment was designed to quantify the value of using GAC selection versus the traditional random selection in HexArrays. The used GAC is shown in figure 6.4, which reduces the search space from 2^{640} to 2^{568}.

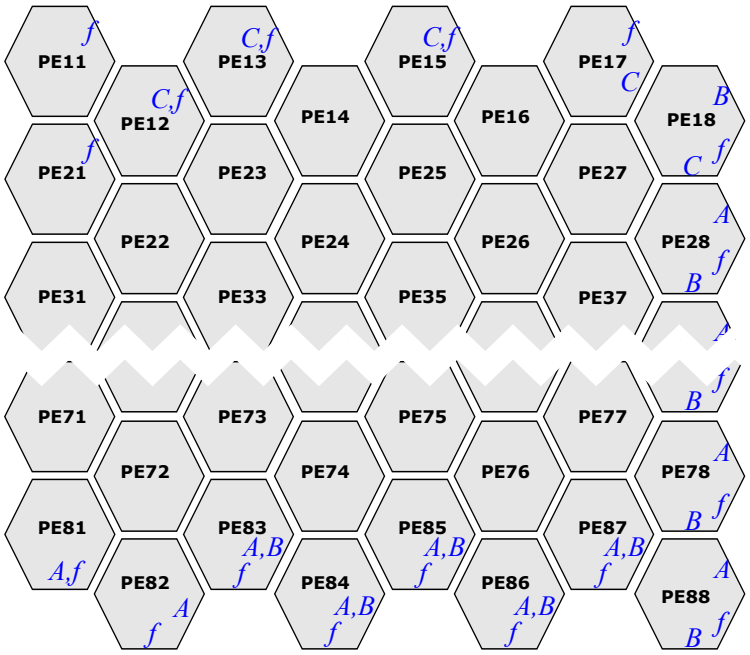


Figure 6.4: GAC for an 8×8 HexArray where the search space is reduced by 2^{72}.

Hypotheses

1. Evolution is accelerated using GAC selection in contrast to the classical unconstrained random selection.

Experimental Setup

The experiment was conducted using the same 5 image groups used in the previous experiment. For each image group, using a 5×5 data window, 1000 genomes were evaluated, and the best fitness was selected. This procedure was repeated 100 times. The experiment was conducted on the following platforms:

1. 8×8 HexArray with randomly generated genomes.

2. 8×8 HexArray with GAC selection.

Results

For each image group and each hardware architecture, we collected the best solution generated for each iteration. The collected data are summarized in table 6.3 and plotted in figure 6.5.

Table 6.3: Data were collected by running 100 iterations of 1000 genomes generated by unconstrained random selection in comparison to GAC random selection.

Image Group	**S&P 25%**	**S&P 10%**	**EdgeDetect**	**Thresholding**	**Gaussian**
	Median Fitness				
Random	0.691	0.712	0.507	0.387	0.809
GAC Selection	0.676	0.64	0.503	0.331	0.809
	Best (Min) Fitness				
Random	0.518	0.429	0.453	0.18	0.771
GAC Selection	0.455	0.508	0.445	0.145	0.785
	Standard Deviation				
Random	0.056	0.128	0.016	0.085	0.04
GAC Selection	0.061	0.116	0.015	0.087	0.032

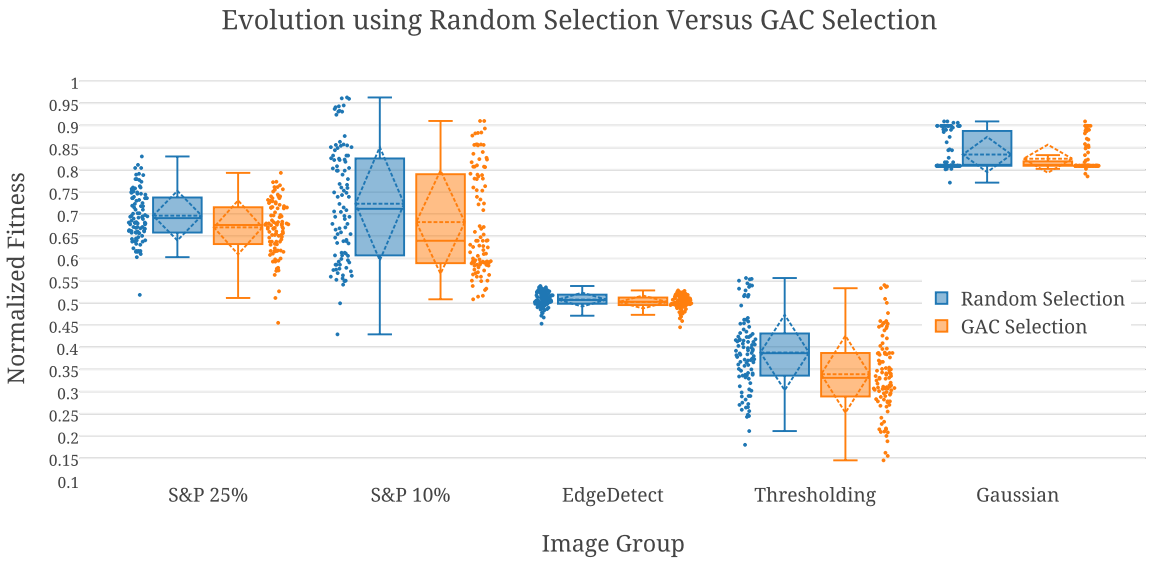


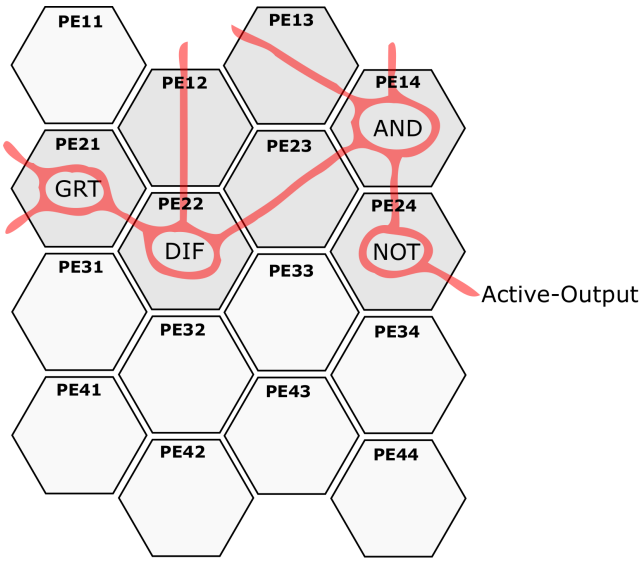
Figure 6.5: A side-by-side comparison of the evolution results generated by randomly selected genomes versus GAC selected genomes by running 100 iterations with 1000 genomes. Dashed lines show the data mean and standard deviation. Generated solutions were improved by GAC selection.

Discussion

In four image groups, GAC selection reported improvements in the generated solutions' medians ranging from 0.4% to 7.2%. Since the fifth image group (Gaussian) showed no improvement for the median, we examined the mean and the standard deviation of the generated solutions and observed small improvements on both, from 0.834 ± 0.040 to 0.825 ± 0.032. Additionally, the generated best solutions were better using GAC selection in three image groups. For the other two image groups, if we considered the second-best solutions, then they would be within $\pm 1\%$ of each other. This is insignificant to the gain that GAC selection is providing in the overall population. This improvement was expected due to the size reduction of the search space. In general, evolution using GAC improves the generated genome populations in many aspects.

6.2.3 Experiment 3: GAM Outperforms Traditional Mutation

The mutation operator plays a significant role in maintaining the genetic diversity of populations in GA [175]. In the previous chapter, we proposed GAM, an improved mutation operator for HexArray, which restricts the mutation to a subset of bits contributing to the Active-Output; figure 6.6 presents an example.




```
                Genome: PE11 PE12 PE13 PE14 PE21 PE22 PE23 PE24 PE31 PE32 ... PE44
Mutation-Enabled Bits: ---- MMMM MMMM MMMM MMMM MMMM MMMM MMMM ---- ---- ... ----
```

Figure 6.6: An example of GAM where mutation is restricted to a subset bits of the genome, where "M" means mutation is allowed and "-" means it is not allowed.

Hypotheses

1. Classical mutation on HexArray yielded the best genomes when $M_{mutation}$ bits are used.

2. GAM on HexArray yielded the best genomes when M_{GAM} bits are used.

3. Evolution using 1-bit GAM converges faster than that using 1-bit classical mutation.

4. Evolution using M_{GAM}-bit GAM converges faster than that using $M_{mutation}$-bit classical mutation.

Experimental Setup

The experiment was conducted using 5 image groups. For each image group, using a 5×5 window size, we evaluated 10 generations with 1000 genomes each and selected the best fitness obtained. We repeated the process for 100 iterations. The experiment was conducted using an 8×8 HexArray with traditional mutation with $M_{mutation} \in \{1\text{-}bit, 2\text{-}bit, ..., 9\text{-}bit\}$ and using GAM with $M_{GAM} \in \{1\text{-}bit, 2\text{-}bit, ..., 9\text{-}bit\}$.

Results

For each image group and each hardware architecture, we collected the best solution generated for each iteration. The solutions of the first generation were generated using GAC selection, and afterward, 100% of the generated genomes were by mutation (i.e., $m\_rate = 1.0$). The median and best fitnesses of the collected data are summarized in table 6.4 and table 6.5, respectively. Mutation with different $M_{mutation}$ is shown in figure 6.7, and GAM with different M_{GAM} is shown in figure 6.8. GAM using 1-bit and the best M_{GAM}-bit versus traditional mutation using 1-bit and the best $M_{mutation}$-bit are shown in figure 6.9.

Discussion

For traditional mutation, as shown in figure 6.7, a significant[1] improvement was observed by increasing the mutation from 1 to 2 bits, and slight improvements were observed for every increase up to 7 bits. This was expected because the probability of mutating a bit (or

[1]The term significant used in this dissertation for its general meaning and it does not mean statistically significant.

Table 6.4: Median normalized fitnesses were collected by running 100 iterations of 10,000 genomes generated by traditional mutation in opposition to GAM with different numbers of mutation bits.

	Image Group	**S&P 25%**	**S&P 10%**	**EdgeDetect**	**Thresholding**	**Gaussian**
		Median Fitness				
Traditional Mutation	$M_{mutation}$ *=1 bit*	0.735	0.754	0.507	0.342	0.861
	2 bits	0.638	0.593	0.495	0.287	0.809
	3 bits	0.621	0.594	0.493	0.277	0.809
	4 bits	0.621	0.604	0.493	0.272	0.809
	5 bits	0.618	0.591	0.493	0.27	0.809
	6 bits	0.629	0.586	0.49	0.264	0.809
	7 bits	0.607	0.587	0.489	0.252	0.809
	8 bits	0.626	0.594	0.492	0.269	0.809
	9 bits	0.614	0.591	0.493	0.265	0.809
GAM	M_{GAM} *=1 bit*	0.597	0.565	0.487	0.24	0.809
	2 bits	0.597	0.56	0.487	0.243	0.809
	3 bits	0.6	0.563	0.486	0.251	0.809
	4 bits	0.581	0.567	0.485	0.236	0.809
	5 bits	0.581	0.562	0.486	0.251	0.809
	6 bits	0.595	0.553	0.486	0.242	0.809
	7 bits	0.598	0.564	0.486	0.233	0.809
	8 bits	0.593	0.561	0.484	0.24	0.809
	9 bits	0.582	0.564	0.488	0.234	0.809

Table 6.5: Best normalized fitnesses were collected by running 100 iterations of 10000 genomes generated by traditional mutation in opposition to GAM with different numbers of mutation bits.

	Image Group	**S&P 25%**	**S&P 10%**	**EdgeDetect**	**Thresholding**	**Gaussian**
		Best (min) Normalized Fitness				
Traditional Mutation	$M_{mutation}$ *=1 bit*	0.548	0.44	0.475	0.205	0.78
	2 bits	0.414	0.424	0.43	0.143	0.771
	3 bits	0.465	0.47	0.437	0.139	0.788
	4 bits	0.481	0.468	0.441	0.154	0.785
	5 bits	0.468	0.406	0.449	0.149	0.789
	6 bits	0.432	0.421	0.448	0.139	0.754
	7 bits	0.457	0.412	0.424	0.133	0.779
	8 bits	0.418	0.346	0.46	0.139	0.788
	9 bits	0.408	0.456	0.436	0.139	0.788
GAM	M_{GAM} *=1 bit*	0.458	0.339	0.449	0.139	0.778
	2 bits	0.419	0.258	0.44	0.14	0.748
	3 bits	0.46	0.381	0.429	0.139	0.765
	4 bits	0.397	0.47	0.404	0.136	0.739
	5 bits	0.448	0.343	0.446	0.136	0.756
	6 bits	0.42	0.396	0.441	0.139	0.763
	7 bits	0.419	0.389	0.431	0.15	0.771
	8 bits	0.428	0.407	0.441	0.139	0.781
	9 bits	0.435	0.33	0.442	0.139	0.732

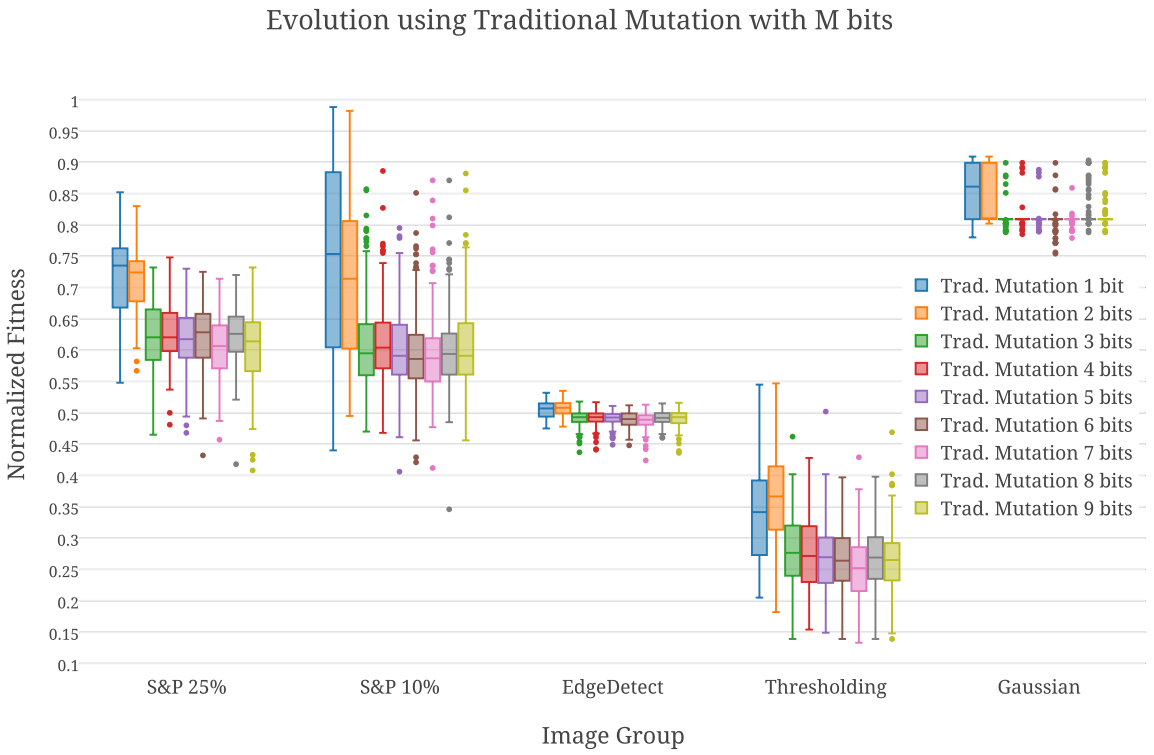


Figure 6.7: Comparison of traditional mutation with different numbers of mutation bits. One-bit mutation is the worst option because of the high probability of mutating bits of inactive cells. Seven-bit mutation appears to be the best option for an 8×8 HexArray.

bits) that affects the Active-Output increases with increasing number of bits, while at the same time mutating too many bits can degrade the genome.

For GAM, as shown in figure 6.8, increasing mutation bits caused slight improvements to the generated fitnesses, where 4-bit mutation was mostly the best choice.

Comparisons between 1-bit and 7-bit mutations and 1-bit and 4-bit GAMs are shown in figure 6.9. The generated solutions were improved in all image groups when GAM was used. The gaps between filters generated by 1-bit mutation and 1-bit GAM were significant, but the gaps were smaller when we examined the results of 7-bit mutation and 4-bit GAM.

A note of caution is due here since the traditional mutation does not care about whether

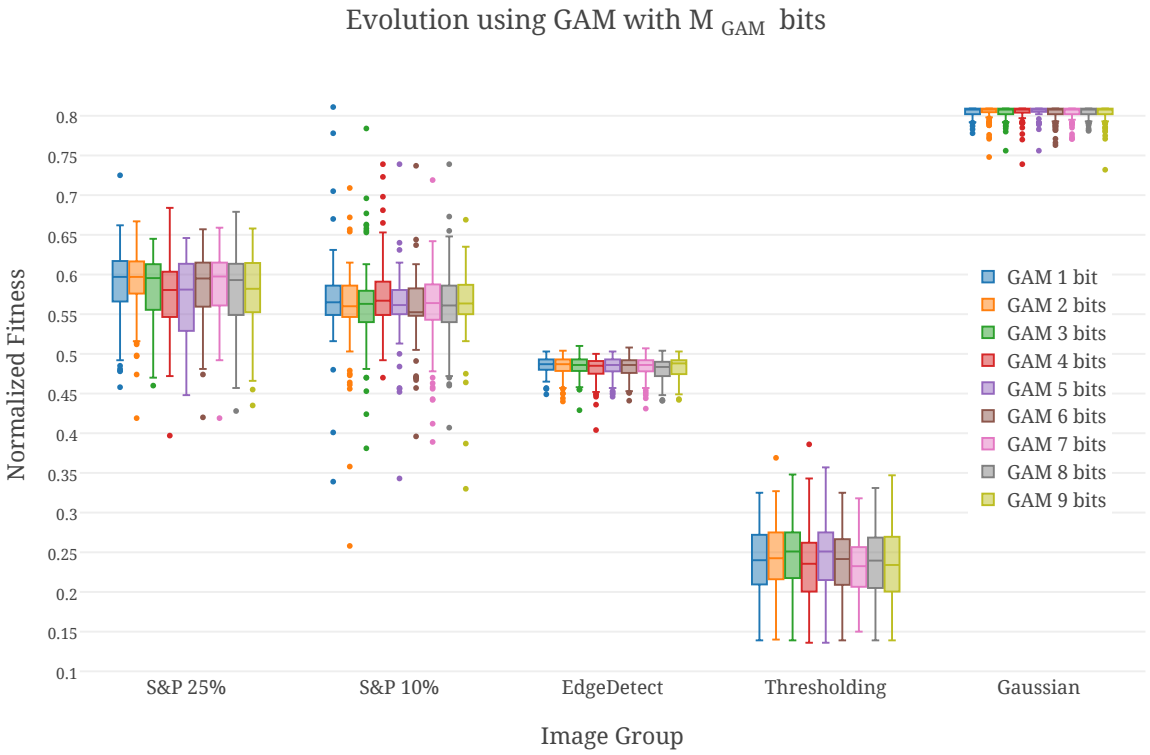


Figure 6.8: Comparison of GAM with different numbers of mutation bits. One-bit GAM is the worst case, while 4-bit is the best case for an 8×8 HexArray.

the cell is used and performs the mutation blindly, and because HexArray has multiple outputs (with different numbers of cells), there will be possible bias where short subgenomes are mutated lightly while long ones are mutated heavily. This is not the case for GAM since mutations are applied only on active cells. Overall, for an 8×8 HexArray, 4-bit GAM significantly improves the quality of the generated solutions.

6.2.4 Experiment 4: GAX Outperforms Traditional Crossover

The crossover operation guides the GA toward recombining good genomes to generate better ones. The proposed GAX can operate in three different modes, which describe the mechanism of recombining the parent genomes.

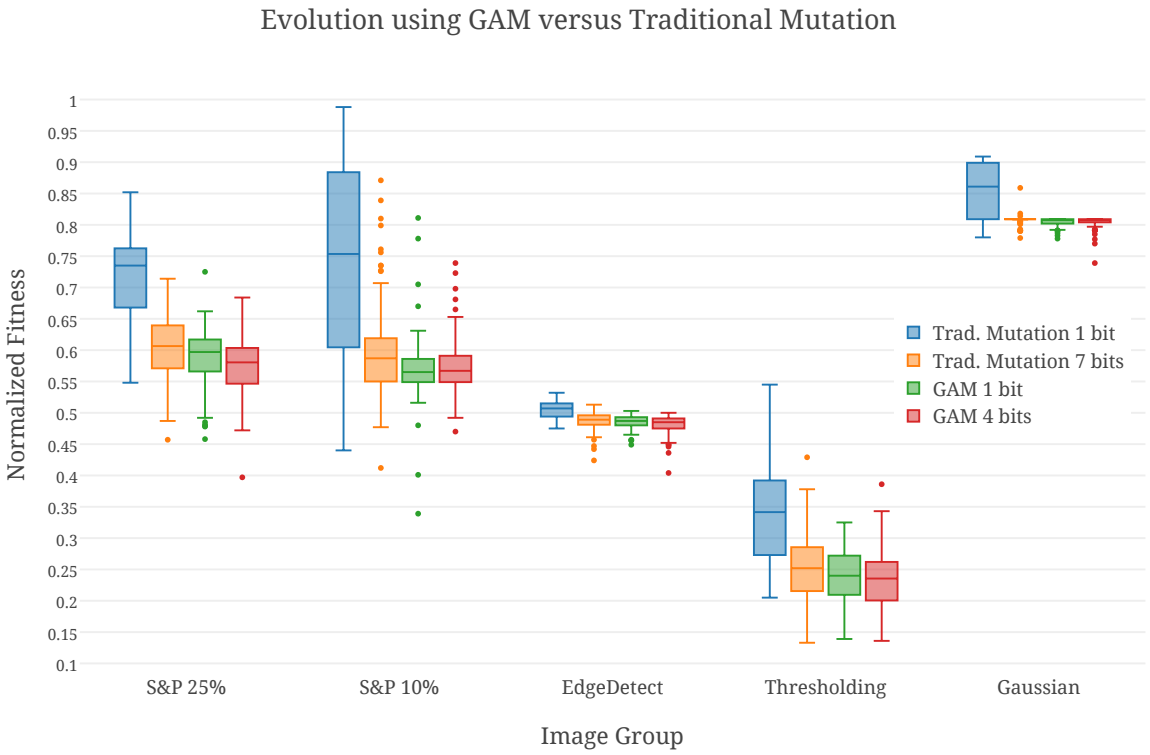


Figure 6.9: GAM showed improvement to all data sets' median and best solutions. Moreover, the distribution of solutions became more condensed and biased toward better fitness.

Hypotheses

1. GAX in cascade mode improves evolution as opposed to the traditional (two-point) crossover.

2. GAX in interleave mode improves evolution as opposed to the traditional crossover.

3. GAX in parallel mode improves evolution as opposed to the traditional crossover.

Experimental Setup

The experiment was conducted using 5 image groups. For each image group, using a 5×5 window size, we evaluated 10 generations with 1000 genomes each and selected the best fitness obtained. We repeated the process 100 times. The experiment was conducted on the following hardware architectures:

1. 8×8 HexArray with two-point chromosome-level crossover.
2. 8×8 HexArray with GAX in cascade mode.
3. 8×8 HexArray with GAX in interleave mode.
4. 8×8 HexArray with GAX in parallel mode.

Results

Using the five image groups used in the previous experiments, the data were collected by running 100 iterations of evolution using 10 generations of a population of 1000 genomes generated by a traditional crossover or GAX running in different modes. The genomes of the first generation were generated using GAC selection, and afterward, 100% of the generated genomes were using crossover (i.e., $c\_rate = 1.0$). The collected data are summarized in table 6.6 and plotted in figure 6.10.

Discussion

As shown in figure 6.10, the solutions generated using GAX were better than those using traditional two-point crossover. All modes of GAX reported improvements of the median fitnesses in all image groups ranging from 1.3% to 11.8% and better best solutions for almost all cases. Certain GAX modes outperformed others in particular image groups.

Table 6.6: Best and median fitnesses obtained from 100 runs of 8×8 HexArray with crossover and GAX running in three modes.

Image Group	S&P 25%	S&P 10%	EdgeDetect	Thresholding	Gaussian
	Median Fitness				
Trad. Crossover	0.619	0.561	0.499	0.282	0.809
GAX-Cascade	0.572	0.501	0.486	0.23	0.691
GAX-Interleave	0.552	0.517	0.48	0.221	0.785
GAX-Parallel	0.522	0.496	0.484	0.23	0.792
	Best Fitness				
Trad. Crossover	0.428	0.305	0.443	0.139	0.734
GAX-Cascade	0.366	0.278	0.442	0.139	0.615
GAX-Interleave	0.374	0.167	0.433	0.132	0.703
GAX-Parallel	0.417	0.263	0.445	0.138	0.721

GAX-Parallel appears to be the best mode for impulsive noise (S&P 25% and S&P 10%). The reason for this result may be due to the nature of "salt and pepper" noise, where evolved filters might target different noise pixels and using GAX-Parallel might allow the in-between cells to merge these parent filters to provide better offspring. For feature extraction tasks, however, GAX-Interleave appears to be the best mode. A possible explanation for this result might be that the functions used in the feature extractions were somewhat independent of others in the subgenome (e.g., `GRT` and `DIF`), and swapping (interleaving) them would be possible and could yield better filters. Finally, the GAX-Cascade performance was significantly better than that of the other modes for the Gaussian image group. This might be because the statistical noise had a relatively small magnitude (PSNR[2]=28.6 dB) and evolving a good filter was difficult and could only be performed by "cascading" filters, where each made a small improvement. In general, GAX shows promising results

[2]Peak signal-to-noise ratio is the maximum value for the ratio between a signal and its noise. Since the range is wide, PSNR is often expressed in logarithmic decibel scale.

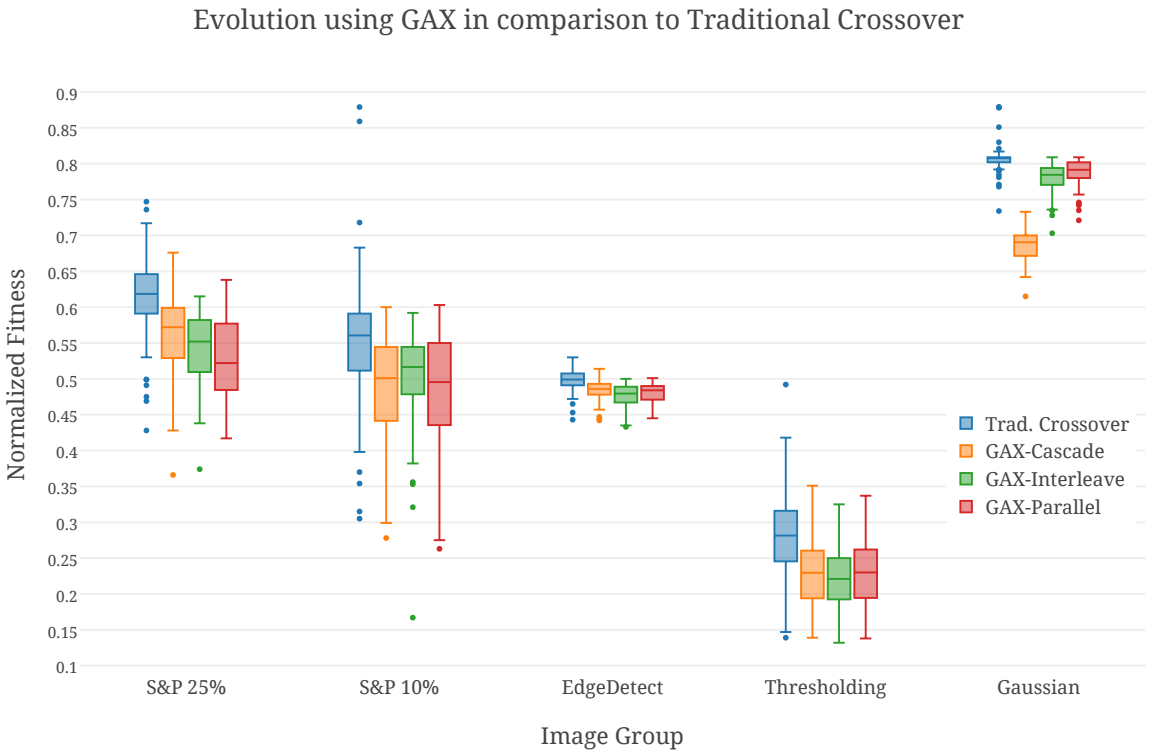


Figure 6.10: Best fitness obtained in 100 iterations using 2-point traditional crossover, GAX-Cascade, GAX-Interleave, and GAX-Parallel.

in comparison with traditional crossover. Each of the GAX modes appear to be essential for targeting specific problems. This result indicates that an adaptive selection method for genetic operations that is based on their performance should be used, similar to that proposed in [156].

6.2.5 Experiment 5: The Effect of Population Size on Evolution

Two important parameters for evolution are the generation and population sizes. Increasing the number of generations allows for improving the populations incrementally from a generation to the next through genetic operations. Increasing the population size (number

of genomes per generation) allows generation of a larger pool of parent genomes for the next generation, which can result in diverse genetic operations.

Hypotheses

1. Increasing the number of generations improves evolution (implies decreasing the population size).

2. Increasing the population per generation improves the evolution (implies decreasing number of generations).

Experimental Setup

The experiment was conducted using 5 image groups, where genomes were generated using GAC selection (20%), 4-bit GAM (20%), GAX-Cascade (20%), GAX-Interleave (20%), and GAX-Parallel (20%). For each image group, using a 5×5 window size, we evaluated 10,000 genomes using the genome/population combinations described in table 6.7. We tested each combination in 50 independent runs and selected the best fitness obtained.

Table 6.7: Different combinations of number of generations and genome size with a fixed total number of genomes.

Combination name	Number of generations	Population size
G200_P50	200	50
G100_P100	100	100
G40_P250	40	250
G20_P500	20	500
G14_P750	14	750
G10_P1000	10	1000
G5_P2000	5	2000

Results

For each of the five image groups, 50 data points were collected for every combination outlined previously, as summarized in table 6.8 and plotted in figure 6.11.

Table 6.8: Best and median fitnesses obtained from 50 runs of 8×8 HexArray with different generation/population combinations.

	S&P 25%	**S&P 10%**	**EdgeDetect**	**Thresholding**	**Gaussian**
	Median Normalized Fitness				
G200_P50	0.53	0.523	0.482	0.209	0.761
G100_P100	0.556	0.517	0.487	0.231	0.767
G40_P250	0.552	0.523	0.486	0.226	0.769
G20_P500	0.547	0.54	0.486	0.231	0.782
G14_P750	0.584	0.546	0.488	0.242	0.786
G10_P1000	0.58	0.552	0.482	0.225	0.791
G5_P2000	0.578	0.561	0.482	0.223	0.802
	Best (min) Normalized Fitness				
G200_P50	0.385	0.348	0.439	0.138	0.702
G100_P100	0.434	0.347	0.441	0.138	0.665
G40_P250	0.426	0.389	0.454	0.139	0.678
G20_P500	0.445	0.33	0.437	0.147	0.712
G14_P750	0.41	0.294	0.427	0.139	0.692
G10_P1000	0.405	0.246	0.456	0.139	0.722
G5_P2000	0.371	0.461	0.435	0.032	0.73

Discussion

Among all the image groups, $G200\_P50$ reported better solutions with improvement in the medians reaching 5%. A trend was clear on the S&P 25%, S&P10% and Gaussian image groups, where the more generations (and smaller populations) there were, the better

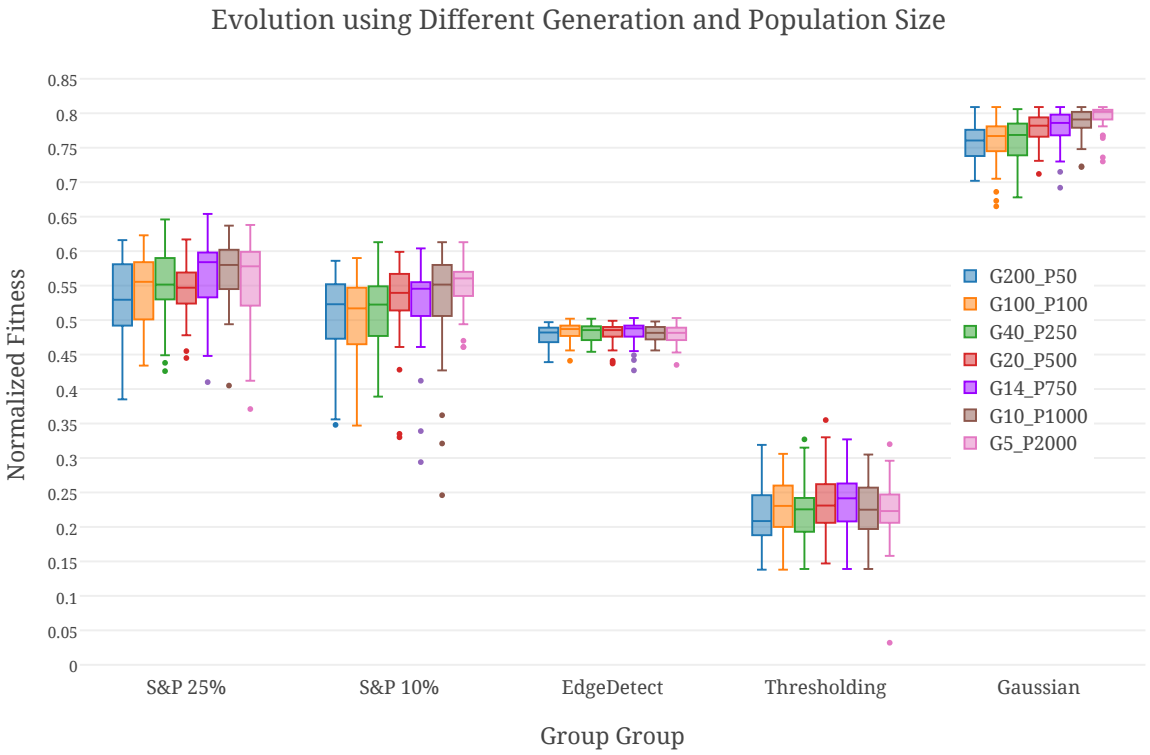


Figure 6.11: Fitness distribution for different numbers of generations and population size. The best combination for improving generated solutions overall was using the smallest population size with the largest number of generations. However, the best combination for finding high-quality solutions occasionally was using the largest population size with the smallest number of generations.

the fitness was. The explanation of this finding was that evolution used genetic operations iteratively on small sets of genomes, resulting in a "depth-search" in a subdomain of the search space. An unexpected observation was that the best solutions were not obtained using smaller populations such as medians, but rather using larger populations. We believe the reason for this result is that the genetic operations were running using a large group of selected (diverse) parents, which resulted in a "breadth-search". Generally, small populations (many generations) cause a depth-search in a small subset of the solution space, where the improvement is slow but steady. Conversely, larger populations (few

generations) resulted in a breadth-search, where good outliers (solution) were occasionally found. Further research should be undertaken to investigate the gain achieved by allowing dynamic assignment of the generation/population throughout the evolution process. The dynamic assignment should use large populations with few generations in the early stages of evolution and smaller populations with many generations toward the end of evolution.

6.2.6 Experiment 6: Adaptive Filter Evaluations

The online adaptation feature of EHW is a significant advantage over using traditional methods. An evolved filter for an image with a low SNR (signal-to-noise ratio) would be different than that evolved for a high SNR. The target of this experiment is not comparing the evolved filters to commercial filters, but rather studying their behavior for different levels of noise to determine how efficient they are.

Hypotheses

1. Evolved filters consistently adapt to noise levels.
2. Although filters are made specifically for one noise level, they work on other levels.
3. Although filters are made specifically using one image, they work on other images.

Experimental Setup

The experiment was conducted using 18 image groups, where 9 of them were for the same image group (Lena) with different impulsive noise levels $\in \{2.5\%, 5\%, 7.5\%, 10\%, 15\%, 20\%, 30\%, 40\%, 50\%\}$, while the other were for another image group (cameraman).

For each image group, using a 5×5 window size, we evaluated 1000 generations with 50 genomes each and selected the best filter evolved. This process was repeated 51 times,

resulting in 51 filters. We selected the median filter and the best filter of the current image group and ran them on all other image groups (the other 17 image groups).

The experiment was conducted using an 8×8 HexArray. The genomes in every generation were generated using GAC selection (20%), 4-bit GAM (20%), GAX-Cascade (20%), GAX-Interleave (20%), and GAX-Parallel (20%)

Results

For the first image (Lena), the *average* filter evolved for each noise level was tested on all other noise levels of the same image (right side of figure 6.12) and all noise levels of other images (left side of figure 6.12). Additionally, the result of using the median filter with a 5×5 window size was added to the plot. Similar to the previous results, the data for the *best* filters obtained for the first image tested on the other is shown in figure 6.13. Accordingly, figure 6.14 was plotted using data obtained when running *average* evolved filters of the second image on all other images, while figure 6.15 was using the *best* evolved filters.

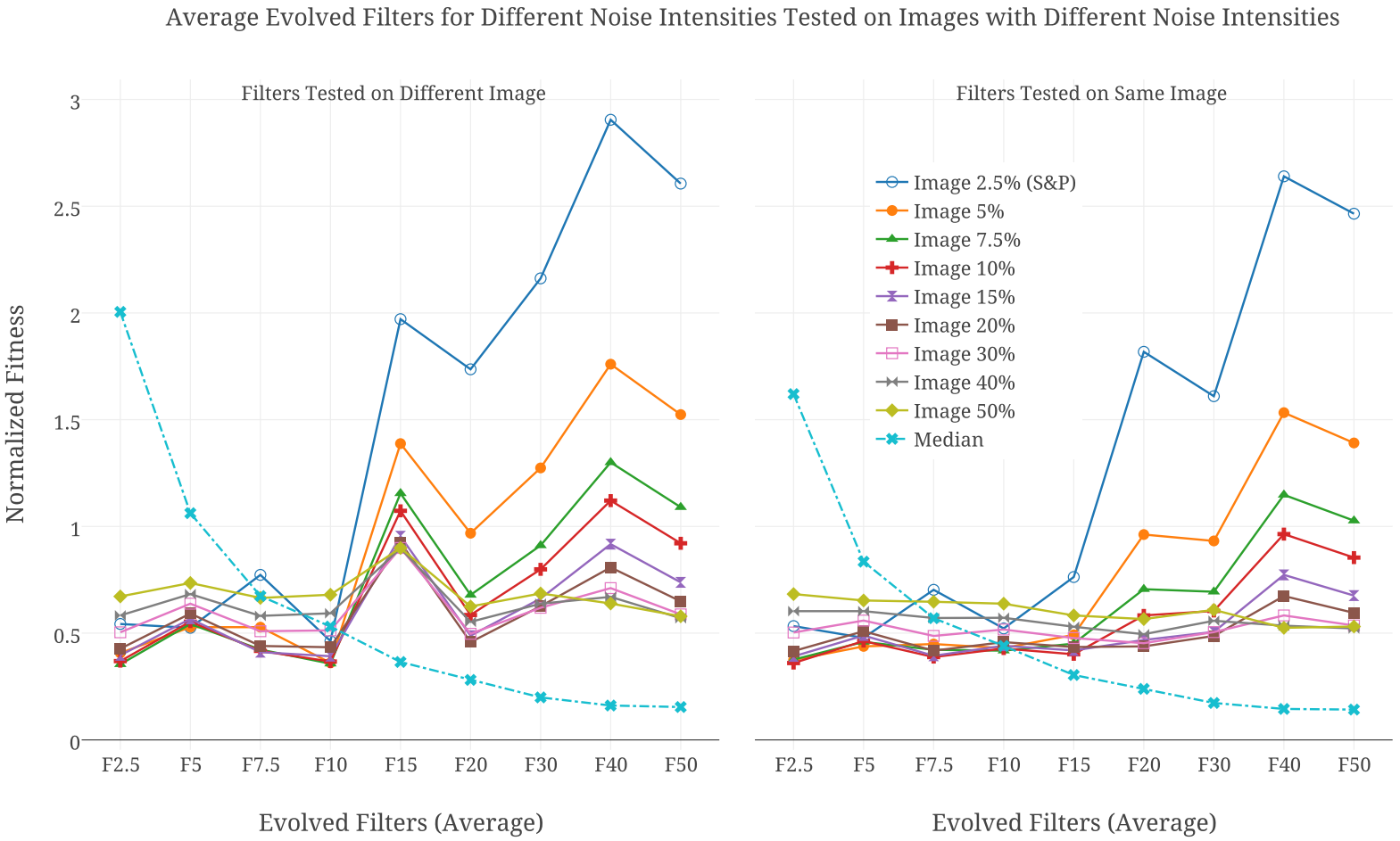


Figure 6.12: Average of filters evolved for every noise level of an image (Lena) were tested on other noise levels of the same image (right) and a different image (left).

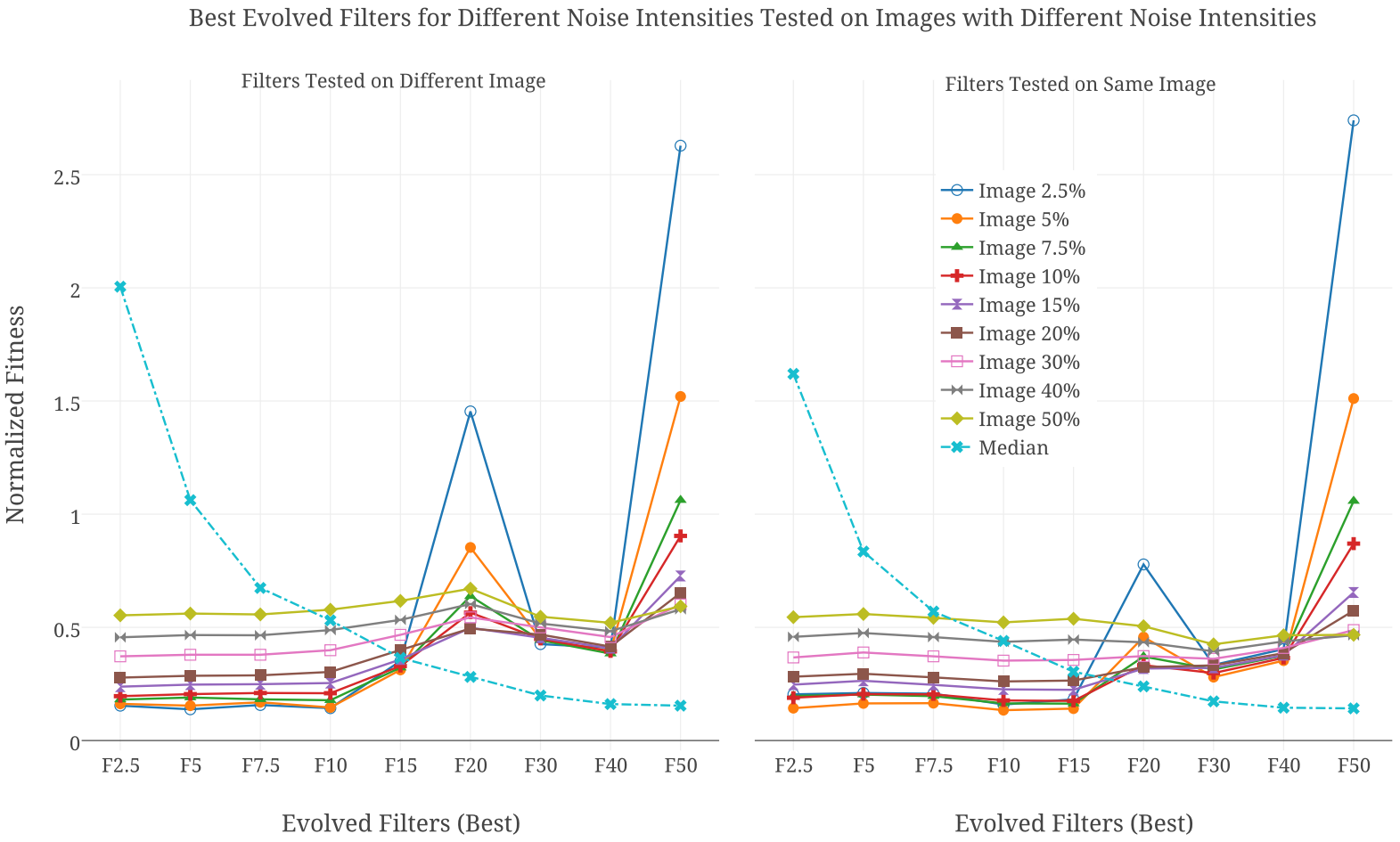


Figure 6.13: Best of filters evolved for every noise level of an image (Lena) were tested on other noise levels of the same image (right) and a different image (left). Some evolved filters showed consistent behavior on a wide spectrum of noise, unlike the median filter. Filters developed for images with high SNR performed poorly on images with a low SNR.

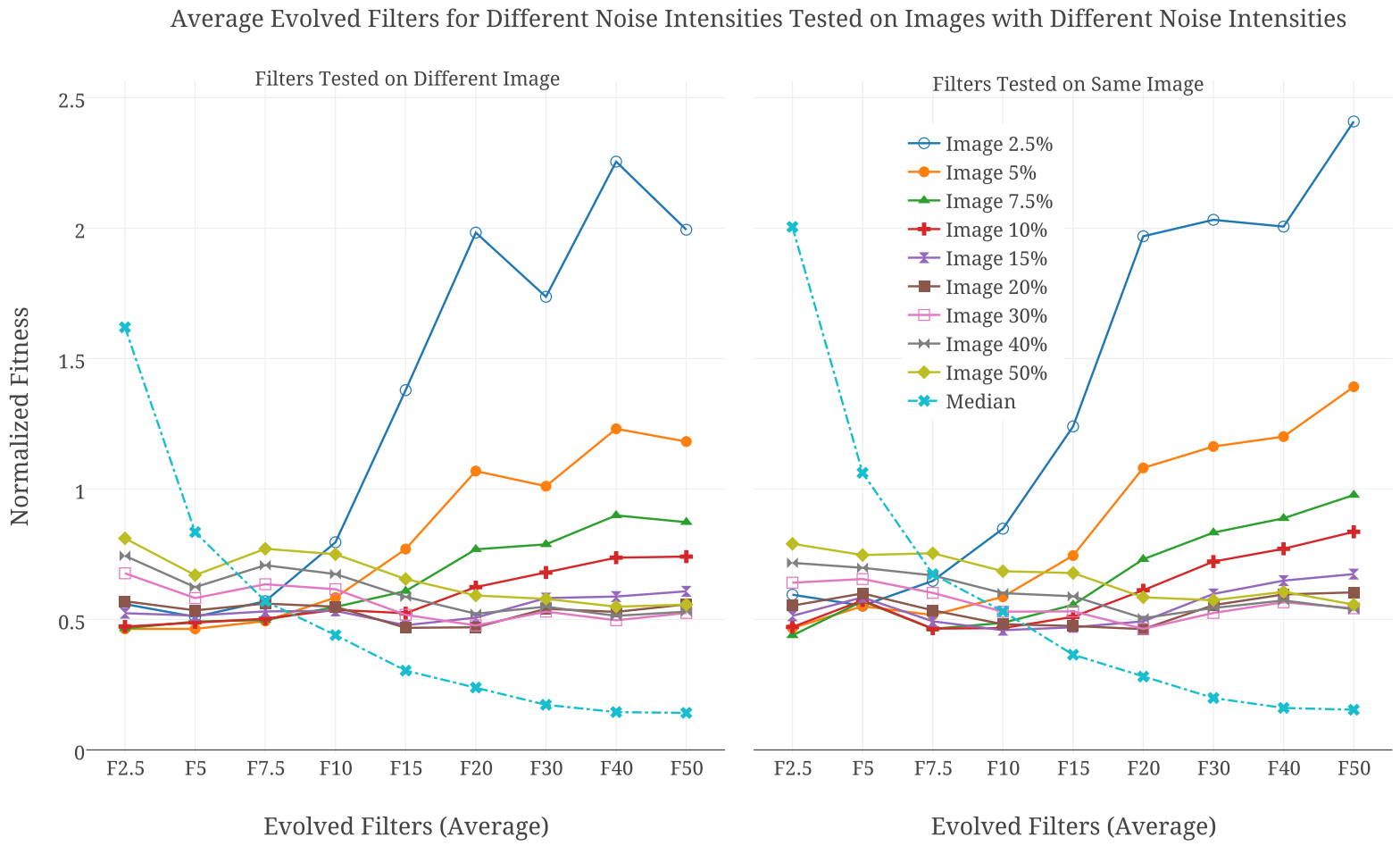


Figure 6.14: Average of filters evolved for every noise level of an image (cameraman) were tested on other noise levels of the same image (right) and a different image (left).

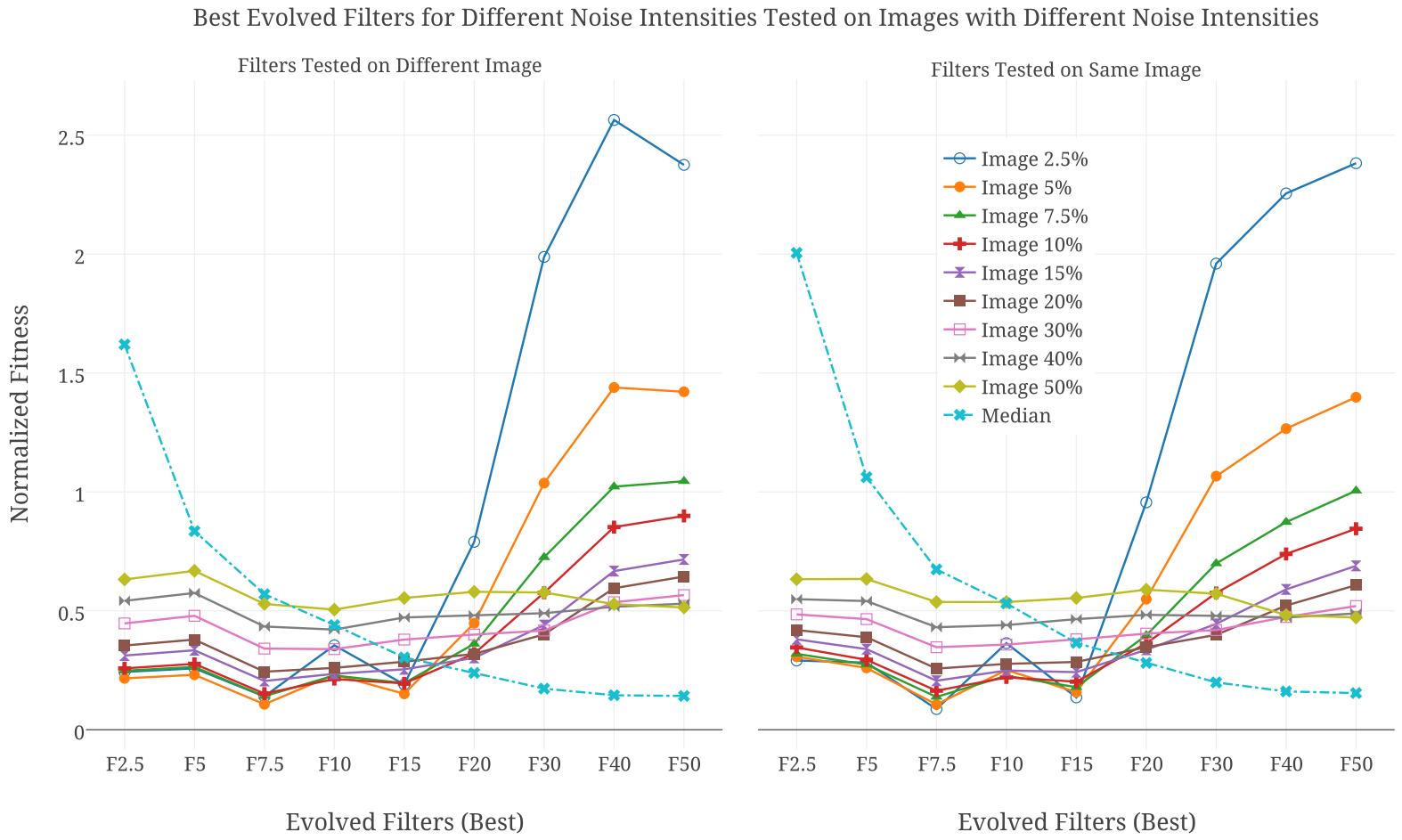


Figure 6.15: Best of filters evolved for every noise level of an image (cameraman) were tested on other noise levels of the same image (right) and a different image (left). The median filter did not perform well for an image with a high SNR.

Discussion

The performance of the median filter was dependent on the noise level. It performed poorly on images with a high SNR ($\leqslant$ 5% of S&P noise); in fact, images were degraded as the fitness was $>$ 1.0. However, for images with a low SNR ($\geqslant$ 40% of S&P noise), the filter performed well. The performance of the median filter did not change using different images.

Filters evolved on a specific image with a specific SNR consistently scored good fitness regardless of the noise level, unlike the median filter. Additionally, the evolved filters performed similarly on different image groups (but with the same noise level), similar to the median filter.

The average evolved filters[3] outperformed the median filter in all images with noise levels of 10% or less. The best evolved filters outperformed the median filter in all images with noise levels of approximately 18% or less.

When examining the results of running all filters on images with different noise levels, two trends were observed. First, the filters evolved for low noise levels performed best at their levels but poorly on images with high noise levels. In some cases (e.g., F2.5 on 50% S&P), degradation for the resulting image was observed. Second, filters evolved for high noise levels performed best at their levels, but moderately on images with low noise levels. These differences can be explained in part by considering the two corner cases, i.e., F2.5 on 50% S&P and F50 on 2.5% S&P. A 2.5% noise indicates that approximately 24 pixels of the 25 pixels of the data window are not corrupted, allowing the evolved filter to use them. Consequently, running this filter on an image with 50% S&P where almost half of the pixels are corrupted will produce undesired fitness.

[3]The average evolved filter is the 26^{th} genome after ranking the best genomes of the 51 runs.

For 50% noise, however, approximately half of the pixels are corrupted, resulting in filters that avoid using half of the data. Therefore, when running this filter on an image with 2.5% noise, it performs relatively well. Filters developed for 15% to 30% S&P noise appear to perform well on the two sides of the noise spectrum.

In summary, evolved filters consistently adapt to the targeted noise level. These filters are not image specific and perform comparably on different images. For images with a high SNR, HexArray evolved high-quality filters, whereas the median filter degraded them. For images with a low SNR, HexArray evolved decent quality filters, whereas the median filter reduced noise significantly. Filters developed for 15% to 30% S&P might be good candidates as noise-level-independent filters. Filters developed on images with a low SNR can be used on images with a high SNR, whereas the opposite may corrupt images.

6.2.7 Experiment 7: Autonomous Evolution for Variety of Filters

The power of EHW is its ability to autonomously find solutions for unknown problems (without manually identifying the problem). In other words, EHW needs no more than a fitness function to start developing a filter for a training image irrespective of the problem type. Since the system works locally on a sliding window of 5×5 pixels, it will not be able to develop some of the advanced image filters that require a large window size. This should be acceptable since the image processing application was selected as a case study to explore the online adaptability of the HexArray system. Another target of this experiment is to quantify the value of having GAC selection, GAM and GAX.

Hypotheses

1. HexArray can autonomously develop a variety of filters.

2. All genetic operations participate in generating the evolved filters.

Experimental Setup

The experiment was conducted using 16 image groups (as described in table 6.9). For each image group, using a 5×5 window size and the pixel location (X, Y), we evaluated 1000 generations with 50 genomes each and selected the best fitness obtained. We repeated the process 11 times (an odd number to ease selecting the median value). The genomes in every generation were generated using GAC selection (20%), 4-bit GAM (20%), GAX-Cascade (20%), GAX-Interleave (20%), and GAX-Parallel (20%).

Results

We collected 11 data points for every image group (plotted in figure 6.16); their best and median values are shown in table 6.9. The data points were classified based on the genetic operation that created them and are plotted in figure 6.16.

Table 6.9: Variety of image groups to explore the autonomous adaptivity of the system.

	Normalized Fitness		**Generated by (in %)**		
Image Group Name	**Median**	**Best**	**GAC**	**GAM**	**GAX**
Blurring	0.65	0.637	0	9	91
Deblurring	0.955	0.949	27	18	55
Edge detection (Roberts)	0.349	0.335	9	18	73
Edge detection (Canny)	0.253	0.247	9	27	64
Edge detection (Sobel)	0.118	0.11	36	9	55
Gradient adjustment	0.632	0.587	9	18	73
Periodic dark rows	0.135	0.135	9	18	73
Histogram equalization	0.416	0.405	0	0	100
Morphological (erosion)	0.351	0.329	0	18	82
White balancing	0.174	0.165	9	18	73
Blob detection (Laplacian)	0.287	0.258	18	9	73
Contrast adjustment	0.371	0.297	18	9	73
Darkness equalization	0.58	0.458	18	18	64
Brightness equalization	0.148	0.133	27	18	55
De-pixelate	0.845	0.833	0	36	64
Periodic dark columns	0.865	0.835	0	36	64
		All image groups	12	18	70

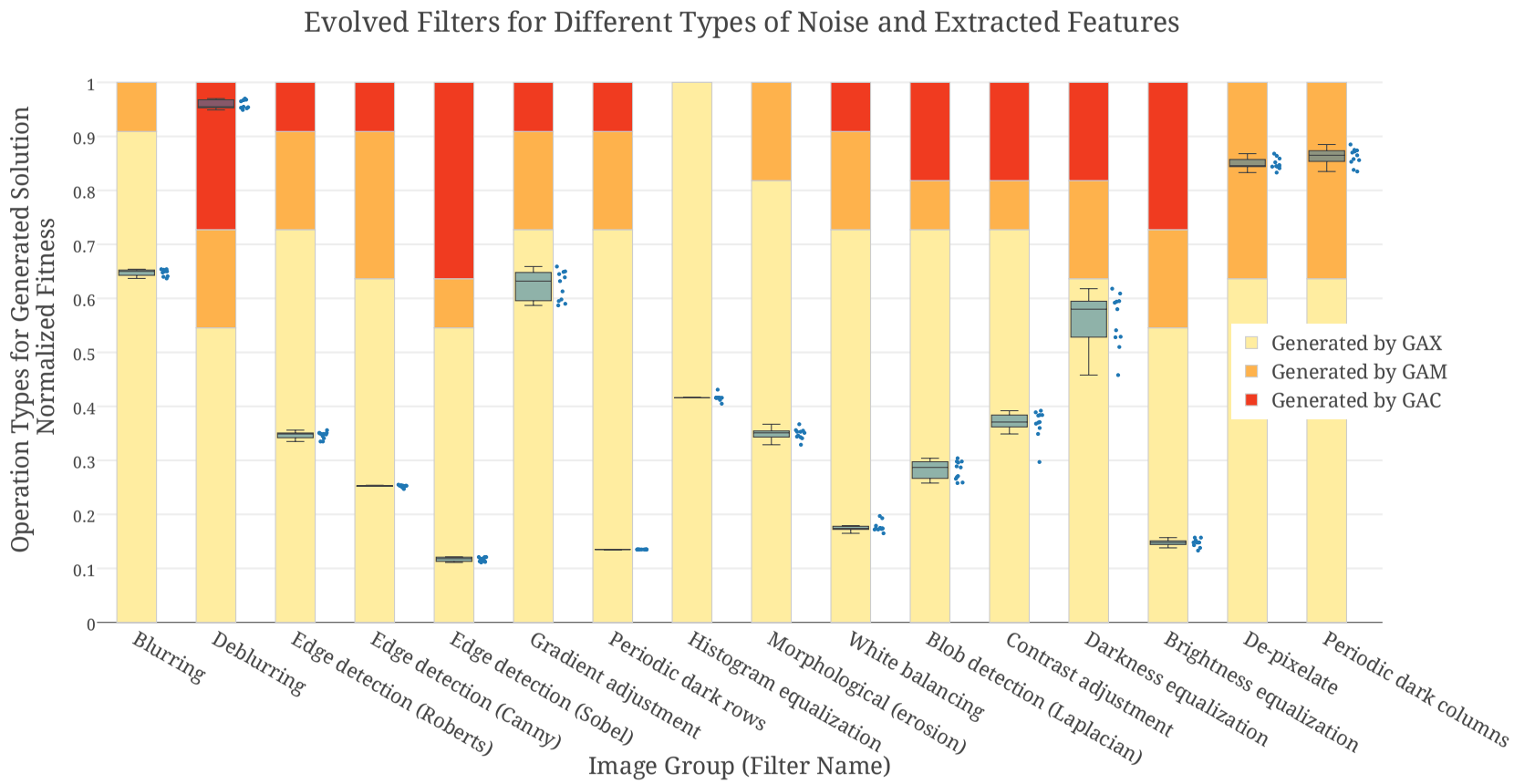


Figure 6.16: HexArray could autonomously evolve many filters. All genetic operators contributed in evolution. Some filters were solely generated using GAX (or GAX and GAM).

Discussion

We will divide and discuss the developed filters in five independent groups. The first group includes deblurring, blurring, and de-pixelate filters, which are relatively hard problems, as shown in order in figure 6.17, 6.18, and 6.19, respectively.

(a) Training image **(b)** Evolved image **(c)** Reference image

Figure 6.17: Deblurring was difficult because the blurred image was constructed using a 6×6 window.

Deblurring was the hardest problem, and the system could not make more than 5% improvement to the fitness. One reason for this poor performance was that the blurry image was created using a Gaussian filter with radius=6 [4], which is larger than the window size that HexArray was using. The second reason is that blurring (convolution) is a degrading process, and reconstructing pixels using simple functions is difficult. The fact that 27% of the generated filters were generated using GAC selection indicated that the system had difficulties in generating better offspring of selected parents.

However, the system could develop better filters for the *blurring* problem. This result occurred because blurring is inherently "easier" than deblurring. Most of the generated solutions were produced using GAX, which was expected as GAX in cascade mode would

[4]The Gaussian filter radius is the standard deviation sigma.

(a) Training image **(b)** Evolved image **(c)** Reference image

Figure 6.18: The system performed moderately in developing a blurring filter with a 35% fitness improvement; the filters were mostly generated using GAX.

be ideal for this problem. Finally, although the *de-pixelate* is a hard reconstruction mechanism of a degraded image (16 pixels were compressed to one pixel), the system found filters with fitness improved by 17%. The number of filters generated by GAM was higher than expected, 36% while expecting 20%, indicating that mutation is an effective operation for this problem. Generally, the system evolved filters with improved fitness (5%-36%), even for hard problems and using a small data window.

(a) Training image **(b)** Evolved image **(c)** Reference image

Figure 6.19: The system achieved a 17% fitness improvement for the de-pixelate filter.

The second group includes three edge detection filters (as shown in figure 6.20-6.22), a blob detection filter (as shown in figure 6.23), and a morphological filter (as shown in figure 6.24).

(a) Training image **(b)** Evolved image **(c)** Reference image

Figure 6.20: The system evolved an edge detection filter (Roberts cross).

(a) Training image **(b)** Evolved image **(c)** Reference image

Figure 6.21: The system generated an edge detection filter (Canny operator).

The system performed well and evolved some fine filters, such as that for Roberts edge detection (as shown in figure 6.20) and Sobel edge detection (as shown in figure 6.22). This result was anticipated since the functions needed for developing these filters are generally

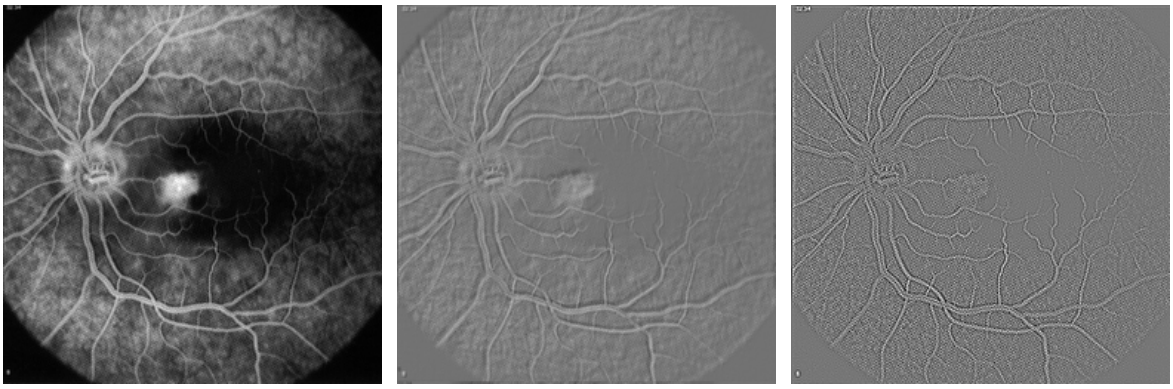
similar to those that we are using in the HexArray system. A filter with a fitness value of 0.25 is considered to be a good filter since the noise is reduced by 75%.

(a) Training image **(b)** Evolved image **(c)** Reference image

Figure 6.22: The system developed an edge detection filter (Sobel operator).

The evolved Canny filter had a similar fitness, but the evolved image was not appealing (the second image group from the top in figure 6.21). The explanation for this result was that the system's only gauge to determine the quality of a solution was the fitness function, which was the normalized MAE function, and for this specific image group, the evolved filter scored good fitness. In fact, the fitness value for an "all black" image was 0.26, and the system would choose it if it could not find better solutions. This result indicated that for real-world applications, sophisticated or multi-objective fitness functions [38] are needed.

All genetic operations contributed to finding solutions for this group. However, none of the developed morphological filters used GAC selection, and most of them were generated using GAX. A possible explanation for this might be that a complex filter, such as the morphological filter, would typically be evolved by means of the genetic operations rather than random selection. According to these data, we can infer that the system is able to autonomously evolve satisfactory filters in this group.



(a) Training image **(b)** Evolved image **(c)** Reference image

Figure 6.23: The system found a good filter for the blob detection problem.

(a) Training image **(b)** Evolved image **(c)** Reference image

Figure 6.24: A gray-scale morphological filter was developed with good fitness.

The third group of filters consists of those for adjusting the tonal distribution, such as brightness equalization (as shown in figure 6.25) and darkness equalization filters (as shown in figure 6.26). The system evolved relatively good filters for the bright training image, but did not perform as well for the dark training image. The difference in performance can be explained by highlighting the difference in the tonal distribution (which might not be noticed by the naked eye), where the dark image had a narrower spectrum of pixel values ranging between 41 and 114 in comparison with 41 and 255 for the bright image. Filters

(a) Training image **(b)** Evolved image **(c)** Reference image

Figure 6.25: The system evolved a good filter for image brightness adjustment.

were generated using all genetic operations. GAC selection was quite effective for the brightness equalization filters.

(a) Training image **(b)** Evolved image **(c)** Reference image

Figure 6.26: The generated filter was decent because the training image had a narrow tonal distribution.

The fourth group of filters is for variable noise intensities based on pixel location, such as periodic dark rows, periodic dark columns, and gradient adjustment filters. The developed filters for the periodic dark rows were different versions of the same function, as shown in figure 6.27. Surprisingly, this function did not include the pixel X-location as

we predicted; rather, it used a simple "maximum" function of a 3×1 data window, which allowed filtering these dark lines. This observation may support the hypothesis that the EHW can find solutions that might be overlooked by humans.

(a) Training image **(b)** Evolved image **(c)** Reference image

Figure 6.27: Removable of periodic dark rows noise with static shade on a 4-pixel period – the noise is X-coordinate dependent.

In contrast to the previous image, HexArray did not find good filters for the "Periodic dark columns", as shown in figure 6.28. Two of the evolved filters included the pixel Y-location, but they did not score the best fitness. The difference between this problem and the previous one may explain the difference in performance. The earlier problem was easy since it included a horizontal pattern of two normal rows and two shaded rows. Conversely, the latter problem was comparatively hard because it included a vertical nonlinear noise pattern generated by a Fourier transform with a period of eight pixels. Thus, given that the system worked on a 5×5 window, it would not be able to develop good filters. Another observation that indicated that this was a hard problem, was that none of the solutions were generated by GAC selection.

The system evolved gradient adjustment filters with moderate fitness, as shown in figure 6.29. All evolved functions included the pixel X-location, as expected. The evolved image

(a) Training image **(b)** Evolved image **(c)** Reference image

Figure 6.28: Periodic dark columns noise with a nonlinear Fourier transform on an 8-pixel period – the noise is Y-coordinate dependent.

with the best fitness was not appealing, and we thought that the other evolved images with a slight fitness difference appeared better. However, this all goes back to the fitness function in use.

(a) Training image **(b)** Evolved image **(c)** Reference image

Figure 6.29: Gradient noise is a spatially variant degradation where pixels with a small X-location were brightened and pixels with a high X-location were darkened.

Finally, the last group in this discussion includes histogram equalization, contrast adjustment, and white balancing problems, as shown in figure 6.30. The system performed well for all the problems using all genetic operations, except for the histogram equaliza-

(a) Training image **(b)** Evolved image **(c)** Reference image

(d) Training image **(e)** Evolved image **(f)** Reference image

(g) Training image **(h)** Evolved image **(i)** Reference image

Figure 6.30: Evolving filters for brightness equalization problems. (Top) Histogram equalization. (Middle) Contrast adjustment. (Bottom) White balancing.

tion problem, where GAX was the only operator that yielded good solutions. This result might indicate that good filters could be generated for this type of problem by cascading, interleaving, or parallel-recombining parent genomes.

Overall, the system performed quite well on all problems except for a subset of relatively hard problems. The problem difficulty was due to many reasons, including the following:

1. The noise degraded the training image, and reversing it was impossible.

2. A large data window size was required.

3. Nonlinear functions were needed.

Per the diversity of generated filters, we may conclude that the system features an acceptable level of autonomous adaptation. The EHW system used the genetic operators throughout the evolution process to improve populations. Altogether, the system utilized all genetic operations efficiently, where 70% of the generated solutions were using GAX, 18% using GAM, and 12% using GAC selection. This may indicate that we could increase the crossover rate and decrease the randomization rate. Another observation was that the system was utilizing GAM and GAX (and not GAC selection) for many of the hard problems, which suggests that genome-aware mutating, cascading, interleaving, and parallel-recombining genomes are effective for hard problems. Another finding was that for real-world applications, the MAE fitness function might not be an adequate function and more sophisticated or multi-objective fitness functions might be needed.

6.3 Implementation Analysis

The proposed HexArray platform was implemented on a ZedBoard kit, a development board manufactured by Avnet that contains Xilinx Zynq-7000 All-Programmable SoC XC7Z020 [176]. The SoC consists of a processing system (PS) and a programmable logic (PL). The PS is a dual ARM Cortex-A9 MPCore running at 667 MHz with a dedicated 512 MB of DDR3. The PL is an Artix-7 FPGA with 53,200 LUTs, 106,400 Flip-Flops, 4.9 MB of block RAM, and 220 DSP slices. The FPGA typically operates at a 100 MHz clock speed, but can be increased to 250 MHz.

For the chosen image processing application, an 8×8 HexArray was adequate for evolving many filters. The hardware modules of the designed 8×8 HexArray hardware core were written in HDL, synthesized, placed, and routed using the Vivado 2015.2 IDE tool. The implemented modules are as follows:

1. A data window controller
2. A genome register
3. 31 AICs
4. 31 AOCs
5. 64 HexCells
6. Supporting modules, including AXI interconnects and general-purpose inputs/outputs (GPIOs).

The data window controller incorporates a DMA operating in a scatter/gather mode with a data width of 4 bytes. The DMA feeds into an AXI4-Stream data FIFO with a depth of 2048, which is connected to a converter with a 4-to-8-byte data width. The resulting 8

bytes form one slice of the 8×5 sliding window of the array input data. At any time, the module provides 5×8=40 bytes of data (pixels in our case). These pixels are available for the AIC modules to select from. However, the data are not valid until the ready signal is asserted. This allow the user to control (using AXI) the process using the start and speed signals. Providing a sliding window for the boundary pixels of an image involves some complexity. However, by preprocessing the data by the PS before sending it through the DMA and by utilizing the ready signal, complexity was avoided.

A genome register consists of an `I_GENOME` and an `A_GENOME`. The `I_GENOME` includes the configuration for 31 AICs. Each AIC needs to select a pixel out of 5×8 pixels, which means that 6 bits are needed. Therefore, the `I_GENOME` size is 186 bits (6 32-bit words were used). Conversely, the `A_GENOME` is needed to hold the chromosome data of the cells. In fact, the Sel_f is not needed as the encoded information is utilized by the PS to initiate DPR transactions through PCAP. Therefore, the `A_GENOME` size is 384 bits (12 32-bit words were used). These 18 32-bit are programmed by AXI transactions.

AIC is simply a 40×1 multiplexer with an 8-bit bandwidth. The selection signal of the multiplexer is encoded by 6 bits in the `I_GENOME`.

Every array output feeds into an AOC. This module is responsible for calculating the fitness value, which is the MAE. The MAE is the accumulation of the absolute difference between the generated output pixel and the reference one for all pixels in the image. This requires the AOC to be aware of the first and last pixels of the data, which is achieved by including a pixel counter and "expected count" register assigned by the user using AXI transactions. The expected count is the total number of pixels in the used image, which is 256×256 in our case. When the pixel counter reaches the expected count, meaning the last pixel in an image, the AOC pushes the accumulator value (i.e., MAE value) to an FIFO that is accessible by the PS.

The main component of the reconfigurable hardware core is the systolic array; here, it is an 8×8 HexArray. The array includes static and dynamic partitions. The static circuitry for one HexCell consists of three 4×1 8-bit-wide multiplexers for the output ports and three input buffers with different sizes based on equations 5.1, 5.2, and 5.3. The dynamic partitions involved two challenges. The first one was that ideally 64 PEs need 64 reconfigurable regions, but because the smallest reconfigurable region, which is 200 CLBs for the selected device, is three times larger than the largest functional unit, we had to combine every three functional units to be programmed into one region, resulting in 22 regions. The other challenge was that the IDE tool, Vivado, did not support bitstream relocation where a single partial bitstream can be used to program multiple reconfigurable regions. Solving the shortening of this tool was not the focus of this research, especially because this feature was supported by the previous IDE tool (ISE) and there are some techniques to enable it, such as that proposed by Oomen et al. [177]. To overcome this challenge and still be able to practice DPR, we merged all functions into a single functional unit that become controlled by the `A_GENOME`[5] to select the intended function. Consequently, the required partial bitstreams were 22 files. All files were stored in the PS memory, where each file was 55.3 KB in size. To program the bitstream, Zynq supports two methods: ICAP and PCAP [178]. PCAP was chosen for two reasons. It does not need any hardware instantiations in the PL, and it is relatively fast.

The GAGA program was written in the C/C++ programming language, which ran on the PS. The executable binary was built using Xilinx SDK 2015.2 with a size less than 2 MB for the bare metal application[6] and 1.2 MB for the partial bitstreams. A high-level monitoring

[5]Note that this requires the `A_GENOME` to include the Sel_f, resulting in expanding the `A_GENOME` size by an additional 8 32-bit words, as $\mathbb{P} = 4$.

[6]Bare metal application is an application that runs directly in hardware and does not need an operating system.

dashboard, written in Python, (shown in figure 6.31) was created to communicate with the PS using USB. Using this dashboard, users can select the image set, adjust the evolution parameters, control the evolution process, and obtain the evolution results in real time. Since the reconfigurable hardware core (in the PL) and GAGA (in the PS) are independent, GAGA can perform genetic operations in parallel with the array evaluating a genome.

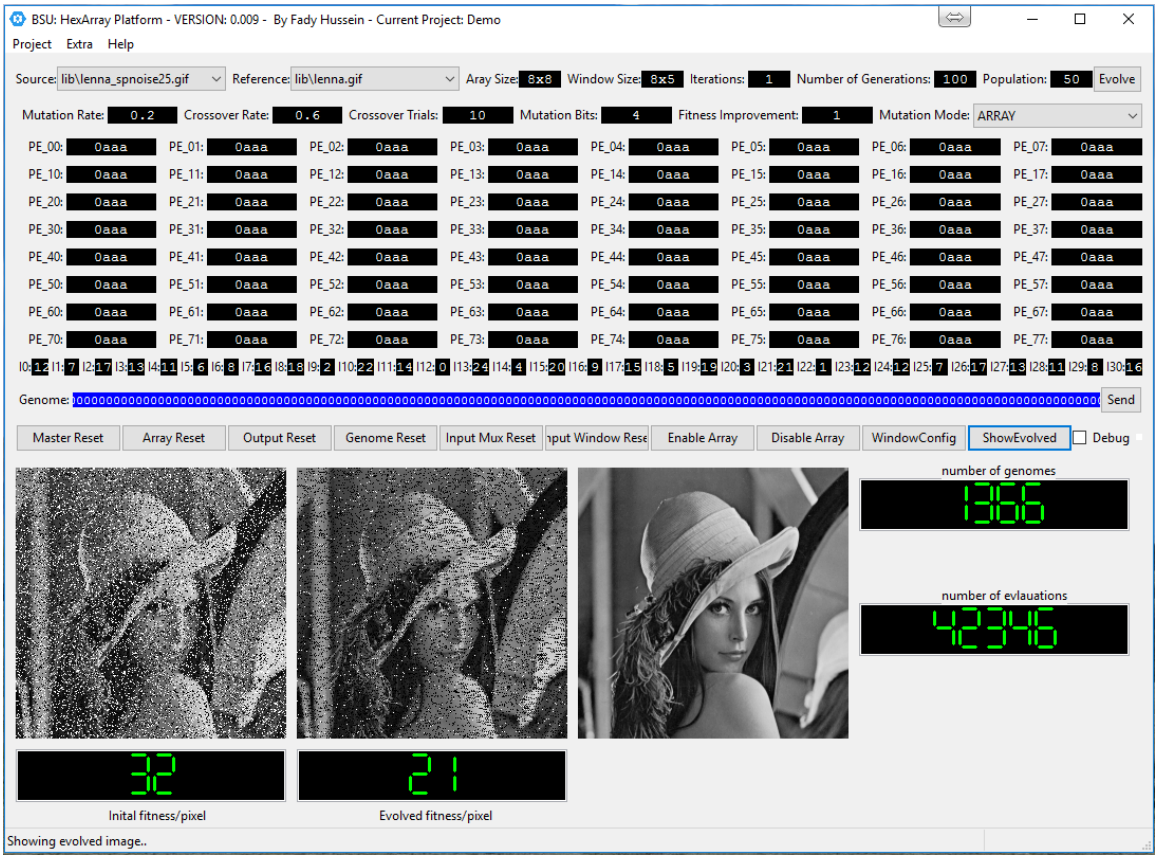


Figure 6.31: High-level dashboard for monitoring evolution is created. It allows the user to customize inputs and visualize the results.

In the following subsections, resource utilization and time analysis will be discussed.

6.3.1 Resource Utilization

The resource utilization for the full system was 76.8% of the LUTs and 78.4% on the flip-flops, as shown in table 6.10. In general, reducing the resource utilization was not targeted during system implementation as there was no need for it in the selected case study. However, optimizing the resource utilization is achievable. In addition to simply optimizing the modules themselves, one major technique for optimizing resource utilization is to exploit the 48-bit DSP blocks in the FPGA. For example, for the AOCs, which are the top resource utilizers, we could use 31 DSP slices rather than using LUTs to build the subtractors/accumulators needed.

Table 6.10: Resource utilization reported by Vivado for 8×8 HexArray.

Module	**LUTs (53200)**	**Slice Registers (106400)**	**BRAM (140)**
Data Window Controller	2586 (4.8%)	4162 (3.9%)	4 (2.9%)
Array Input Controllers	3416 (6.4%)	670 (0.6%)	0
Array Output Controllers	15522 (29.2%)	41099 (38.6%)	0
Reconfigurable Partitions	17600 (33.1%)	35200 (33.1%)	0
Genome Register	884 (1.7%)	1333 (1.3%)	0
GPIOs	118 (0.2%)	197 (0.2%)	0
Others	715 (1.3%)	745 (0.7%)	0
Total	40841 (76.8%)	83406 (78.4%)	4 (2.9%)

6.3.2 Time Analysis

The time consumed for evolution is the time for generating genomes, which is performed by the GAGA, and the time for evaluating genomes, performed by the reconfigurable hardware core. All PS-PL interfaces used in the implemented system are shown in figure 6.32.

The time consumed for evaluating a genome is the summation of times spent for setting up the HexArray, reconfiguring HexCells, sending the image data, and executing the data.

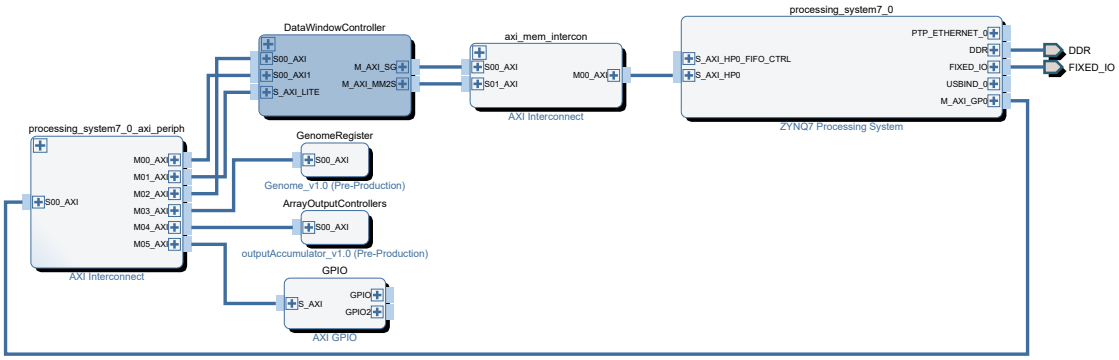


Figure 6.32: DMA and AXI interfaces between the PS and HexArray, generated by Vivado.

The time for setting up the HexArray includes AXI transactions for updating the genome register ($\approx 0.62\mu$ *seconds*) and resetting other modules ($\approx 0.12\mu$ *seconds*). Reconfiguring HexCells includes DMA transactions ($\approx 1.2\mu$ *seconds* each), while the reconfiguration itself took 139μ *seconds* to program a single dynamic region using a 100 MHz clock. This means that reconfiguring 22 partitions took 3.1 milliseconds.

Sending the data to the array occurs automatically when data are delivered to the data window controller, and the controller sends data every two clock cycles. Because a 100 MHz clock was used and propagating $256 \times 256 = 65,536$ pixels required 1.3 milliseconds plus some overhead due to the extra pixels needed for the sliding window of boundary pixels, getting the PS to handle the interrupt and read the fitness data from the AOCs FIFOs took $3.7\mu seconds$. In summary, executing a genome with all other subprocesses can take 4.4 milliseconds, where 70% of that time is used for DPR, and 29.5% went for processing the data and 0.5% as overhead.

Note that the time discussed thus far does not include the time consumed by the GAGA. Although the GAGA was designed to perform most of the processing while the array is processing data, there was a variable time overhead of approximately 1.1 milliseconds.

Additionally, a processing time is needed after every generation, e.g., for selecting parents, calculating the DV and boundboxes, and shifting them. This time is almost fixed regardless of the population size, meaning a slow down for evolution with a smaller population. The time was in the range of 250 milliseconds, which means an overhead of 0.25 milliseconds per genome if the population size was 1000, while it is 5 milliseconds if the population size was 50 genomes. The reason for this slow down on small populations can be explained as follows. Early in the system design phase, "the use of large populations" was an assumption that was made based on a hypothesis that this would allow more diverse genetic operations. Since for every generation, all genetic operators work on the *same* set of parents, preprocessing some of the common operations (e.g., traversing and shifting genomes) on these parents would reduce redundant operations and eventually the time per genome. As smaller populations are desired per Experiment 5, a change to the algorithm may be recommended to avoid any time penalties. Essentially, the algorithm should not preprocess parents and should just process per request.

Finally, the average overall time for generating and evaluating one genome was 5.75 milliseconds when the population size was 1000 and 10.5 milliseconds when the population size was 50. This means that 32,200 genomes (or 1,000,000 evaluations for an 8×8 HexArray) can take approximately three minutes. Figure 6.33 shows evolution traces for 100 independent runs of 50K genomes using small and large populations. These traces exhibit the normalized fitness at a given time; note that the time is on a log_{10} scale. The data show that the best evolved filter can be (equivalent to, better than, or significantly better than) the median filter after approximately (2 seconds, 12 seconds, or 130 seconds) of evolution for a small population size and approximately (5 seconds, 5 seconds, and 40 seconds) for a large population size. Conversely, based on the average of evolved filters, more than 50% of the evolved filters perform better than the median filter after 250 seconds

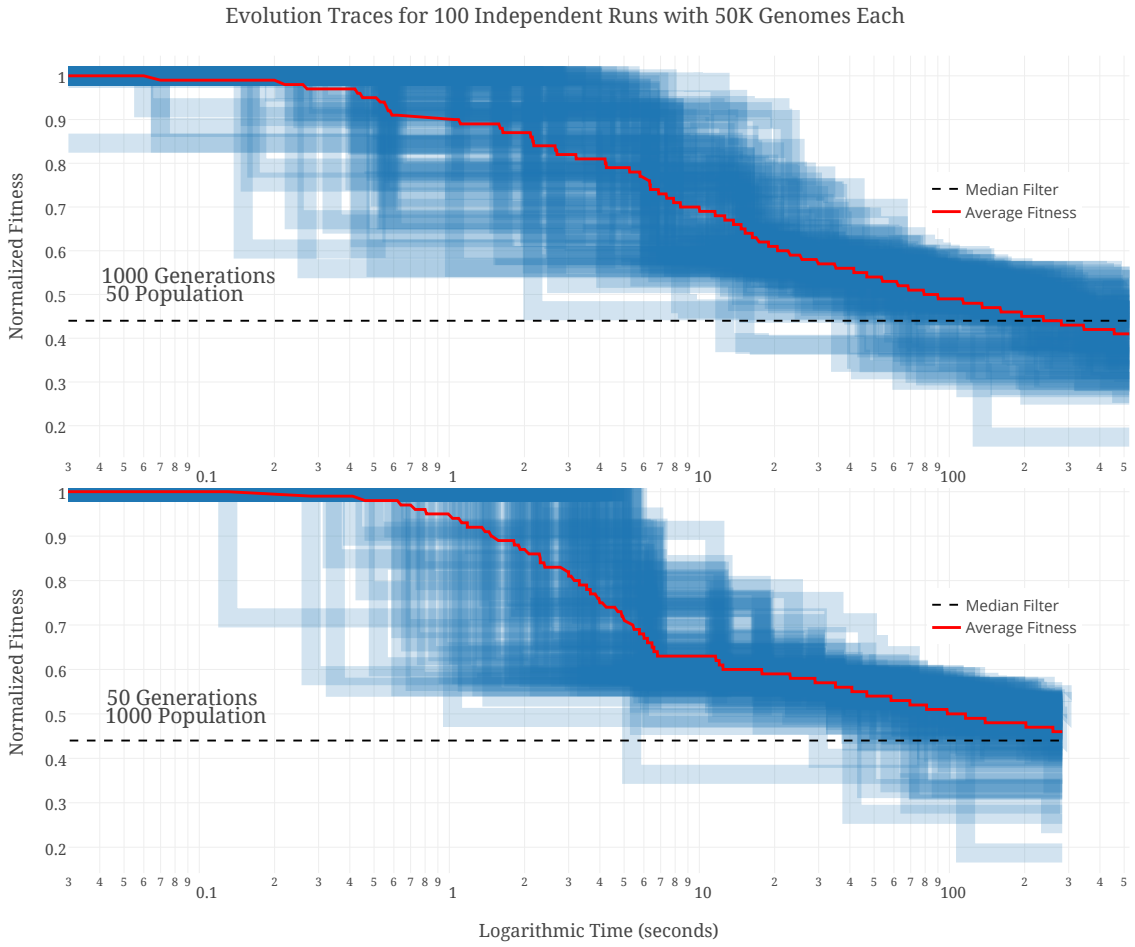


Figure 6.33: Evolution traces for 100 independent runs using (top) 1000 generations and 50 population size or (bottom) 50 generations and 1000 population size. Note that evaluating 50K of genomes using larger populations takes less time. In 2 to 5 seconds, filters comparable to the median filter are evolved. In approximately 250 to 300 seconds, most of the evolved filters outperform the median filter.

for a small population size and 300 seconds for a large population size.

6.4 Summary

This chapter discussed the results of the implemented design. The first part includes a set of experiments that study the system from the perspective of evolution speed. The second part is dedicated to analyzing the implementation details, including resource utilization and timing analysis.

The findings of the seven experiments that were conducted on HexArray are as follows:

- Experiment 1: HexArray converges faster than the state-of-the-art systolic array; this may be a result of many reasons, including improved parallelism, routing flexibility, and improved data propagation. The state-of-the-art systolic array performs significantly better if parallelism is added and slightly better if routing flexibility is added. HexArray outperforms all enhanced versions of the state-of-the-art systolic array.

- Experiment 2: Genomes generated by GAC selection perform better than those generated with no constraints because the search space is reduced.

- Experiment 3: Seven-bit mutation appears to be the best option for traditional mutation in an 8×8 HexArray, whereas it is 4-bit mutation for GAM. Comparing these together, GAM significantly improves the quality of the generated solutions since it concentrates on active chromosomes.

- Experiment 4: GAX running in any of the proposed modes outperforms the traditional crossover. GAX-Cascade performs the best on problems where small improvements are the only improvements possible (e.g., images with a high SNR). GAX-Interleave performs the best on feature extraction tasks where “off/on” chromosomes

(e.g., GRT and DIF) are used since they are swappable. GAX-Parallel performs the best on problems that can be divided into smaller tasks (e.g., impulsive noise).

- Experiment 5: Evolution using small populations (many generations) simulates a depth-search in a subset of the solution space, which results in a slow but consistent improvement. Using large populations (few generations), however, simulate a breadth-search, which results in occasionally finding exceptionally good solutions due to genetic operations operating on a diverse pool of parents.

- Experiment 6: In contrast to traditional filters, evolved filters consistently adapt to the targeted noise level. Evolved filters significantly outperform the median filter for images with a high SNR. The median filter slightly outperforms evolved filters for images with a small SNR. HexArray evolved some noise-level-independent filters using a moderate SNR.

- Experiment 7: HexArray performed well on most of the cases. Poor performances in some problems were justified as some of the noise was irreversible, a wider data window was needed, or the functions needed were more complex than those that HexArray was using. HexArray showed an desirable level of autonomous adaptation. All genetic operators were used efficiently.

The HexArray platform (reconfigurable hardware and the GAGA) was implemented in Zynq-7000 SoC, which has a PL and PS. An 8×8 array consumed 77% of the LUTs, where the majority of the utilized resources were used for the systolic array and the AOCs. PCAP was chosen to perform the DPR because it does not need any hardware instantiations and the reconfiguration performance is adequate for the selected application. The GAGA is implemented as a bare metal application on the PS. The communication between the

PS and PL is through AXI, DMA, and PCAP transactions, whereas the communication between the PS and a high-level monitoring dashboard is through USB and serial ports. Programming all functional units of the designed system takes 3.1 milliseconds. Processing a 256×256 image takes 1.3 milliseconds. The PS overhead for generating and executing a genome is approximately 1.1 milliseconds. Depending on the population size, there is a variable overhead for conducting some of the inter-generation processing. This can be 0.25 milliseconds per genome for large populations or 5 milliseconds per genome for small populations. Consequently, generating and evaluating a genome costs 5.75 milliseconds to 10.25 milliseconds. Overall, HexArray required approximately 3.5 seconds, 8.5 seconds, or 85 seconds to generate filters equivalent to, better than, or significantly better than, respectively, the median filter.

CHAPTER 7

CONCLUSION

Currently, system design is a challenging task for human designers even with the help of software tools. In many practical applications, the only known specifications about the targeted design are its desired behavior. Another challenge is to keep the designed system operating in an unpredicted environment or even when it itself is degraded. Evolvable hardware emerged to solve these challenges. EHW is the means to automate system design and/or allow it to autonomously adapt to changes.

EHW is a hardware system driven by an evolutionary algorithm. The algorithm suggests solutions that are evaluated and assessed by the hardware. Based on their fitnesses, solutions are selected to be mutated or recombined to generate better offspring. However, why can not a software model be used instead of the hardware system?

Using a software model is not practical for many evolution applications. Evaluating a solution in software is slow, e.g., in seconds, and since a large number of solutions, e.g., millions, need to be tested, the performance will be impractical, e.g., months. For example, experiment 3 took 6 days in hardware, but it would take more than 11 years[1] to run on the simulator.

In this dissertation, an evolvable hardware system is proposed – the HexArray platform. HexArray is a modular, scalable, architecture and domain independent, and single-chip platform. The system was successfully implemented using a commercial SoC that in-

[1] 10K genomes $\times$ 5 image groups $\times$ 18 mutation bits $\times$ 100 iterations $\times$ 4 seconds per genome

tegrates an FPGA. The case study used was an image processing application where the system generated adaptive filters.

The robustness of the hardware part of the proposed EHW was proven to be better than that of the state-of-the-art (in experiment 1) even when we boosted the later with some enhancements. The GAGA, the EA part of HexArray, utilizes a variety of genome-aware operators that accelerate evolution. These are GAC selection, GAM and GAX, and all of them were proven to improve (i.e., accelerate) evolution, and some of them were effective on specific problems. These operators are not restricted to HexArray and can be applied to systolic arrays in general. Certain evolution parameters were selected by experiments, e.g., M_{GAM}, population, and so forth. The claimed adaptability of the developed system was proven experimentally. The system adapted to the noise levels and performed consistently well, unlike off-the-shelf solutions. The autonomous behavior of the system was explored by evolving a variety of filters. The system performed quite well considering what was possible with the given function set and data window. In one of the image groups, the noise in an image was 2-pixel-wide dark horizontal stripes, which caused us to predict that any evolved filter had to use the pixel location to be able to cancel the noise. This was not the case, however. The system was able to evolve a simpler filter. The filter was a single-cell solution, using the "maximum" function. It used a vertical 3-pixel-wide slice of the data window, and since the noise is dark (low pixel value) and the slice is guaranteed to have at least one noise-free pixel (as the noise was every other two rows), the filter was possible. It was an example of simple solutions that may be overlooked by (human) designers.

Although HexArray is domain independent, the function set and fitness function are application specific. Selection of the function set is a critical task in constructing an efficient EHW system. A poorly designed function set can cause evolution to diverge, slow down, or be biased. However, this may not be fatal for the system since the system has an intrinsic

adaptation. A practical example of this was encountered during the implementation of the case study. The function set consists of 16 functions, where one of them was defective[2]. This did not break the system. Evolution was slow, but filters were evolved. This issue was uncovered when we observed that the system avoided the defective function, while it should be a popular choice.

Designing the fitness function is the greatest challenge for using EHW systems in an application. The purpose of using EHW to solve complicated design problems is defeated if the design of the fitness function itself is more complex. In our case study, the fitness function used the reference image to calculate the MAE. Note that the system did not require the reference image; rather, the fitness function did. The fitness function can be redesigned to avoid using the reference image, which may not be available. Designing such a fitness function is possible when the application is specified. Consider an example where HexArray is used to enhance images taken by a smartphone camera. The fitness function can be formed by a weighted sum of subfunctions that describe certain properties of the evolved image. These properties describe the quality of the image, such as histogram, exposure, color temperature, and white balance. The challenge in this approach is that the functions need to be computed at high speed, which may require some custom hardware blocks or the use of fitness estimation methods.

HexArray is a powerful computation system. In fact, our image processing system – while it is not optimized – can process a full-HD[3] stream in real time.

In some applications, distinct cells of an array need to execute fixed functions, not randomly selected functions. Typically, since systolic arrays are homogeneous systems, they do not allow this case. However, these applications are natively supported by HexArray

[2]The `AVRG` function had a timing violation, resulting in 0x00 for most of the operations
[3]In 1080p24 standard (1920x1080 pixels in 24 frame/sec)

using GLOBAL_RULES. These are rules and constraints on cells' functions and output ports that all GAGA operators follow without violating. This concept can broaden the range of applications of HexArray. Consider the example of using HexArray for letter classification (a pattern recognition application). In this application, the input data and output data are not the same type of data. The input data may be bits/pixels of the original image. The output data may be a classification (i.e., a letter). In this sense, the rules can constrain cells on the array boundary to be classifiers.

7.1 Future Work

Future work can be directed toward making further improvements to the HexArray implementation proposed in this work or to exploring other general observations. They are outlined as follows:

- HexArray can be optimized for speed. For instance, the PL clock can be boosted from 100 MHz to 250 MHz to obtain a 2.5× speedup. Moreover, performing preprocessing operations per generation for small population sizes is not time efficient; therefore, changing it to be performed per genome as needed would reduce time overhead as the number of genomes is small.

- HexArray can be optimized for space. AOCs are the largest resource consumers, and implementing them using the DSP units, offered in the FPGA, would significantly reduce the resources.

- The evolution speed for the selected image processing application can be accelerated if a tile of the training image is used rather than the entire image. This technique was used to accelerate the simulator and proven to be efficient. One finding was that the

tile has to be constructed by sparse pixels (not neighboring pixels) to improve fitness estimations; for example, a (32×32)-pixel tile was created by selecting a pixel every 8 pixels.

- An observation on experiment 5 was that large populations occasionally yield some exceptionally good solutions, whereas small populations allow slow but consistent improvements. In this sense, we predict that evolution can be improved if it uses a dynamic population size where the population size starts as large and shirks with time.

- One idea to explore is virtual resizing of HexArray, which is achieved by starting evolution using a small array size and increasing it as necessary. This technique can result in an early intensive search in relatively small search spaces, which can be beneficial to evolve shorter solutions that can be optimized for power.

- The shift operation in GAGA is built with the assumption that array inputs have the same significance and hold the same type of data; therefore, shifting a genome does not require shifting the array inputs. For the selected case study where inputs are pixels in a data window, this may be a reasonable assumption. However, for other applications where inputs might have totally different data, GAP should be used to shift a genome along with its array inputs.

- A technique that can improve the quality of generated genomes is by broadcasting genomes. A genome may occupy a subset of cells in the HexArray. The cells that are on the genome boundaries will have some ports that are not used in the active datapath, but they are feeding into the non-used cells of the array. The broadcast operation is performed by altering the selection of these ports to match the ports in

the active datapath such that the ports are copying that same data being propagated through the active datapath.

- A different use case of HexArray is using it in optical character recognition (OCR) application. In this application the EHW system converts an image of typed, hand-written, or printed character to a digital-encoded text (e.g., ASCII code).

  We assume that the input data is the character image which consists of an 8×8 matrix of bits (i.e., 64 white or black bits). Since the bandwidth of the HexCells' functional units is 1 byte, the input data of the HexArray are slices of the character image as shown in figure 7.1.

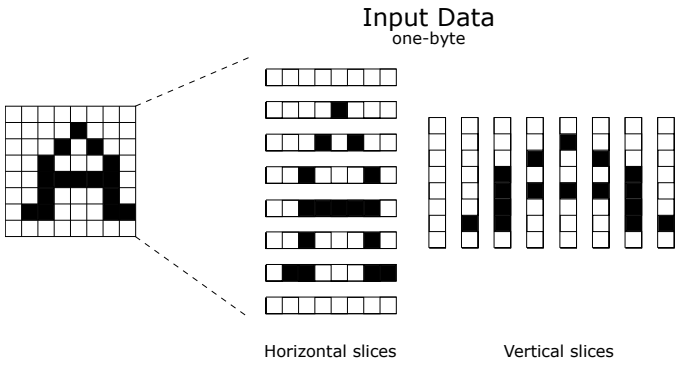


Figure 7.1: Every character is represented by an 8×8 bit matrix (i.e., input data). Vertical or horizontal slices of the input data are fed into the HexArray's AICs.

To enable HexArray for OCR, a function set and a fitness function need to be defined. A suggested function set is presented in table 7.1 where 16 application-specific functions are identified. Each function works on all or some of the HexCell inputs as described by dependency field.

Each output of HexArray is an 8-bit number which is the classification of the input data; in other words, the given character will be classified as one of 256 classifica-

Table 7.1: Function set for an OCR application.

Func. Index	Function Name	Function Description	Depend. A, B, C
f_0	Bitwise OR	OR:=A|B|C	1, 1, 1
f_1	Bitwise AND	AND:=A&B&C	1, 1, 1
f_2	Bitwise XOR	XOR:=A⊕B⊕C	1, 1, 1
f_3	Bitwise XNOR	XNR:=∼(A⊕B⊕C)	1, 1, 1
f_4	Bitwise NOT	NOT:=∼A	1, 0, 0
f_5	Any One	ANY1:=B?0xFF:0x00	0, 1, 0
f_6	Reduction XOR	rXOR:= (⊕C)?0xFF:0x00	0, 0, 1
f_7	Align Right	ALNR:=Shift right A till A[0]≠0	1, 0, 0
f_8	Align Left	ALNL:=Shift left A till A[7]≠0	1, 0, 0
f_9	Maximum	MAX:=max(A,B,C)	1, 1, 1
f_{10}	Minimum	MIN:=min(A,B,C)	1, 1, 1
f_{11}	Count Ones	CNT1:=countOnes(A, B, C)	1, 1, 1
f_{12}	Mix Low	MIXL:=A[0],B[0],A[1],B[1],...,B[3]	1, 1, 0
f_{13}	Mix High	MIXH:=A[4],B[4],A[5],B[5],...,B[7]	1, 1, 0
f_{14}	MUX	MUX:=C[7]?A:B	1, 1, 1
f_{15}	Median	MDIN:=median(A,B,C)	1, 1, 1

tions. A character set is a collection of many images for one character. The target is to minimize the number of *different* character sets classified as a single class. Ideally, the best solution will classify each character set as a single class. An acceptable solution is where one class is classifying one character. In this case, one character can be classified as multiple classes. An undesired solution is where a class represents more than one character. To satisfy all previous requirements, the fitness function is defined as follows:

$$Fitness_{OCR} = \sum_{i=0}^{255} CharacterCounter_i^2,$$

where $CharacterCounter_i$ is the count of how many different character sets are classified as class i. Clearly the lower the fitness value is, the fitter the solution is. To guide

HexArray to avoid the solutions where multiple characters classified as a single class, $CharacterCounter_i$ is raised to the power 2 which increases the fitness exponentially on these undesired cases. An example of the suggested fitness function is shown in figure 7.2 where the fitness is $(1)^2+(1)^2+(0)^2+(1)^2+(1)^2+(1)^2+(0)^2+(1)^2+...+(0)^2+(2)^2$.

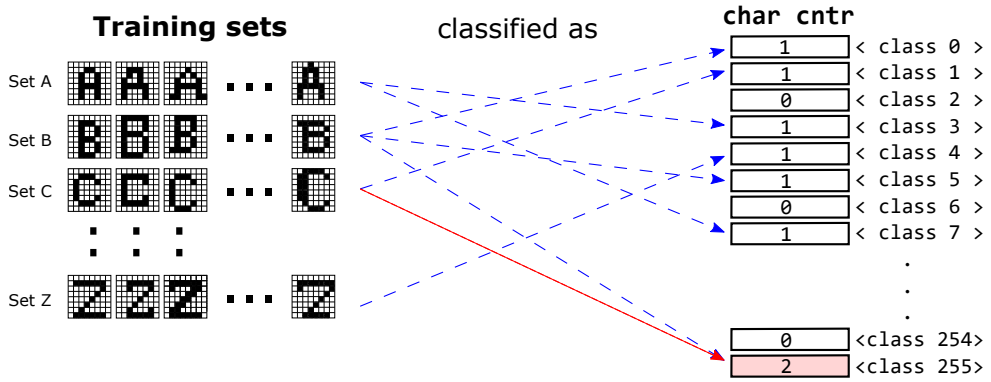


Figure 7.2: Example of characters get classified to one or more classes. The class is the output of HexArray which can hold the value of 0 to 255. An undesired case is when the characters “C” and “B” are classified as class 255.

REFERENCES

[1] G. W. Greenwood and A. M. Tyrrell, *Introduction to Evolvable Hardware: a Practical Guide for Designing Self-Adaptive Systems*. John Wiley & Sons, 2006, vol. 5.

[2] C. Lambert, T. Kalganova, and E. Stomeo, "FPGA-Based Systems for Evolvable Hardware," *World Academy of Science, Engineering and Technology, International Journal of Electrical, Computer, Energetic, Electronic and Communication Engineering*, vol. 1, no. 12, pp. 1890–1896, 2007.

[3] J. Mora, A. Otero, E. d. l. Torre, and T. Riesgo, "Fast and Compact Evolvable Systolic Arrays on Dynamically Reconfigurable FPGAs," in *Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC), 2015 10th International Symposium on*, 2015, Conference Proceedings, pp. 1–7.

[4] M. Ferdjallah, *Introduction to Digital Systems: Modeling, Synthesis, and Simulation Using VHDL*. John Wiley & Sons, 2011.

[5] R. H. Katz, *Contemporary Logic Design*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[6] Z. Navabi, *Digital Design and Implementation with Field Programmable Devices*. Springer Science & Business Media, 2004.

[7] Altera, "MAX V Device Handbook," Altera Inc., San Jose, CA, Tech. Rep., 2011.

[8] Xilinx, "7 Series FPGAs Configurable Logic Block," Xilinx, Inc., San Jose, CA, Catalog, 2016. [Online]. Available: https://www.xilinx.com

[9] P. M. Heysters and G. J. Smit, "Mapping of DSP Algorithms on the MONTIUM Architecture," in *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*. IEEE, 2003, Conference Proceedings, p. 6 pp.

[10] G. Mermoud, A. Upegui, C.-A. Pea, and E. Sanchez, "A Dynamically-Reconfigurable FPGA Platform for Evolving Fuzzy Systems," in *International Work-Conference on Artificial Neural Networks*. Springer, 2005, Conference Proceedings, pp. 572–581.

[11] R. Bittner, P. M. Athanas, and M. Musgrove, "Colt: an Experiment in Wormhole Run-Time Reconfiguration," in *Photonics East'96*. International Society for Optics and Photonics, 1996, Conference Proceedings, pp. 187–194.

[12] J. R. Hauser and J. Wawrzynek, "Garp: a MIPS Processor with a Reconfigurable Coprocessor," in *Field-Programmable Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on*, 1997, Conference Proceedings, pp. 12–21.

[13] M. Trefzer and A. Tyrrell, *Evolvable Hardware: from Practice to Application*. Springer Berlin Heidelberg, 2015. [Online]. Available: https://books.google.com/books?id=-Y2QCgAAQBAJ

[14] A. Abnous, H. Zhang, M. Wan, G. Varghese, V. Prabhu, and J. Rabaey, "The Pleiades Architecture," *The Application of Programmable DSPs in Mobile Communications*, pp. 327–360, 2002.

[15] A. M. Tyrrell, E. Sanchez, D. Floreano, G. Tempesti, D. Mange, J.-M. Moreno, J. Rosenberg, and A. E. Villa, "Poetic Tissue: an Integrated Architecture for Bio-Inspired Hardware," in *International Conference on Evolvable Systems*. Springer, 2003, pp. 129–140.

[16] J. F. Miller, *Cartesian Genetic Programming*. Springer, 2011, pp. 17–34.

[17] H.-T. Kung, "Why Systolic Architectures?" *IEEE computer*, vol. 15, no. 1, pp. 37–46, 1982.

[18] H. Kung and C. E. Leiserson, "Systolic arrays (for VLSI)," in *Sparse Matrix Proceedings 1978*, vol. 1. Society for Industrial and Applied Mathematics, 1979, pp. 256–282.

[19] A. P. Engelbrecht, *Computational Intelligence: an Introduction*. John Wiley & Sons, 2007.

[20] W. Nantian, Q. Yanling, L. Yue, Z. Qingqi, and L. Tingpeng, "Survey on Evolvable Hardware and Embryonic Hardware," in *Electronic Measurement & Instruments (ICEMI), 2013 IEEE 11th International Conference on*, vol. 2, 2013, Conference Proceedings, pp. 1021–1026.

[21] T. Higuchi, M. Iwata, I. Kajitani, H. Iba, Y. Hirao, T. Furuya, and B. Manderick, *Evolvable Hardware and its Application to Pattern Recognition and Fault-Tolerant Systems*. Springer, 1996, pp. 118–135.

[22] R. Tessier, K. Pocek, and A. DeHon, "Reconfigurable Computing Architectures," *Proceedings of the IEEE*, vol. 103, no. 3, pp. 332–354, 2015.

[23] B. Dunham, D. Fridshal, R. Fridshal, and J. H. North, "Design by Natural Selection," *Synthese*, vol. 15, no. 1, pp. 254 – 259, 1963.

[24] T. Kalganova, J. F. Miller, and T. C. Fogarty, *Some Aspects of an Evolvable Hardware Approach for Multiple-Valued Combinational Circuit Design*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 78–89. [Online]. Available: http://dx.doi.org/10.1007/BFb0057609

[25] J. D. Lohn and G. S. Hornby, "Evolvable Hardware: Using Evolutionary Computation to Design and Optimize Hardware Systems," *IEEE Computational Intelligence Magazine*, vol. 1, no. 1, pp. 19–27, 2006.

[26] N. Singh, Poonam, H. Chaturvedi, and K. Honey, "Computational Intelligence in Circuit Synthesis through Evolutionary Algorithms and Particle Swarm Optimization," *International Journal of Advances in Engineering & Technology*, vol. 1, no. 2, pp. 198–205, 2011.

[27] Y. Zhang, S. L. Smith, and A. M. Tyrrell, "Digital Circuit Design using Intrinsic Evolvable Hardware," in *Evolvable Hardware, 2004. Proceedings. 2004 NASA/DoD Conference on*. IEEE, 2004, Conference Proceedings, pp. 55–62.

[28] H. de Garis, "Evolvable Hardware Genetic Programming of a Darwin Machine," in *Artificial Neural Nets and Genetic Algorithms*. Springer, 1993, pp. 441–449.

[29] P. C. Haddow and G. Tufte, "An Evolvable Hardware FPGA for Adaptive Hardware," in *Evolutionary Computation, 2000. Proceedings of the 2000 Congress on*, vol. 1, 2000, Conference Proceedings, pp. 553–560 vol.1.

[30] R. Salvador, A. Otero, J. Mora, E. de la Torre, T. Riesgo, and L. Sekanina, "Self-Reconfigurable Evolvable Hardware System for Adaptive Image Processing," *IEEE Transactions on Computers*, vol. 62, no. 8, pp. 1481–1493, 2013.

[31] B. K. Hall, *Evolution: Principles and Processes*. Jones & Bartlett Publishers, 2011.

[32] J. Felsenstein, "Inbreeding and Variance Effective Numbers in Populations with Overlapping Generations," *Genetics*, vol. 68, no. 4, p. 581, 1971.

[33] H. Kitano, *Morphogenesis for Evolvable Systems*. Springer, 1996, pp. 99–117.

[34] A. Thompson, "Silicon Evolution," in *Proceedings of the 1st Annual Conference on Genetic Programming*. MIT press, 1996, Conference Proceedings, pp. 444–452.

[35] A. Stoica, R. Zebulum, and D. Keymeulen, "Mixtrinsic Evolution," in *International Conference on Evolvable Systems*. Springer, 2000, Conference Proceedings, pp. 208–217.

[36] A. J. Greensted and A. M. Tyrrell, "Extrinsic Evolvable Hardware on the RISA Architecture," in *International Conference on Evolvable Systems*. Springer, 2007, Conference Proceedings, pp. 244–255.

[37] T. Kalganova, "An Extrinsic Function-Level Evolvable Hardware Approach," in *European Conference on Genetic Programming*. Springer, 2000, Conference Proceedings, pp. 60–75.

[38] T. Kalganova and J. Miller, "Evolving More Efficient Digital Circuits by Allowing Circuit Layout Evolution and Multi-Objective Fitness," in *Evolvable Hardware, 1999. Proceedings of the First NASA/DoD Workshop on*. IEEE, 1999, Conference Proceedings, pp. 54–63.

[39] A. Thompson, *Hardware Evolution: Automatic Design of Electronic Circuits in Reconfigurable Hardware by Artificial Evolution*. Springer Science & Business Media, 2012.

[40] S. Ando and H. Iba, "Analog Circuit Design with a Variable Length Chromosome," in *Evolutionary Computation, 2000. Proceedings of the 2000 Congress on*, vol. 2. IEEE, 2000, Conference Proceedings, pp. 994–1001.

[41] T. Higuchi, M. Iwata, D. Keymeulen, H. Sakanashi, M. Murakawa, I. Kajitani, E. Takahashi, K. Toda, N. Salami, and N. Kajihara, "Real-World Applications of Analog and Digital Evolvable Hardware," *IEEE transactions on evolutionary computation*, vol. 3, no. 3, pp. 220–235, 1999.

[42] K.-J. Kim and S.-B. Cho, "Automated Synthesis of Multiple Analog Circuits Using Evolutionary Computation for Redundancy-Based Fault-Tolerance," *Applied Soft Computing*, vol. 12, no. 4, pp. 1309–1321, 2012.

[43] J. R. Koza, "Human-Competitive Results Produced by Genetic Programming," *Genetic Programming and Evolvable Machines*, vol. 11, no. 3-4, pp. 251–284, 2010.

[44] A. Stoica, D. Keymeulen, R. Zebulum, A. Thakoor, T. Daud, Y. Klimeck, R. Tawel, and V. Duong, "Evolution of Analog Circuits on Field Programmable Transistor Arrays," in *Evolvable Hardware, 2000. Proceedings. The Second NASA/DoD Workshop on*. IEEE, 2000, Conference Proceedings, pp. 99–108.

[45] F. Cancare, M. D. Santambrogio, and D. Sciuto, "A Direct Bitstream Manipulation Approach for Virtex4-Based Evolvable Systems," in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, 2010, Conference Proceedings, pp. 853–856.

[46] V. Coimbra and M. V. Lamar, "Design and Optimization of Digital Circuits by Artificial Evolution Using Hybrid Multi Chromosome Cartesian Genetic Programming," in *International Symposium on Applied Reconfigurable Computing*. Springer, 2016, pp. 195–206.

[47] L. Huelsbergen, E. Rietman, and R. Slous, "Evolving Oscillators in Silico," *IEEE Transactions on Evolutionary Computation*, vol. 3, no. 3, pp. 197–204, 1999.

[48] D. Levi and S. A. Guccione, "GeneticFPGA: Evolving Stable Circuits on Mainstream FPGA Devices," in *Evolvable Hardware, 1999. Proceedings of the First NASA/DoD Workshop on*. IEEE, 1999, Conference Proceedings, pp. 12–17.

[49] E. Stomeo, T. Kalganova, and C. Lambert, "Generalized Disjunction Decomposition for Evolvable Hardware," *IEEE Trans Syst Man Cybern B Cybern*, vol. 36, no. 5, pp. 1024–1043, 2006. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/17036810

[50] A. Thompson, "On the Automatic Design of Robust Electronics Through Artificial Evolution," in *International Conference on Evolvable Systems*. Springer, 1998, Conference Proceedings, pp. 13–24.

[51] A. Upegui and E. Sanchez, "Evolving Hardware with Self-Reconfigurable Connectivity in Xilinx FPGAs," in *First NASA/ESA Conference on Adaptive Hardware and Systems (AHS'06)*. IEEE, 2006, Conference Proceedings, pp. 153–162.

[52] Z. Vasicek, L. Sekanina, and M. Bidlo, "A Method for Design of Impulse Bursts Noise Filters Optimized for FPGA Implementations," in *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 2010, Conference Proceedings, pp. 1731–1736.

[53] K. Glette, J. Torresen, and M. Hovin, "Intermediate Level FPGA Reconfiguration for an Online EHW Pattern Recognition System," in *Adaptive Hardware and Systems, 2009. AHS 2009. NASA/ESA Conference on*. IEEE, 2009, Conference Proceedings, pp. 19–26.

[54] K. Glette, J. Torresen, M. Yasunaga, and Y. Yamaguchi, "On-Chip Evolution Using a Soft Processor Core Applied to Image Recognition," in *First NASA/ESA Conference on Adaptive Hardware and Systems (AHS'06)*. IEEE, 2006, Conference Proceedings, pp. 373–380.

[55] J. Torresen, G. A. Senland, and K. Glette, "Partial Reconfiguration Applied in an On-Line Evolvable Pattern Recognition System," in *NORCHIP, 2008.* IEEE, 2008, Conference Proceedings, pp. 61–64.

[56] R. S. Oreifej, R. N. Al-Haddad, H. Tan, and R. F. DeMara, "Layered Approach to Intrinsic Evolvable Hardware Using Direct Bitstream Manipulation of Virtex II Pro Devices," in *2007 International Conference on Field Programmable Logic and Applications*. IEEE, 2007, Conference Proceedings, pp. 299–304.

[57] L. Sekanina, "Evolutionary Functional Recovery in Virtual Reconfigurable Circuits," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 3, no. 2, p. 8, 2007.

[58] A. M. Tyrrell, G. Hollingworth, and S. L. Smith, "Evolutionary Strategies and Intrinsic Fault Tolerance," in *Evolvable Hardware, 2001. Proceedings. The Third NASA/DoD Workshop on*. IEEE, 2001, Conference Proceedings, pp. 98–106.

[59] G. Hollingworth, S. Smith, and A. Tyrrell, "The Intrinsic Evolution of Virtex Devices Through Internet Reconfigurable Logic," in *International Conference on Evolvable Systems*. Springer, 2000, Conference Proceedings, pp. 72–79.

[60] E. Yadegari and S. M. Fakhraie, "Implementation of Image Processing Applications with Evolutionary Fault Recovery Scheme," in *2014 22nd Iranian Conference on Electrical Engineering (ICEE)*, 2014, Conference Proceedings, pp. 458–462.

[61] L. Sterpone, M. Porrmann, and J. Hagemeyer, "A Novel Fault Tolerant and Runtime Reconfigurable Platform for Satellite Payload Processing," *IEEE Transactions on Computers*, vol. 62, no. 8, pp. 1508–1525, 2013.

[62] D. Dasgupta and Z. Michalewicz, *Evolutionary Algorithms in Engineering Applications*. Springer Science & Business Media, 2013.

[63] R. Dunkley, "Supporting a Wide Variety of Communication Protocols Using Partial Dynamic Reconfiguration," in *2012 IEEE AUTOTESTCON Proceedings*, 2012, Conference Proceedings, pp. 120–125.

[64] D. S. Linden, "Optimizing Signal Strength In-Situ Using an Evolvable Antenna System," in *Evolvable Hardware, 2002. Proceedings. NASA/DoD Conference on*. IEEE, 2002, pp. 147–151.

[65] R. A. Sutton, V. P. Srini, and J. M. Rabaey, "A Multiprocessor DSP System using PADDI-2," in *Proceedings of the 35th annual Design Automation Conference*. ACM, 1998, Conference Proceedings, pp. 62–65.

[66] A. K. W. Yeung and J. M. Rabaey, "A Reconfigurable Data-Driven Multiprocessor Architecture for Rapid Prototyping of High Throughput DSP Algorithms," in *System Sciences, 1993, Proceeding of the Twenty-Sixth Hawaii International Conference on*, vol. i, 1993, Conference Proceedings, pp. 169–178 vol.1.

[67] F. Gruau, *Neural Network Synthesis Using Cellular Encoding and the Genetic Algorithm*. Université de Lyon 1, 1994. [Online]. Available: https://books.google.com/books?id=PTn5rQEACAAJ

[68] M. Murakawa, S. Yoshizawa, I. Kajitani, and T. Higuchi, "Evolvable Hardware for Generalized Neural Networks," in *Proceedings of the Fifteenth International Joint Conference on Artifical Intelligence-Volume 2*. Morgan Kaufmann Publishers Inc., 1997, Conference Proceedings, pp. 1146–1151.

[69] A. Upegui, Y. Thoma, H. F. Satizbal, F. Mondada, P. Rtornaz, Y. Graf, A. Perez-Uribe, and E. Sanchez, "Ubichip, Ubidule, and Marxbot: a Hardware Platform for the Simulation of Complex Systems," in *International Conference on Evolvable Systems*. Springer, 2010, Conference Proceedings, pp. 286–298.

[70] X. Yao, "Evolutionary Artificial Neural Networks," *International journal of neural systems*, vol. 4, no. 03, pp. 203–222, 1993. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/8293227

[71] X. Yao and Y. Liu, "A New Evolutionary System for Evolving Artificial Neural Networks," *IEEE Transactions on Neural Networks*, vol. 8, no. 3, pp. 694–713, May 1997.

[72] I. Kajitani, T. Hoshino, N. Kajihara, M. Iwata, and T. Higuchi, "An Evolvable Hardware Chip and Its Application As a Multi-Function Prosthetic Hand Controller," in *Proceedings of the Sixteenth National Conference on Artificial Intelligence and the Eleventh Innovative Applications of Artificial Intelligence Conference Innovative Applications of Artificial Intelligence*. American Association for Artificial Intelligence, 1999, Conference Proceedings, pp. 182–187.

[73] D. Keymeulen, M. Iwata, Y. Kuniyoshi, and T. Higuchi, "Online Evolution for a Self-Adapting Robotic Navigation System Using Evolvable Hardware," *Artificial Life*, vol. 4, no. 4, pp. 359–393, 1998.

[74] T. Murali, S. Perumal, R. Mohan, and P. Palanisamy, "Design and Synthesis of Six Legged Walking Robot Using Single Degree of Freedom Linkage," *Imperial Journal of Interdisciplinary Research*, vol. 2, no. 3, 2016.

[75] A. F. Winfield and J. Timmis, *Evolvable Robot Hardware*. Springer, 2015, pp. 331–348.

[76] K. Glette, J. Torresen, P. Kaufmann, and M. Platzner, "A Comparison of Evolvable Hardware Architectures for Classification Tasks," in *International Conference on Evolvable Systems*. Springer, 2008, Conference Proceedings, pp. 22–33.

[77] P. Kaufmann, K. Glette, T. Gruber, M. Platzner, J. Torresen, and B. Sick, "Classification of Electromyographic Signals: Comparing Evolvable Hardware to Conventional Classifiers," *IEEE Transactions On Evolutionary Computation*, vol. 17, no. 1, pp. 46–63, 2013.

[78] H. Sakanashi, M. Salami, M. Iwata, S. Nakaya, T. Yamauchi, T. Inuo, N. Kajihara, and T. Higuchi, "Evolvable Hardware Chip for High Precision Printer Image Compression," in *AAAI/IAAI*, 1998, Conference Proceedings, pp. 486–491.

[79] M. Salami, M. Murakawa, and T. Higuchi, "Data Compression Based on Evolvable Hardware," in *International Conference on Evolvable Systems*. Springer, 1996, Conference Proceedings, pp. 167–179.

[80] N. Nedjah and L. de Macedo Mourelle, "Secure Evolvable Hardware for Public-Key Cryptosystems," *New Generation Computing*, vol. 23, no. 3, pp. 259–275, 2005.

[81] S. Picek, "Evolutionary Computation and Cryptology," in *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*. ACM, 2016, Conference Proceedings, pp. 883–909.

[82] S. Picek, D. Sisejkovic, V. Rozic, B. Yang, D. Jakobovic, and N. Mentens, "Evolving Cryptographic Pseudorandom Number Generators," in *International Conference on Parallel Problem Solving from Nature*. Springer, 2016, Conference Proceedings, pp. 613–622.

[83] M. A. Lones and S. L. Smith, *Medical Applications of Evolvable Hardware*. Springer, 2015, pp. 253–271.

[84] R. Dobai and L. Sekanina, "Low-Level Flexible Architecture with Hybrid Reconfiguration for Evolvable Hardware," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 8, no. 3, pp. 1–24, 2015.

[85] B. Hutchings, B. Nelson, and M. J. Wirthlin, "Designing and Debugging Custom Computing Applications," *IEEE Design & Test of Computers*, vol. 17, no. 1, pp. 20–28, 2000.

[86] N. Steiner, A. Wood, H. Shojaei, J. Couch, P. Athanas, and M. French, "Torc: Towards an Open-Source Tool Flow," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. ACM, 2011, pp. 41–44.

[87] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings, "Rapidsmith: Do-It-Yourself CAD Tools for Xilinx FPGAs," in *2011 21st International Conference on Field Programmable Logic and Applications*. IEEE, 2011, Conference Proceedings, pp. 349–355.

[88] T. M. Mitchell, *Machine Learning*. McGraw-Hill, Inc., 1997.

[89] B. Osterloh, H. Michalik, S. A. Habinc, and B. Fiethe, "Dynamic Partial Reconfiguration in Space Applications," in *Adaptive Hardware and Systems, 2009. AHS 2009. NASA/ESA Conference on*, 2009, Conference Proceedings, pp. 336–343.

[90] A. Otero, R. Salvador, J. Mora, E. d. l. Torre, T. Riesgo, and L. Sekanina, "A Fast Reconfigurable 2D HW Core Architecture on FPGAs for Evolvable Self-Adaptive Systems," in *Adaptive Hardware and Systems (AHS), 2011 NASA/ESA Conference on*, 2011, Conference Proceedings, pp. 336–343.

[91] C. Rossmeissl, A. Sreeramareddy, and A. Akoglu, "Partial Bitstream 2-D Core Relocation for Reconfigurable Architectures," in *Adaptive Hardware and Systems, 2009. AHS 2009. NASA/ESA Conference on*. IEEE, 2009, pp. 98–105.

[92] I. Monolithic Memories, "Monolithic Memories Announces: a Revolution in Logic Design," *Electronic Design*, vol. 26, pp. 148B–148C, March, 18, 1978 1978.

[93] K. Vipin and S. A. Fahmy, "ZyCAP: Efficient Partial Reconfiguration Management on the Xilinx Zynq," *IEEE Embedded Systems Letters*, vol. 6, no. 3, pp. 41–44, 2014.

[94] H. Amano, "A Survey on Dynamically Reconfigurable Processors," *IEICE transactions on Communications*, vol. 89, no. 12, pp. 3179–3187, 2006.

[95] A. Marshall, T. Stansfield, I. Kostarnov, J. Vuillemin, and B. Hutchings, "A Reconfigurable Arithmetic Array for Multimedia Applications," in *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*. ACM, 1999, Conference Proceedings, pp. 135–143.

[96] N. Voros, A. Rosti, and M. Hübner, *Dynamic System Reconfiguration in Heterogeneous Platforms: The MORPHEUS Approach*. Springer Science & Business Media, 2009, vol. 40.

[97] B. Plunkett and J. Watson, "Adapt2400 ACM Architecture Overview, QuickSilver Technology Inc," *San Jose, Jan*, 2004.

[98] S. Azizi, F. Safaei, and N. Hashemi, "On the Topological Properties of HyperX," *The Journal of Supercomputing*, vol. 66, no. 1, pp. 572–593, 2013.

[99] P. Q. S. Guide, "Updated: 14, dec, 2013 "http://www.adapteva.com/wp-content/uploads/Parallella-Quick-Start-Guide_rev3.pdf"," *Last access*, vol. 15, 2015.

[100] D. Dye, "Partial Reconfiguration of Xilinx FPGAs Using ISE Design Suite," *White Paper*, 2012.

[101] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, and R. W. Stewart, *The Zynq Book: Embedded Processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC*. UK: Strathclyde Academic Media, 2014.

[102] K. Vipin and S. A. Fahmy, "Mapping Adaptive Hardware Systems with Partial Reconfiguration Using CoPR for Zynq," in *Adaptive Hardware and Systems (AHS), 2015 NASA/ESA Conference on*, 2015, Conference Proceedings, pp. 1–8.

[103] M. A. Kadi, P. Rudolph, D. Gohringer, and M. Hubner, "Dynamic and Partial Reconfiguration of Zynq 7000 under Linux," in *2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, 2013, Conference Proceedings, pp. 1–5.

[104] M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, D. Sweely, and D. Lopresti, "Building and Using a Highly Parallel Programmable Logic Array," *Computer*, vol. 24, no. 1, pp. 81–89, 1991.

[105] J. Becker, T. Pionteck, and M. Glesner, *DReAM: A Dynamically Reconfigurable Architecture for Future Mobile Communication Applications*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 312–321. [Online]. Available: http://dx.doi.org/10.1007/3-540-44614-1_34

[106] X. Wang and S. G. Ziavras, "A MultiprocessoronaProgrammableChip Reconfigurable System for Matrix Operations with PowerGrid Case Studies," *International Journal of Computational Science and Engineering*, vol. 10, no. 1-2, pp. 181–191, 2015.

[107] C. Ebeling, D. C. Cronquist, and P. Franklin, "RaPiD-Reconfigurable Pipelined Datapath," in *International Workshop on Field Programmable Logic and Applications*. Springer, 1996, Conference Proceedings, pp. 126–135.

[108] E. Mirsky and A. DeHon, "MATRIX: a Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources," in *FPGAs for Custom Computing Machines, 1996. Proceedings. IEEE Symposium on*, 1996, Conference Proceedings, pp. 157–166.

[109] N. J. Macias, "The PIG Paradigm: the Design and Use of a Massively Parallel Fine Grained Self-Reconfigurable Infinitely Scalable Architecture," in *Evolvable Hardware, 1999. Proceedings of the First NASA/DoD Workshop on*, 1999, Conference Proceedings, pp. 175–180.

[110] R. W. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger, "Using the KressArray for Reconfigurable Computing," in *Photonics East (ISAM, VVDC, IEMB)*. International Society for Optics and Photonics, 1998, pp. 150–161.

[111] R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger, "KressArray Xplorer: a New CAD Environment to Optimize Reconfigurable Datapath Array Architectures," in *Design Automation Conference, 2000. Proceedings of the ASP-DAC 2000. Asia and South Pacific*, 2000, Conference Proceedings, pp. 163–168.

[112] M. Herz, T. Hoffmann, U. Nageldinger, and C. Schreiber, "Interfacing the MoM-PDA to an Internet-Based Development System," in *Systems Sciences, 1999. HICSS-32. Proceedings of the 32nd Annual Hawaii International Conference on*, vol. Track3, 1999, Conference Proceedings, p. 7 pp.

[113] D. C. Chen, "Programmable arithmetic devices for high speed digital signal processing," Ph.D. dissertation, EECS Department, University of California, Berkeley, 1992. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/1992/2033.html

[114] D. Alnajjar, H. Konoura, Y. Ko, Y. Mitsuyama, M. Hashimoto, and T. Onoye, "Implementing Flexible Reliability in a Coarse-Grained Reconfigurable Architecture," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 12, pp. 2165–2178, 2013.

[115] J. A. Walker, M. A. Trefzer, S. J. Bale, and A. M. Tyrrell, "PAnDA: A Reconfigurable Architecture that Adapts to Physical Substrate Variations," *IEEE Transactions on Computers*, vol. 62, no. 8, pp. 1584–1596, 2013.

[116] K. DeJong, "An Analysis of the Behavior of a Class of Genetic Adaptive Systems," *Ph. D. Thesis, University of Michigan*, 1975.

[117] L. B. Booker, "Intelligent Behavior As an Adaptation to the Task Environment," Ph.D. dissertation, University of Michigan, Ann Arbor, MI, USA, 1982, aAI8214966.

[118] D. E. Goldberg and K. Deb, "A Comparative Analysis of Selection Schemes Used in Genetic Algorithms," *Foundations of genetic algorithms*, vol. 1, pp. 69–93, 1991.

[119] M. Srinivas and L. M. Patnaik, "Adaptive Probabilities of Crossover and Mutation in Genetic Algorithms," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 24, no. 4, pp. 656–667, 1994.

[120] J. Cervantes and C. R. Stephens, ""Optimal" Mutation Rates for Genetic Search," in *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '06. New York, NY, USA: ACM, 2006, pp. 1313–1320. [Online]. Available: http://doi.acm.org/10.1145/1143997.1144201

[121] H. Mühlenbein, "How Genetic Algorithms Really Work: Mutation and Hillclimbing," in *PPSN*, vol. 92, 1992, Conference Proceedings, pp. 15–26.

[122] M. Srinivas and L. M. Patnaik, "Genetic Algorithms: a Survey," *Computer*, vol. 27, no. 6, pp. 17–26, 1994.

[123] S. Tsutsui, M. Yamamura, and T. Higuchi, "Multi-Parent Recombination with Simplex Crossover in Real Coded Genetic Algorithms," in *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation-Volume 1*. Morgan Kaufmann Publishers Inc., 1999, Conference Proceedings, pp. 657–664.

[124] G. Syswerda, "Uniform Crossover in Genetic Algorithms," in *Proceedings of the 3rd International Conference on Genetic Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1989, pp. 2–9. [Online]. Available: http://dl.acm.org/citation.cfm?id=645512.657265

[125] W. M. Spears, "Adapting Crossover in Evolutionary Algorithms," in *Evolutionary Programming*, 1995, Conference Proceedings, pp. 367–384.

[126] J. Vasconcelos, J. A. Ramirez, R. Takahashi, and R. Saldanha, "Improvements in Genetic Algorithms," *IEEE Transactions on magnetics*, vol. 37, no. 5, pp. 3414–3417, 2001.

[127] Y. Jin, "A Comprehensive Survey of Fitness Approximation in Evolutionary Computation," *Soft computing*, vol. 9, no. 1, pp. 3–12, 2005.

[128] A. I. Esparcia-Alcázar and J. Moravec, "Fitness Approximation for Bot Evolution in Genetic Programming," *Soft Computing*, vol. 17, no. 8, pp. 1479–1487, 2013. [Online]. Available: http://dx.doi.org/10.1007/s00500-012-0965-7

[129] M. Colby, T. Duchow-Pressley, J. J. Chung, and K. Tumer, "Local Approximation of Difference Evaluation Functions," in *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*. International Foundation for Autonomous Agents and Multiagent Systems, 2016, pp. 521–529.

[130] P. Yan and H. Takagi, "Comparative Study on Fitness Landscape Approximation with Fourier Transform," in *2012 Sixth International Conference on Genetic and Evolutionary Computing*, Aug 2012, pp. 400–403.

[131] T. Kuyucu, M. Trefzer, A. Greensted, J. Miller, and A. Tyrrell, "Fitness Functions for the Unconstrained Evolution of Digital Circuits," in *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, June 2008, pp. 2584–2591.

[132] J. H. Holland, "Genetic Algorithms," *Scientific american*, vol. 267, no. 1, pp. 66–72, 1992.

[133] H.-G. Beyer and H.-P. Schwefel, "Evolution Strategies – A Comprehensive Introduction," *Natural computing*, vol. 1, no. 1, pp. 3–52, 2002.

[134] J. R. Koza, *Genetic Programming: on the Programming of Computers by Means of Natural Selection*. MIT press, 1992, vol. 1.

[135] R. Storn and K. Price, "Differential Evolution–A Simple and Efficient Heuristic for Global Optimization Over Continuous Spaces," *Journal of global optimization*, vol. 11, no. 4, pp. 341–359, 1997.

[136] D. Floreano, P. Dürr, and C. Mattiussi, "Neuroevolution: from Architectures to Learning," *Evolutionary Intelligence*, vol. 1, no. 1, pp. 47–62, 2008.

[137] P. L. Lanzi, W. Stolzmann, and S. W. Wilson, *Learning Classifier Systems: from Foundations to Applications*. Springer, 2003.

[138] T. BDack, F. Hoffmeister, and H. Schwefel, "A Survey of Evolution Strategies," in *Proceedings of the 4th International Conference on Genetic Algorithms*, 1991, pp. 2–9.

[139] J. R. Koza, *Genetic Programming III: Darwinian Invention and Problem Solving*. Morgan Kaufmann, 1999, vol. 3.

[140] J. F. Miller, "An Empirical Study of the Efficiency of Learning Boolean Functions Using a Cartesian Genetic Programming Approach," in *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation-Volume 2*. Morgan Kaufmann Publishers Inc., 1999, pp. 1135–1142.

[141] S. L. Harding, J. F. Miller, and W. Banzhaf, "Self-Modifying Cartesian Genetic Programming," in *Cartesian Genetic Programming*. Springer, 2011, pp. 101–124.

[142] J. Miller and A. Turner, "Cartesian Genetic Programming," in *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*. ACM, 2015, pp. 179–198.

[143] A. J. Turner and J. F. Miller, "Recurrent Cartesian Genetic Programming," in *International Conference on Parallel Problem Solving from Nature*. Springer, 2014, pp. 476–486.

[144] P. Winston, *Artificial Intelligence*, ser. A-W Series in Computerscience. Addison-Wesley Publishing Company, 1992. [Online]. Available: https://books.google.com/books?id=b4owngEACAAJ

[145] R. Shonkwiler, "Parallel Genetic Algorithms." in *ICGA*. Citeseer, 1993, pp. 199–205.

[146] R. Hinterding, Z. Michalewicz, and T. C. Peachey, "Self-Adaptive Genetic Algorithm for Numeric Functions," in *International Conference on Parallel Problem Solving from Nature*. Springer, 1996, pp. 420–429.

[147] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II," *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.

[148] D. Goldberg, K. Deb, and B. Korb, "Messy Genetic Algorithms: Motivation, Analysis, and First Results," *Complex systems*, vol. 3, no. 3, pp. 493–530, 1989.

[149] G. R. Harik, "Learning Gene Linkage to Efficiently Solve Problems of Bounded Difficulty Using Genetic Algorithms," Ph.D. dissertation, PhD thesis, University of Michigan, Ann Arbor, Ann Arbor, MI, USA, 1997, uMI Order No. GAX97-32090.

[150] H. Kargupta, "Gene Expression: the Missing Link in Evolutionary Computation," Los Alamos National Lab., NM (United States), Tech. Rep., 1997.

[151] T. Higuchi, M. Iwata, I. Kajitani, H. Yamada, B. Manderick, Y. Hirao, M. Murakawa, S. Yoshizawa, and T. Furuya, "Evolvable Hardware with Genetic Learning," in *Circuits and Systems, 1996. ISCAS '96., Connecting the World., 1996 IEEE International Symposium on*, vol. 4, 1996, Conference Proceedings, pp. 29–32 vol.4.

[152] M. Murakawa, S. Yoshizawa, I. Kajitani, T. Furuya, M. Iwata, and T. Higuchi, *Hardware Evolution at Function Level*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 62–71. [Online]. Available: http://dx.doi.org/10.1007/3-540-61723-X_970

[153] J. C. Gallagher, S. Vigraham, and G. Kramer, "A Family of Compact Genetic Algorithms for Intrinsic Evolvable Hardware," *IEEE Transactions on Evolutionary Computation*, vol. 8, no. 2, pp. 111–126, 2004.

[154] E. Stomeo, T. Kalganova, and C. Lambert, "A Novel Genetic Algorithm for Evolvable Hardware," in *2006 IEEE International Conference on Evolutionary Computation*. IEEE, 2006, Conference Proceedings, pp. 134–141.

[155] R. Huan-Huan, P. Xu-Dong, and T. Jun-Lin, "Research on Evolvable Hardware Based on Population Hybridization Monkey-King Genetic Algorithm," in *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*, 2015, Conference Proceedings, pp. 665–668.

[156] K. Li, A. Fialho, S. Kwong, and Q. Zhang, "Adaptive Operator Selection with Bandits for a Multiobjective Evolutionary Algorithm Based on Decomposition," *IEEE Transactions on Evolutionary Computation*, vol. 18, no. 1, pp. 114–130, 2014.

[157] A. Thompson, P. Layzell, and R. S. Zebulum, "Explorations in Design Space: Unconventional Electronics Design Through Artificial Evolution," *IEEE Transactions on Evolutionary Computation*, vol. 3, no. 3, pp. 167–196, 1999.

[158] J. Torresen, "Increased Complexity Evolution Applied to Evolvable Hardware," in *Smart Engineering System Design: Neural Networks, Fuzzy Logic, Evolutionary Programming, Data Mining, and Complex Systems, Proceedings of ANNIE*, vol. 99, 1999.

[159] T. Kalganova, "Bidirectional Incremental Evolution in Extrinsic Evolvable Hardware," in *Evolvable Hardware, 2000. Proceedings. The Second NASA/DoD Workshop on*. IEEE, 2000, pp. 65–74.

[160] F. Cancare, S. Bhandari, D. B. Bartolini, M. Carminati, and M. D. Santambrogio, "A bird's eye view of FPGA-based Evolvable Hardware," in *Adaptive Hardware and Systems (AHS), 2011 NASA/ESA Conference on*, 2011, Conference Proceedings, pp. 169–175.

[161] J. Huang, M. Parris, J. Lee, and R. F. Demara, "Scalable FPGA-Based Architecture for DCT Computation using Dynamic Partial Reconfiguration," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 9, no. 1, p. 9, 2009.

[162] W. Lie and W. Feng-Yan, "Dynamic Partial Reconfiguration in FPGAs," in *Third International Symposium on Intelligent Information Technology Application*, vol. 2, 2009, Conference Proceedings, pp. 445–448.

[163] K. S. Prasada Kumari, "Self-Adaptive Image Processing Using Blind Image Quality Assessment Technique," *Perspectives in Science*, vol. 8, pp. 639–641, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2213020916301823

[164] D. Vernekar, G. Malhotra, and V. Colaco, "Reconfigurable FPGA Using Genetic Algorithm," in *Proceedings of the International Conference and Workshop on Emerging Trends in Technology*. ACM, 2010, pp. 493–497.

[165] W. M. Gentleman and H. Kung, "Matrix Triangularization by Systolic Arrays," in *25th Annual Technical Symposium*. International Society for Optics and Photonics, 1982, pp. 19–26.

[166] H. Kung and P. L. Lehman, "Systolic (VLSI) Arrays for Relational Database Operations," in *Proceedings of the 1980 ACM SIGMOD International Conference on Management of Data*. ACM, 1980, pp. 105–116.

[167] J. D. Crisman and J. A. Webb, "The Warp Machine on Navlab," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 13, no. 5, pp. 451–465, 1991.

[168] G. M. Megson and I. M. Bland, "Generic Systolic Array for Genetic Algorithms," *IEE Proceedings - Computers and Digital Techniques*, vol. 144, no. 2, p. 107, 1997.

[169] R. Salvador, A. Otero, J. Mora, E. d. l. Torre, T. Riesgo, and L. Sekanina, "Evolvable 2D Computing Matrix Model for Intrinsic Evolution in Commercial FPGAs with Native Reconfiguration Support," in *Adaptive Hardware and Systems (AHS), 2011 NASA/ESA Conference on*, 2011, Conference Proceedings, pp. 184–191.

[170] J. L. A. van de Snepscheut and J. B. Swenker, "On the Design of Some Systolic Algorithms," *Journal of the ACM*, vol. 36, no. 4, pp. 826–840, 1989. [Online]. Available: http://dl.acm.org/citation.cfm?doid=76359.76365

[171] R. Salvador, A. Otero, J. Mora, E. d. l. Torre, L. Sekanina, and T. Riesgo, "Fault Tolerance Analysis and Self-Healing Strategy of Autonomous, Evolvable Hardware Systems," in *2011 International Conference on Reconfigurable Computing and FPGAs*, 2011, Conference Proceedings, pp. 164–169.

[172] R. Salvador, A. Otero, J. Mora, E. d. l. Torre, T. Riesgo, and L. Sekanina, "Implementation Techniques for Evolvable HW Systems: Virtual vs. Dynamic Reconfiguration," in *22nd International Conference on Field Programmable Logic and Applications (FPL)*, 2012, Conference Proceedings, pp. 547–550.

[173] A. Gallego, J. Mora, A. Otero, E. de la Torre, and T. Riesgo, "A Scalable Evolvable Hardware Processing Array," in *2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, 2013, Conference Proceedings, pp. 1–7.

[174] R. McGill, J. W. Tukey, and W. A. Larsen, "Variations of Box Plots," *The American Statistician*, vol. 32, no. 1, pp. 12–16, 1978. [Online]. Available: http://amstat.tandfonline.com/doi/abs/10.1080/00031305.1978.10479236

[175] D. Whitley, "A Genetic Algorithm Tutorial," *Statistics and computing*, vol. 4, no. 2, pp. 65–85, 1994.

[176] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, and R. W. Stewart, *The Zynq Book: Embedded Processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC*. Strathclyde Academic Media, 2014.

[177] R. Oomen, T. Nguyen, A. Kumar, and H. Corporaal, “An Automated Technique to Generate Relocatable Partial Bitstreams for Xilinx FPGAs,” in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, 2015, Conference Proceedings, pp. 1–4.

[178] C. Kohn, “Partial Reconfiguration of a Hardware Accelerator on Zynq-7000 All Programmable SoC Devices,” *Xilinx, XAPP1159 (v1. 0)*, 2013.

APPENDIX A

IMAGE PROCESSING

A.1 Introduction

This appendix is dedicated for providing details for all image groups used in the experiments. Each image group consists of a pair of images; a training image used as an input to the EHW system and a reference image required by the fitness function (MAE function). These images were selected from a commonly used image processing library[1]. All images are gray-scale with size of 256×256 pixels .

A.2 Image Groups:

S&P 25% and S&P 10%

This image group consists of a noise-free reference image and a noisy training image. The intention here is to evolve a filter that cleans noise in the training image. The training image has a 25% (or 10%) impulsive noise. Impulsive noise (also called fat-tail distribution or Salt and Pepper) is modeled by giving the minimum or maximum pixel values to randomly selected pixels; in other words, the image consists of random black and white pixels. The initial fitness (i.e., training image fitness), fitness per pixel, and the peak SNR are presented in table A.1.

Table A.1: Properties of 10% and 25% Salt and Pepper noise images.

Image group	**Fitness (MAE)**	**Fitness/pixel**	**PSNR (dB)**
S&P 25%	2102943	32.1	11.1
S&P 10%	837642	12.8	15.2

[1] `http://www.imageprocessingplace.com/root_files_V3/image_databases.htm`

(a) Reference image (Lena) **(b)** Training image (with S&P 25%) **(c)** Training image (with S&P 10%)

Figure A.1: S&P 25% and S&P 10% image groups.

EdgeDetect

This image group consists of a noise-free training image and a reference image that represents the detected edge. Edge detection is the feature extraction method of detecting sharp changes in neighboring pixels.

(a) Reference image with edge detection **(b)** Training image

Figure A.2: EdgeDetect image group.

Thresholding

Binary image is created by a thresholding method, where pixels are converted to black or white based on a threshold typically midway (e.g., 128 in gray-scale). In other words, if the pixel value is higher than 128 then it is assigned to the value 255 and to 0 otherwise.

(a) Reference image with thresholding at 128

(b) Training image

(c) Reference image

(d) Training image Gaussian noise

Figure A.3: (Top) Thresholding image group. (Bottom) Gaussian image group.

Gaussian

Gaussian image group consists of a noisy training image and a noise-free reference image. Gaussian noise is the noise where it is Gaussian-distributed. The used training image has a

high SNR.

Table A.2: Properties of EdgaDetect, Thresholding, and Gaussian image groups.

Image group	**Fitness**	**Fitness/pixel**	**PSNR (dB)**
EdgaDetect	7881392	120.3	N/A
Thresholding	5576351	85.1	N/A
Gaussian	493012	7.5	28.6

Image groups with different impulsive noise levels

In experiment 6 (section 6.2.6), 18 images were used. Half of them were different noise levels for one image (Lena) and the other half was for another image (Cameraman), called S&P X% and S&P2 X% respectively. These image groups properites are summarized in table A.3 and table A.4. The images are shown in figure A.4 and figure A.5.

Table A.3: Properties of Lena image with different levels of impulsive noise.

Image group	**Fitness**	**Fitness/pixel**	**PSNR (dB)**
S&P 2.5%	215519	3.3	21.4
S&P 5%	423036	6.5	18.1
S&P 7.5%	639059	9.8	16.4
S&P 10%	837642	12.8	15.2
S&P 15%	1267799	19.3	13.4
S&P 20%	1667570	25.4	12.2
S&P 30%	2524533	38.5	10.3
S&P 40%	3349595	51.1	9.1
S&P 50%	4167287	63.6	8.2

(a) Training image (with S&P 2.5%) **(b)** Training image (with S&P 5%) **(c)** Training image (with S&P 7.5%)

(d) Training image (with S&P 10%) **(e)** Training image (with S&P 15%) **(f)** Training image (with S&P 20%)

(g) Training image (with S&P 30%) **(h)** Training image (with S&P 40%) **(i)** Training image (with S&P 50%)

Figure A.4: Lena image with different levels of impulsive noise.

(a) Training image (with S&P2 2.5%) **(b)** Training image (with S&P2 5%) **(c)** Training image (with S&P2 7.5%)

(d) Training image (with S&P2 10%) **(e)** Training image (with S&P2 15%) **(f)** Training image (with S&P2 20%)

(g) Training image (with S&P2 30%) **(h)** Training image (with S&P2 40%) **(i)** Training image (with S&P2 50%)

Figure A.5: Cameraman image with different levels of impulsive noise.

Table A.4: Properties of Cameraman image with different levels of impulsive noise.

Image group	**Fitness**	**Fitness/pixel**	**PSNR (dB)**
S&P2 2.5%	215905	3.3	20.9
S&P2 5%	412825	6.3	18.2
S&P2 7.5%	657073	10.0	16.1
S&P2 10%	843664	12.9	15
S&P2 15%	1259571	19.2	13.3
S&P2 20%	1683935	25.7	12
S&P2 30%	2496520	38.1	10.3
S&P2 40%	3324956	50.7	9.1
S&P2 50%	4207501	64.2	8.1

Experiment 7 image groups

To explore the proposed system, variety of image groups have been used in experiment 7. These 16 image groups are summarized in table A.5. The image groups are shown in figure A.6 to A.21.

(a) Reference image (blurred) **(b)** Training image

Figure A.6: Blurring image group.

(a) Reference image **(b)** Training image

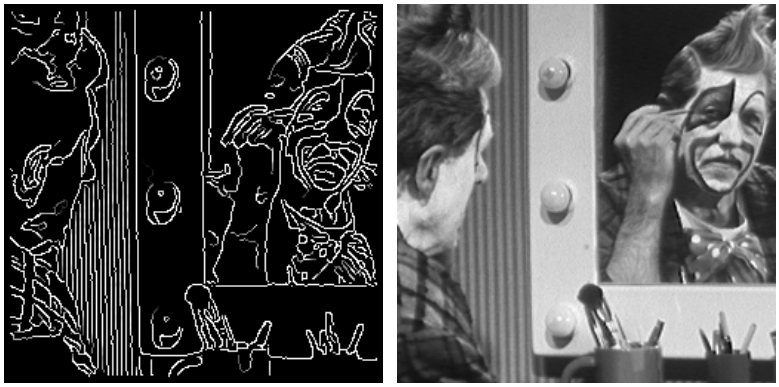
Figure A.7: Deblurring image group.

Table A.5: Properties of experiment 7 image groups.

Image group	**Fitness**	**Fitness/pixel**	**Noise or feature type**
Blurring	1119057	17.1	Gaussian blur filter with σ=6
Deblurring	1119057	17.1	Gaussian blur filter with σ=6
Edge detection (Roberts)	5579355	85.1	Edge detection using Roberts cross operator
Edge detection (Canny)	7867293	120.0	Edge detection using Canny with threshold=255, 1; Gaussian=1
Edge detection (Sobel)	7781534	118.7	Edge detection using Sobel operator
Gradient adjustment	1776295	27.1	Gradient vertically, light on the top and dark on the bottom
Periodic dark rows	1338242	20.4	Periodic dark horizontal 2-pixel lines
Histogram equalization	2807804	42.8	Bright pixels are brighter and dark pixels are darker
Morphological (erosion)	1836910	28.0	Erosion with a square structuring element = 9
White balancing	6027898	92.0	Bright image
Blob detection (Laplacian)	3672245	56.0	Laplacian filter (4-connected)
Contrast adjustment	3747206	57.2	Poor contrast image
Darkness equalization	2344732	35.8	Dark image
Brightness equalization	6633420	101.2	Bright image
Depixelate	766329	11.7	Image constructed by resolving every 4×4 pixels to one pixel
Periodic dark columns	686973	10.5	Periodic vertical lines darkened by Fourier Transform with a period of eight pixels.

(a) Reference image **(b)** Training image

Figure A.8: Edge detection (Roberts) image group.



(a) Reference image **(b)** Training image

Figure A.9: Edge detection (Canny) image group.

(a) Reference image **(b)** Training image

Figure A.10: Edge detection (Sobel) image group.

(a) Reference image **(b)** Training image

Figure A.11: Gradient adjustment image group.

(a) Reference image **(b)** Training image

Figure A.12: Periodic dark rows image group.

(a) Reference image **(b)** Training image

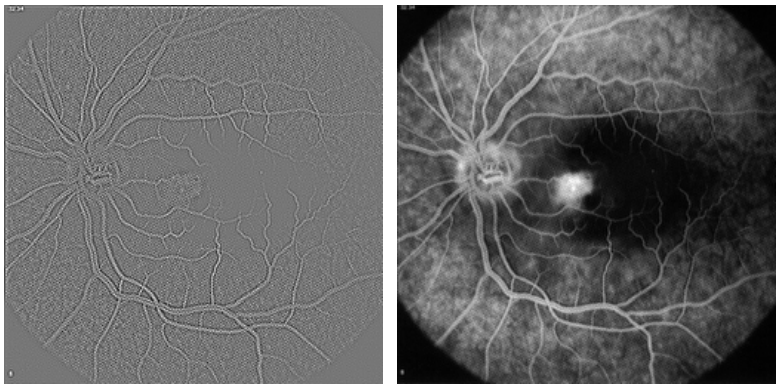
Figure A.13: Histogram equalization image group.

(a) Reference image **(b)** Training image

Figure A.14: Morphological (erosion) image group.

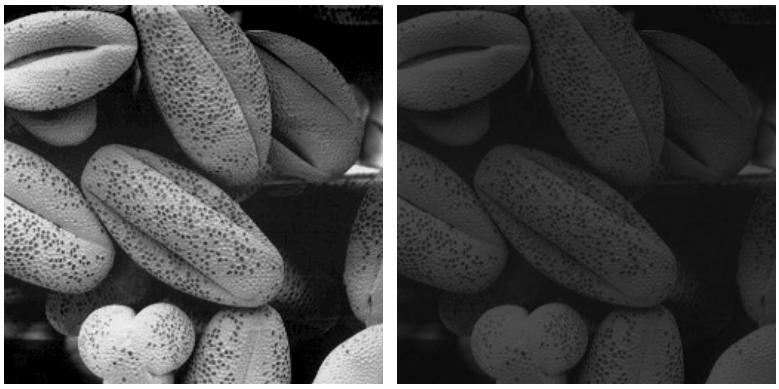
(a) Reference image **(b)** Training image

Figure A.15: White balancing image group.



(a) Reference image **(b)** Training image

Figure A.16: Blob detection (Laplacian) image group.



(a) Reference image **(b)** Training image

Figure A.17: Contrast adjustment image group.

(a) Reference image **(b)** Training image

Figure A.18: Darkness equalization image group.

(a) Reference image **(b)** Training image

Figure A.19: Brightness equalization image group.

(a) Reference image **(b)** Training image

Figure A.20: De-pixelate image group.

(a) Reference image **(b)** Training image

Figure A.21: Periodic dark columns image group.