Imperial College of Science, Technology and Medicine
Department of Electrical and Electronic Engineering

Reconfigurable Computing for Genetic Algorithms

Liucheng Guo

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Electrical and Electronic Engineering and
the Diploma of Imperial College London, September 2016

Declaration of Originality

This thesis is a presentation of my original research work. The contributions of others are involved, every effort is made to indicate this clearly in the references to the literature and acknowledgement of collaborative research.

Copyright Declaration

Abstract

Genetic Algorithms (GAs) are a class of numerical and combinatorial optimisers which are especially useful for solving non-linear and non-convex problems. However, the required execution time often limits their application to small-scale or latency-insensitive problems, so techniques to increase the speed and computational efficiency of GAs are needed. FPGA-based acceleration has significant potential for speeding up genetic algorithms, but existing FPGA-based GA systems are limited by the need for extensive hardware architecture customisation by end users and the relatively small improvement in performance. The main reasons for these limitations are that most of the units in these GA systems are fixed and the algorithms of these FPGA-based systems are just inherited from software GAs without making full use of reconfigurable hardware.

This thesis examines the previous work and then rethinks the creation of genetic algorithms with reconfigurable computing, and aims to address the problems of low-level programmability, flexibility, and performance in current GA systems. The thesis first presents a general-purpose automated framework for creating and executing a flexible parallel GA system on one FPGA or even multiple FPGAs, which increases the flexibility in GA architecture and removes the low-level hardware programmability barrier for non-expert users. The thesis then proposes a novel hardware-oriented approach called Pipelined Genetic Propagation (PGP), which is intrinsically spatially distributed and fully pipelined. PGP represents a GA solver as a graph of loosely coupled genetic operators, which allows the system to be scaled to the available resources, and also to dynamically change connection topology at run-time to explore different strategies. Finally the thesis extends PGP in three ways: Recursive PGP for multiple level optimisation problems, machine learning guided PGP for automatically changing the connection topology, and partially reconfigurable PGP to increase flexibility. Overall, the thesis presents new ways to perform genetic algorithms in reconfigurable computing, with high-level programmability, sufficient flexibility and high performance.

Acknowledgements

First and foremost, I would like to express my sincere appreciation towards my supervisors Dr. David Thomas and Prof. Wayne Luk. With their immense amount of help, guidance and support, I started my PhD research and obtained some achievements. Both Dr. David Thomas and Prof. Wayne Luk helped me plan and improve my academic papers and thesis, and guided me to the right direction of my research. I meet Dr. David Thomas every week and attend group meeting with Prof. Wayne Luk every month during the four years in Imperial College, we had a lot of conversations and I could not imagine how I could finish my PhD study without either of them.

I am very pleased and honoured to have been given the opportunity to study in the two groups in Imperial College, namely Circuits and Systems Group in Department of Electrical and Electronic Engineering and Custom Computing Group in Department of Computing. I want to thank all the students and staffs in the two groups. Thanks are specially given to Thomas Chau, Ce Guo, Xinyu Niu, Ingrid Funie, Shengjia Shao, Kit Cheung and Jinzhe Yang in Custom Computing Group. We have had invaluable discussions and collaborated in many projects. For example, I cooperated with Shengjia Shao on the research of bilevel optimisation and recursive pipelined genetic propagation. I also want to thank my past and present colleagues in Circuits and Systems Group, especially Kan Shi, Gordon Inggs, Shuanglong Liu, Qiang Li, Jiang Su, Hilda and Marlon Wijeyasinghe. With all of them, I have had a wonderful life in Imperial College.

I also want to thank Yao Lu, who contributes her patience, understanding and care during my hours of research, contemplation and writing. I feel really lucky to have Yao with me.

I want to thank my family. My parents give me the best education they could and make me become who I am. My sister Kejun, and her husband Yu also support me a lot in these years.

Special thanks are also given to the financial supports from UK and China, providing me sufficient resources during my four years' life in Imperial College.

Contents

List of Tables

List of Figures

Glossary

- FPGA: Field Programmable Gate Array.
- ASIC: Application-Specific Integrated Circuit.
- FR: Full Reconfiguration.
- PR: Partial Reconfiguration.
- GA: Genetic Algorithm.
- pGA: Parallel Genetic Algorithm.
- dGA: Distributed Genetic Algorithm.
- cGA: Cellular Genetic Algorithm.
- RNG: Random Number Generator.
- SCM: Selection, Crossover and Mutation Unit.
- MAXSAT: Maximum Satisfiability Problem.
- SCP: Set Covering Problem.
- TSP: Travelling Salesman Problem.
- PGP: Pipelined Genetic Propagation. Pipelined Genetic Propagation is an optimisation algorithm, architecture, and strategy, allowing users to create a hardware-specific graph-based GA optimiser for a chosen objective function. (see Chapter 5)

- RPGP: Recursive Pipelined Genetic Propagation. (see section 6.1)
- QPGP: Q-learning Guided Pipelined Genetic Propagation. (see section 6.2)
- PPGP: Partial Reconfigurable Pipelined Genetic Propagation. (see section 6.3)

Chapter 1

Introduction

This thesis explores reconfigurable computing for genetic algorithms (GAs), first looking at the designs and optimisation of classic hardware-based GA, and then proposing new approaches for FPGA-based GA systems. This chapter focuses on the motivation and objectives of this thesis, and then presents the contributions from chapter 3 to chapter 6, finally lists all my published papers in 1.3, on which the thesis is based.

1.1 Motivation and Objectives

Optimisation problems are common in the real-world, where the aim is to find the optimum solution from a number of candidates. Genetic algorithms (GAs) are a family of nature-inspired search algorithms for general-purpose optimisation. A genetic algorithm finds good solutions to a problem by mimicking the natural evolutionary process, and uses mutation, crossover, and selection to improve the overall fitness of a pool of candidate solutions. However, natural evolution is a slow process, so the GA execution time is significant when a problem is of large-scale or has a complicated search space. Users also need to specify a series of GA parameters to tune the evolutionary process, meaning a GA system requires the execution platform to have a high flexibility. To allow the use of GAs in a wider range of practical problems, researchers have explored many ways of improving efficiency and flexibility, including FPGA-based accelerators.

However, most of the previous FPGA-based GA systems use low-level programmability and lack flexibility in GA tuning, or have a low performance.

This thesis reviews related hardware-based GA systems carefully, aiming to design a new GA system on FPGAs for researchers and developers, with the following objectives:

- Objective 1: high-level programmability. Implementing a GA system on an FPGA is not as easy as writing a GA software for a CPU, especially for the parallel GA system running on one or multiple FPGA platforms. The design and implementation process needs users to have a deep understanding of hardware, which restricts the usage of GA accelerators to non-expert users.
- Objective 2: architectural flexibility. The resources on an FPGA are limited, requiring the developer to carefully tune the architecture used in a GA system. The tuning is hard for non-expert users, and even for experts it is still difficult to modify the previous GA systems, because most of their architectures are fixed.
- Objective 3: run-time flexibility. Genetic algorithm has a group of parameters which must be specified, in order to configure the evolutionary process. However, in hardware most modifications need time-consuming recompilation and verification. So FPGA-based systems should support sufficient run-time parameters, to allow post-compilation tuning.
- Objective 4: high performance. Genetic algorithms can sometimes find reasonable solutions in a short time on CPUs, but there is still a potential to improve performance in FPGAs and get even better solutions in less time. The full pipeline and high parallelism in FPGA can be used to reduce the execution time. In addition, the reconfiguration on FPGAs not only supplies high flexibility, it could also bring benefits to the evolutionary process.

This thesis meets all the objectives using the following ways:

1. High level inputs for objective 1. The thesis develops a general-purpose framework, which reads a high-level input and produces hardware-based generic GA automatically. The

framework also provides architectural and run-time parameters. This removes the barrier of low-level hardware implementation from non-expert users.

2. Architectural parameters for objective 2. The architecture of generated GA kernel in an FPGA can be tuned by a set of compile-time architectural parameters.

3. Run-time parameters and run-time configuration for objective 3. The run-time parameters allow the users to modify the GA system without time-consuming recompilation and verification. The run-time reconfiguration enables the users to configure the running system, partially or completely.

4. Pipelined and parallel architecture for objective 4. Our fully pipelined structures can significantly increase performance of a hardware system, while the parallelism can reduce the execution time and often improve the solution quality of GAs.

5. New ways of escaping from local optima for objective 4. Genetic algorithms may get stuck in local optima, but we provide more ways to escape from them, in order to increase performance or find better quality solution.

1.2 Contributions

The thesis puts all the objectives in mind when designing the genetic algorithms on FPGAs. The contributions of the thesis are as follows:

- A general-purpose GA framework for creating and running one or multiple GA instances with an effective communication scheme on one or multiple FPGAs. (See Chapter 3 and 4)

- A customisable GA architecture allowing non-expert users to specify problem settings and architectural options in a high level without hardware design expertise, reaching objective 1 and 2. (See Chapter 3 and 4)

- A fine-tuning mechanism for the GA architecture, enabling users to tune a set of run-time parameters without time-consuming recompilation and verification of the design, meeting objective 3. (See Chapter 3, 4 and 5)

- A hardware-oriented way of structuring genetic algorithms as a graph of parallel pipelined components with local candidate storage, breaking the data dependency existing in generational GAs and eliminating the centralised memory bottleneck, meeting objective 4. (See Chapter 5)

- A hardware-oriented strategy working with machine learning, multiple level parallelism, partial reconfiguration, and local optimisation, to maximise the flexibility and performance, meeting objective 4. (See Chapter 6 and 7)

The structure of the thesis is:

- Chapter 1 introduces the objectives, motivation, and the publication list during my research period.

- Chapter 2 describes the background of reconfigurable computing and genetic algorithms, and reviews all the related work to the topic of this thesis.

- Chapter 3 designs a general purpose GA framework, which supports good programmability, architectural flexibility and run-time flexibility.

- Chapter 4 extends the framework in chapter 3 by adding external parallelism to GA instances, thus increases performance.

- Chapter 5 proposes a novel strategy called Pipelined Genetic Propagation (PGP), introducing graph theory and additional flexibility into GA. This system is fully pipelined and highly parallel, resulting in high performance; it also uses a new methodology to change the evolutionary process, helping to effectively find good solutions.

- Chapter 6 extends the PGP by using multiple level parallelism, machine learning and run-time partial reconfiguration, which further improves performance and flexibility.

- Chapter 7 summarises the whole thesis and presents the future research directions.

1.3 Publications

Publications during my research period are listed in the two following subsections, namely journal articles and conference papers. My name is also highlighted in the lists.

1.3.1 Journal Articles

The journal articles published in the PhD period include:

Guo, Liucheng, Andreea-Ingrid Funie, David Thomas, and Wayne Luk. "General Purpose Framework for FPGA-accelerated Genetic Algorithms", International Journal of Bio-Inspired Computation Vol. 7, No. 6, pp. 361-375, 2015. (Related to Chapter 3 and Chapter 4)

Z. Xie, X. Liang, **Guo, Liucheng**, A. Kitamoto, M. Tamura, T. Shiroishi, D. Gillies. "Automatic Classification Framework for Ventricular Septal Defects: a pilot study on high-throughput mouse embryo cardiac phenotyping", Journal of Medical Imaging, Vol. 2, No. 4, 041003, 2015.

Guo, Liucheng, David B. Thomas, and Wayne Luk. "Automated Framework for General-Purpose Genetic Algorithms in FPGAs". Applications of Evolutionary Computation, Vol. 8602, pp. 714-725. Springer Berlin Heidelberg, 2014. (Related to Chapter 3 and Chapter 4)

1.3.2 Conference Papers

The conference papers published during my research period include:

Andreea Ingrid Funie, **Guo, Liucheng**, Xinyu Niu, Wayne Luk, and Mark Salmon,"Custom Framework for Run-time Trading Strategies", 13th International Symposium on Applied Reconfigurable Computing, 2017

Shao, Shengjia, **Guo, Liucheng**, Ce Guo, Thomas CP Chau, David B. Thomas, Wayne Luk, and Stephen Weston. "Recursive pipelined genetic propagation for bilevel optimisation", In 25th International Conference on Field Programmable Logic and Applications (FPL), pp. 1-6. IEEE, 2015. (Related to Chapter 6)

Guo, Liucheng, Andreea-Ingrid Funie, David Thomas, Haohuan Fu, and Wayne Luk. "Parallel Genetic Algorithms on Multiple FPGAs", ACM SIGARCH Computer Architecture News, Issue 43, No. 4, pp. 86-93, 2015. (Related to Chapter 4)

Guo, Liucheng, Ce Guo, Andreea-Ingrid Funie, David Thomas, Wayne Luk. "Pipelined Genetic Propagation with Q-Learning", In Proc. of Metaheuristics International Conference, 2015. (Related to Chapter 6)

Guo, Liucheng, Ce Guo, David B. Thomas, and Wayne Luk. "Pipelined Genetic Propagation", In 23rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 103-110, 2015. (Best Paper Candidate and HiPEAC Paper Award, Related to Chapter 5)

Guo, Liucheng, David B. Thomas, Ce Guo, and Wayne Luk. "Automated framework for FPGA-based parallel genetic algorithms", In 24th IEEE International Conference on Field Programmable Logic and Applications (FPL), pp. 1-7, 014. (Related to Chapter 4)

Guo, Liucheng, David B. Thomas, and Wayne Luk. "Customisable architectures for the set covering problem." ACM SIGARCH Computer Architecture News 41, no. 5, pp. 101-106, 2014. (Related to Chapter 3)

Guo, Liucheng, Jiangfang Yi, Liang Zhang, Xiaoyin Wang, and Dong Tong. "CGA: Combining cluster analysis with genetic algorithm for regression suite reduction of microprocessors." In IEEE International SOC Conference (SOCC), pp. 207-212. 2011. (Related to Chapter 3)

Chapter 2

Background and Related Work

The previous chapter introduced the motivation and objectives of this thesis. The topic of the thesis is to make full use of reconfigurable hardware for genetic algorithms, in order to provide a fast and flexible GA system. Genetic algorithms are a common and useful approach for many optimisation problems, but they need the execution platform to have high computational power and high flexibility in tuning GAs' parameters. Researchers have used hardware platforms to meet this performance and flexibility requirements, including microprocessors, Graphics Processing Units (GPUs), Application-Specific Integrated Circuits (ASICs) and FPGAs. FPGAs can provide high flexibility of microprocessors as well as high performance of ASICs and GPUs. This thesis presents a framework to quickly produce and execute genetic algorithms on FPGAs, and also proposes new hardware-oriented strategies to perform evolutionary computing. The main goals of this thesis are to meet the challenges of genetic algorithms such as efficiency, as well as to fully utilise FPGAs' features in terms of flexibility and performance.

Before moving to the proposed designs in the next chapters, this chapter describes the key background knowledge and summarises related work as follows:

- Section 2.1 briefly introduces optimisation problems, to show the circumstances where genetic algorithms are commonly used.
- Section 2.2 describes the genetic algorithms in details, and analyses their advantages and

challenges.

- Section 2.3 compares popular hardware platforms such as FPGAs, GPUs, ASICs and microprocessors for applications' acceleration, and then focuses on the FPGAs, such as theirs architectures, high performance design, high flexibility, as well as challenges and developments.

- Section 2.4 summarises previous hardware-based genetic algorithms and describes some examples for future comparison. From their limitations analysed, a flexible, fast and easy-to-use tool for the hardware-based GAs is needed.

2.1 Overview of the Optimisation Problem

Optimisation is a process to find the most desirable solutions from the input such as parameters for the target function [HH04]. The searching process must consider that there exists more than one solution and these solutions are not equally valued [HH04]. Some optimisation problems aim to find the optimum, while some aim to find the near-optimal solutions. For example, a simple optimisation problem is to find a minimal value of the objective function $f(x)$, where vector x is an element of the input set, and has to meet all the constraints of $g(x)$. For a simple continuous optimisation problem, the mathematical formula is as follows:

$$\begin{aligned} &\min \ f(x) \\ &f(x) : \mathbb{R}^n \rightarrow \mathbb{R} \\ &\text{subject to : } \ g_i(x) \leq c_i, 0 \leq i \leq n \end{aligned} \tag{2.1}$$

Where c_i is the boundary or the limit for the constraint function $g_i(x)$. Figure 2.1 shows two simple optimisation problems. They both aim to find a solution x to reach the minimum of $f(x)$, while the ranges of x are the constraints. As shown in Figure 2.1, there is normally only one global minimum, but may exist many local minima. A search process may get stuck in the local minima, and return a local optimum.

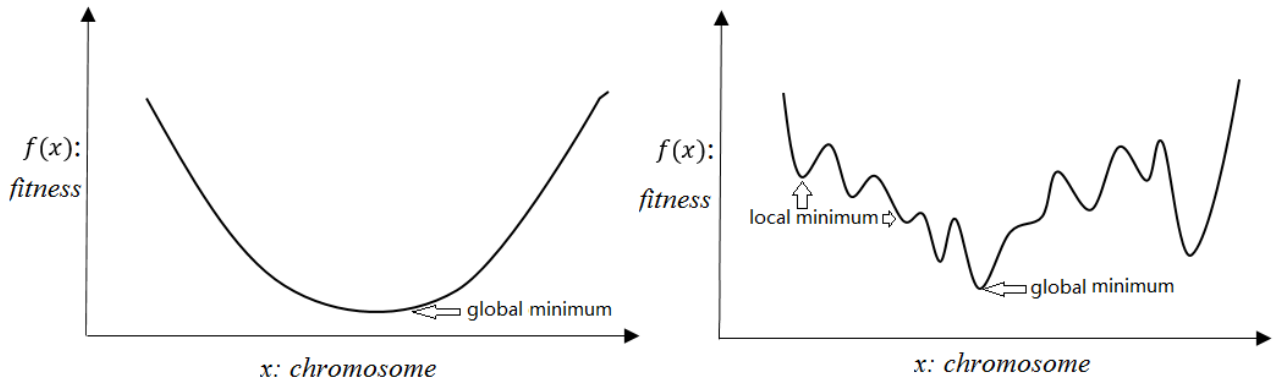


Figure 2.1: Two Simple Optimisation Problems.
The two problems are to find the optimum solution x from a restricted range $\{x\}$ for the objective function $f(x)$. Global minimum: the minimal fitness of $f(x)$, Local minimum: a minimal fitness in a subset of the $f(x)$

Optimisation problems are classified as continuous or combinatorial, depending on the type of input [PS82]. A combinatorial optimisation problem is to find an optimal object from a finite set, and the object can be integer, permutation and so on, while a continuous problem has an infinite number of potential solutions. The optimisation problems can also have multiple parameters, i.e. more than one x is involved, as well as multiple objectives, such as multiple-level optimisation, which will be discussed in Chapter 6. They are likely to increase the difficulty of the optimisation process, and many are transformed into a series of simple optimisation approaches.

Optimisation problems are common in many disciplines and various domains [Rot11], including mechanics, economics, electrical engineering, operations research, and molecular modelling. Classical problems include the travelling salesman problem, maximal satisfaction problems, set covering problems and bi-level optimisation problems. Some combinatorial optimisation problems are NP-hard, meaning they cannot be solved in polynomial time [PS82].

There exist several popular approaches to solve optimisation problems, including exhaustive search, iterative methods, and heuristics. For combinatorial inputs, exhaustive search examines every possible solution. For continuous inputs, the method needs an extra pre-examining step: it samples the input into a number of possible solutions before examining them. Exhaustive approach does not get stuck in any local optimum, but it is rarely used [HH04]. The reasons are that the approach needs large computational power when the search space is large, and

it may miss the global optimum for continuous optimisation problems because of sampling [HH04]. Iterative methods converge to a solution by generating improved approximate solutions, such as Newton's method, the Quasi-Newton method, and Gradient descent. However, these approaches are likely to get stuck in local optima when a problem has many local minima. Heuristics rank the alternative solutions and choose one branch to follow, they usually achieve good solutions in shorter time than other classic optimisation methods [Rot11]. For example, Figure 2.1 shows two optimisation problems, one is convex and the other is non-convex. The first one is easy to solve, a simple iterative method can find the solution quickly. However, the second one has many local minima, so some approaches such as simple iterative methods may get stuck in a local optimum, while heuristics may provide a near-optimum solution quickly.

Popular heuristic algorithms include genetic algorithms, Tabu search and simulated annealing. Genetic algorithms learn from natural evolution to improve initial solution candidates generation by generation, Tabu search uses local search in the metaheuristic searching process [GL13], while simulated annealing uses temperature as a strategy to guide the search in the solution space [SMEH14]. Genetic algorithms work for both continuous and combinatorial problems, and suit parallel computing [LA11]. The details of Tabu search and simulated annealing are out of the scope of the thesis, while that of genetic algorithms are described in the next section.

2.2 Genetic Algorithms

Genetic algorithms are a subset of evolutionary algorithms in evolutionary computing. This section introduces the evolutionary computing and genetic algorithms in five subsections.

2.2.1 Evolutionary Computing

In computer science, evolutionary computing is a sub-field of artificial intelligence, and is defined by the the type of algorithms which follow the principles first laid down by Charles Darwin of *"survival of the fittest"* [BFM97]. Emerging in the late 1950s, evolutionary algorithms are

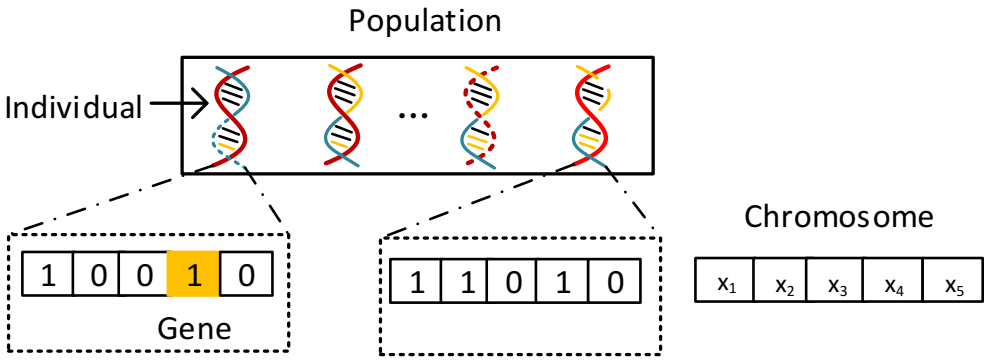


Figure 2.2: The Relationship between Gene, Chromosome, Individual, and Population. A population is a group of individuals, an individual is represented by the chromosome, and a chromosome is made up of genes

inspired by natural evolution and are technically a type of trial and error problem solver from the mathematics view [BFM97]. Typical evolutionary algorithms include genetic algorithms, genetic programming, ant colony optimisation, bee algorithm, and many other evolutionary methods. Although these evolutionary algorithms are applied in different problems, they use similar basic operations to produce new solutions from initial solutions. Genetic programming evolves programs as the genetic chromosomes, ant colony optimisation and bee algorithms learn from the actions of natural animals, but their details are out of the scope of this thesis. Genetic algorithms are general purpose, they evolve candidate solutions of a problem and can be applied to a large range of applications. This thesis focuses on the acceleration of genetic algorithms, but the proposed systems can be extended to support the other evolutionary algorithms since they follow the similar concepts and processes.

2.2.2 Overview of Genetic Algorithms

Introduced in 1960s [BFM97], genetic algorithms are increasingly popular search-based algorithms in evolutionary computing [CP00]. Genetic algorithms make use of three genetic operators: *selection*, *crossover* and *mutation*. Although GAs are randomised algorithms, they do not perform just a random search, instead they make great use of historical information in order to improve performance within the search space. The main terms related to genetic algorithms in the thesis are:

- Chromosome: a chromosome is the data structure $x = \{g_1, ..., g_n\}$ used to represent the properties of a candidate solution for the target problem, while the g_i is a gene in the data structure. The chromosome is often represented in real-valued or a binary encoding format (a vector of bits), and the evaluation function can re-interpret sub-sequences of the binary chromosome as Booleans, permutations or integer components.

- Gene: a gene is a piece of information in the chromosome, for example g_i in the chromosome $x = \{g_1, ..., g_n\}$. In an application, a gene normally represents a parameter in a solution.

- Individual: An individual is a candidate solution x to a problem. Individuals are linked with chromosomes in the genetic analogy, and are also called creatures or phenotypes.

- Population: a population means a number of individuals $\{x_0, ..., x_n\}$, which compete and evolve in one generation. Figure 2.2 shows the relation of gene, chromosome, individual and population.

- Initial population: the initial population it the population the evolution starts from.

- Fitness value: the value represents the quality of an individual x for the target problem, is usually calculated by the fitness function.

- Fitness function: the function evaluates the fitness value for the optimisation problem, which indicates the quality of an individual.

- Generation: a generation means the successive population in each iterative step in the evolution process.

- Evaluator: an operator generates fitness values for individuals.

- Genetic Operators: the operators guide the process of evolution. There exist three main genetic operators: selection, crossover and mutation. The operators are inspired by the natural evolution and evolve the individuals between generations.

  - Selection: in the population, selection operator chooses pairs of individuals as parents p_0, p_1 via a choosing method, which may be based on the fitness values, or a random method. The function form: $S : \{x\} \rightarrow \{p_0, p_1\}$.
  - Crossover: this operation takes two parents p_0, p_1, and creates two offspring individuals p'_0, p'_1 by recombining portions of both parents. Crossover rate controls the probability of a crossover ocurring. The function form: $C : (p_0, p_1) \rightarrow (p'_0, p'_1)$.
  - Mutation: Mutation introduces diversity by modifying the genes of an individual, such as inverting a bit, or changing a variable in a specific range. Mutation rate controls the probability when mutation operation happens. The function form: $M : p' \rightarrow p''$.

- GA parameters: GA parameters refer to a group of rates tuning GA behaviour, such as crossover rate and mutation rate controlling the probability of genetic operations and the random seed affecting the random number generation. Users need to tune the GA's parameters in the evolutionary process to achieve high performance [EHM99].

- Application parameters: The parameters in $f(x)$, for example, $f(x) = a + \{[(x^2 + x) \times cos(x)]/b\}$, a and b are the application parameters.

Genetic algorithms are used in solving many optimisation problems, including classical optimisation problems, such as set covering problems, travelling salesman problems, and maximal satisfaction problems. Recent research has shown that in searching a large state-space or an n-dimensional surface, this heuristic technique may offer significant benefits over other typical optimisation techniques such as iterative methods and greedy methods [HH04, SD07].

2.2.3 A Typical Genetic Algorithm

The code for the typical genetic algorithm is shown in Algorithm 1, and Figure 2.3 shows the data flow of it. As can be seen in the code, the GA starts from an initial population *oldpop*. After entering the while loop, all the individuals in the current population are evaluated by fitness

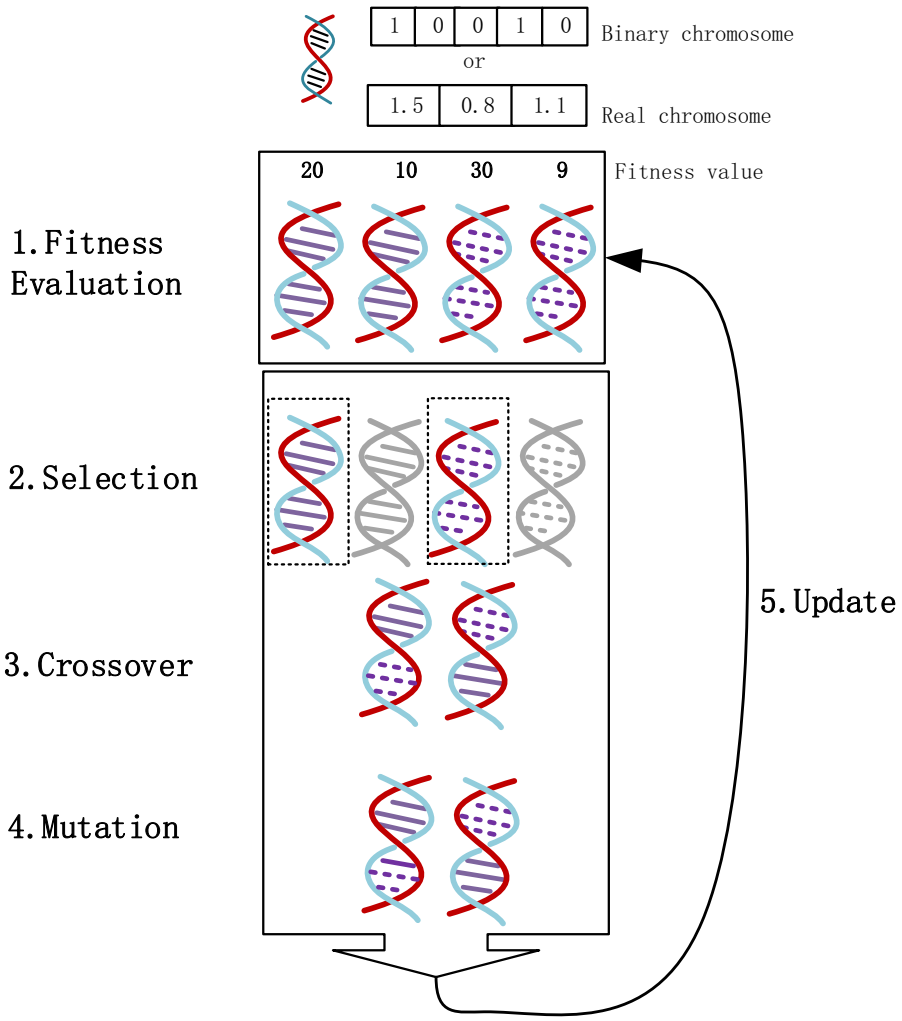


Figure 2.3: Data Flow of a Typical Genetic Algorithm.
Five stages: ①: evaluation, ②: selection, ③: crossover, ④: mutation, ⑤: update

Algorithm 1 A typical genetic algorithm

Require: *random_seed* {random number seed}
gen_max {maximum generation}
ni {number of individuals}
init[ ] {initial population}
oldpop[ ] {current population}
newpop[ ] {new population}
$fitness$[ ] {fitness array}
oldtwo[ ] { two selected individuals}
$findbest$( ) {to find the best solution from an array}
select( ) {to select two individuals from a population}
crossover( ) {to crossover two individuals}
mutate( ) {to perfrom mutation to an individual}
update( ) {to update individuals}

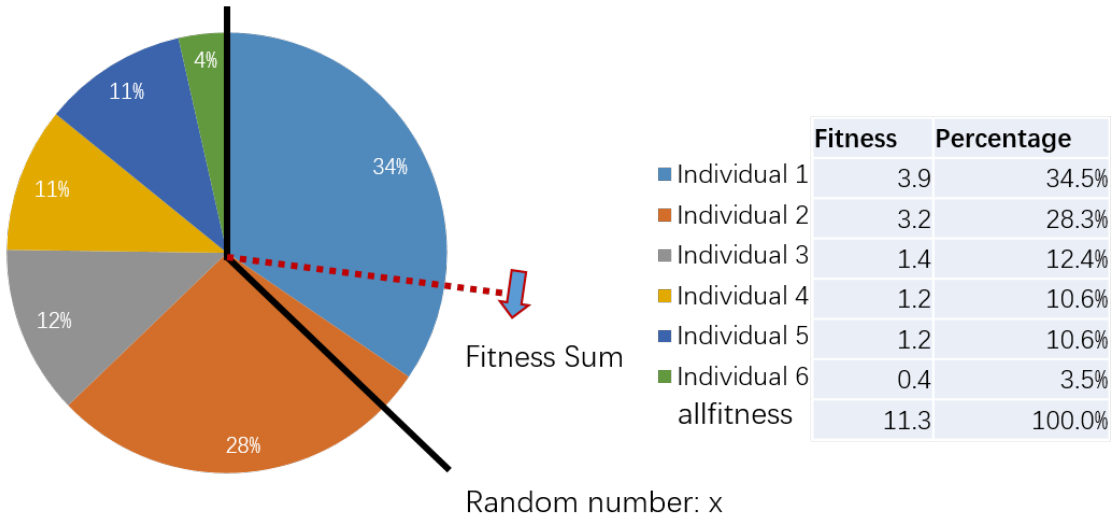
1: $gen\_n \leftarrow 0$
2: **while** $gen\_n \leq gen\_max$ **do**
3: **for** $i = 0$ **to** ni **do**
4: **if** $gen\_n \neq 0$ **then**
5: $oldpop[i] \leftarrow init[i]$ {initial or new population}
6: **end if**
7: $fitness[i] \leftarrow$ **evaluate**$(oldpop[i])$ {fitness evaluation}
8: **end for**
9: **for** $i = 0$ **to** $ni/2$ **do**
10: $oldtwo \leftarrow$ **select** $(oldpop, fitness)$ {select 'parents' and store them in oldtwo}
11: $newtwo \leftarrow$ **crossover**$(oldtwo)$ {exchange information}
12: **for** $j = 0$ **to** 2 **do**
13: $newtwo[j] \leftarrow$ **mutate**$(newtwo[j])$ {introduce diversity}
14: **end for**
15: $newpop \leftarrow$ **update**$(oldtwo, newtwo)$ {update the new individuals}
16: **end for**
17: $oldpop \leftarrow$ **update**$(oldpop, newpop)$ {update oldpop from newpop and oldpop}
18: gen_n ← gen_n + 1{enter next generation}
19: **end while**
20: $S \leftarrow$ **findbest** $(oldpop)$
21: **return** S {acceptable solution}

function *evaluate*() and the values are stored in the $fitness$ array (① in Figure 2.3). Then two individuals *oldtwo* to be included in the new generation *newpop* are selected probabilistically, which can be done in a number of different ways ((② in Figure 2.3)). For example, the probability for an individual to be selected can be proportional to its own fitness and meanwhile inversely proportional to the other individuals' fitness from the current population.

Once the individuals have been selected, additional members are generated using a crossover operation, which exchanges the information of selected individuals (③ in Figure 2.3). After that, random mutations are performed on them, thus creating new altered individuals *newtwo* (④ in Figure 2.3). Some GAs keep only the two best individuals from two parents and two offspring in the *newpop*, and throw away the others, while some GAs just clone the new offspring into the next population.

Once a new generation *newpop* is created it is then used to update the current population *oldpop* (⑤ in Figure 2.3). Some GAs compare *newpop* and *oldpop*, and keep a number of the best individuals in the next population. The new generation will go through the same process again, namely its individuals are evaluated, selected for the offspring and a new generation is being generated. The process will terminate when satisfying a termination condition, for example reaching a pre-set maximal generation (gen_max), or meeting pre-defined acceptances to the users.

When applying a genetic algorithm to a specific problem, a user should first design the structure of the chromosome, which represents the structure of a solution to the problems, as well as define the fitness function, which evaluates the quality of individuals. Except for these two problem-dependent parts (chromosome and evaluation function), the other parts in a typical GA such as genetic operators, are often problem-independent. Therefore, it is possible to build a general-purpose library, which can offer various implementations of genetic operators for different types of chromosomes.



	Fitness	Percentage
Individual 1	3.9	34.5%
Individual 2	3.2	28.3%
Individual 3	1.4	12.4%
Individual 4	1.2	10.6%
Individual 5	1.2	10.6%
Individual 6	0.4	3.5%
allfitness	11.3	100.0%

Figure 2.4: Roulette Wheel Selection.
x: a random number between 0 and $allfitness$. Fitness sum: the sum of the fitness from the first individual. When $sum > x$, the last individual accumulated is the selected one

2.2.4 Chromosomes and Genetic Operators

A chromosome is a data structure of a possible solution. There exist two main types of chromosomes, one is binary and the other is real valued [GPG10]. For example, Figure 2.3 shows a binary chromosome (1,0,0,1,0), and real-valued chromosome (1.5,0.8,1.1). The binary chromosome can represent integers, booleans, permutations and so on. For continuous optimisation problems, genetic algorithms are likely to use real-valued encoding. Although developers can use fixed point arithmetic to represent real values, it might be better to use a floating-point encoding, when the solution require specific precision.

For the different types of chromosomes, there are various methods of selection, crossover and mutation. Users can choose one selection method, crossover method and mutation method, or apply different methods to parts of a population.

For selection, the random method is very simple, randomly choosing individuals from the current population. Roulette wheel selection is a popular selection method [Gol89], as shown in Figure 2.4, which is based on the fitness values. First, all the fitness values in one generation are added together to get the *allfitness*, and a random number x between 0 and *allfitness* is

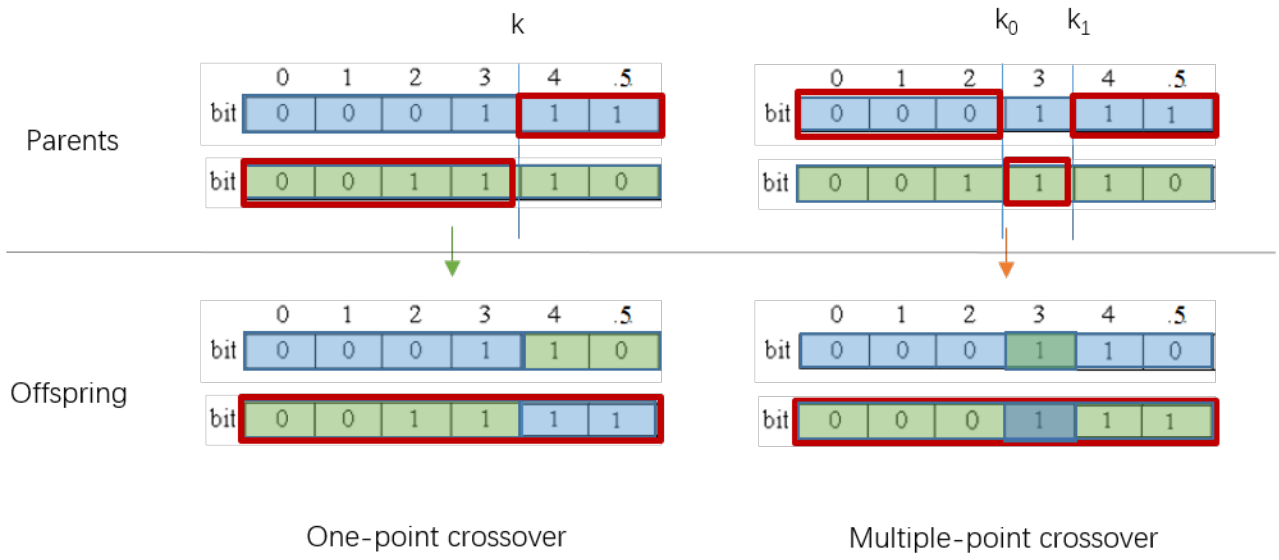


Figure 2.5: One-point and Multiple-point Crossover.
One-point crossover cuts a parent into two pieces, while multiple point crossover cuts parents into multiple pieces and recombines them

generated to pick an individual. Then the fitness values are accumulated again from the first individual until the *sum* is larger than the random number x. The last one accumulated is the parent selected. This method is also called fitness proportionate selection, because the individuals with larger fitness value are more likely to be picked. Another popular method is elitist selection or elitism, which directly chooses the best several individuals from the current generation and puts them in the next one [MEA05]. Elitism guarantees some of the best survive in the evolution process [TG94]. There exists also tournament selection, which chooses the best individual from randomly chosen subsets of a population [MG95].

Crossover exchanges the information of two parents, the most popular methods are one-point crossover, multiple-point crossover, and blending crossover, and they are suitable for different chromosomes. For example, one-point and multiple-point crossovers are commonly used in binary chromosomes. As shown in Figure 2.5, the one-point crossover first chooses a fixed point k in two binary parents $b_0...b_n$ and $b'_0...b'_n$, then splits the chromosomes from position k and recombines them as $b_0...b_k, b'_{k+1}...b'_n$ and $b'_0...b'_k, b_{k+1}...b_n$. Compared with one-point crossover, multiple-point method recombines the chromosomes by cutting them with multiple points. As shown in Figure 2.5, there exist two points k_0 and k_1, and a parent is cut into three sub chromosomes for recombination. For floating point chromosomes, blending crossover is used to

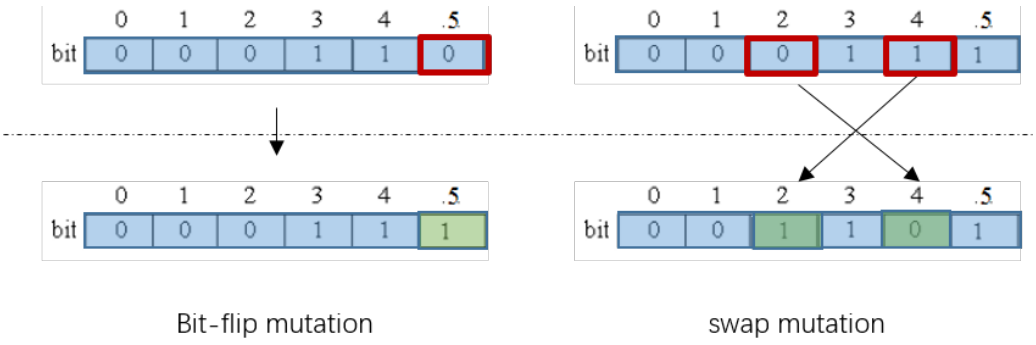


Figure 2.6: Bit-flip Mutation and Swap Mutation.
bit-flip mutation inverts a bit in the chromosome and swaps mutation exchanges the positions of two genes

generate a new value based on a linear mixture of two parents [HH04]. Any crossover method has to meet the constraints of the chromosome. For example, travelling salesman problems need an extra correction after one-point crossover because their chromosomes should not have duplicate cities [Nag06].

Mutation introduces diversity within a generation by changing the chromosome of one individual: the most popular methods are bit-flip mutation, boundary mutation, swap mutation, uniform, and Gaussian mutation. The bit-flip mutation is a simple method, inverting one bit in the binary chromosome. For example in Figure 2.6, a point k is randomly picked as mutation point, then the binary chromosome $x = b_0...b_n, b_i \in \{0, 1\}$ is modified to $x' = b_0...\neg b_k...b_n$. For example in Figure 2.6, k = 5 and b_5 is changed from 0 to 1. The boundary mutation generates one new chromosome between the lower and upper boundaries, which is applied to integer and real-valued chromosomes. The swap mutation is designed for permutation chromosomes, which swaps two permutations in the chromosome because there should be no duplicate permutations. As shown in Figure 2.6, b_2 and b_4 are swapped. Uniform and Gaussian mutations produce new genes by using uniform or Gaussian distributed random values.

Crossover and mutation operators both have rates to control the possibility of performing the genetic operation. For example, a high mutation rate is highly likely to introduce more diversity than a lower rate, and may help to escape from local optima. However, it also may result in a loss of good individuals. It is common to have a high crossover rate and low mutation rate for early generations, and then adjust them many times in the evolutionary process, which is called

adaptive GA [HH04]. This requires the execution platform to have the flexibility to support run-time tuning.

2.2.5 Variety of Genetic Algorithms

The previous subsection presents the *generational GA*, but there exist other types: the *steady-state* GA [Sys91] and parallel GAs (pGAs). There are other GA varieties, for example co-operative GA, which will be discussed in chapter 7. The *steady-state* GA is a simplified version of the generational GA. Holland originally discussed this variety of GA [Hol75], and De Jong evaluated the steady-state GA later [DeJ75]. In this GA version, a pair of parents are selected to produce offspring, which are then mutated and inserted back into the population. This process is then repeated in such a way that the population size remains constant. The steady-state GA requires less memory resources and has a reduced delay before entering the new population. However, this GA is likely to require more iterations to find a good solution than generic GAs because of small change between two iterations.

Genetic algorithms sometimes get stuck in local optimums, so to explore a wider search space as well as accelerate the searching process, parallel GAs are proposed to invoke multiple GA instances evolving a group of sub-populations in parallel by Bethke [Bet76]. When each GA instance carries the same parameters, the system is called uniform pGA or homogeneous pGA; When each carries different parameters, it is called a heterogeneous pGA or non-uniform pGA. The different parameters may bring more diversity in evolving processes between each GA instance [SD07], and thus non-uniform GAs may have a better performance [LA11]. For example, a non-uniform pGA evolves sub-populations under different selection, crossover and mutation schemes and different configurations, finds better results than uniform GAs for graph partitioning problems [LPG94].

A pGA can also be classified as single population GA or multiple population GA [CP98]. The simplest approach is the independent multiple-population pGA (iGA) [AT99], which executes multiple GA instances starting from multiple initial populations, as seen in Figure 2.7. A more popular and effective way is to establish extra communication between GA instances, so there

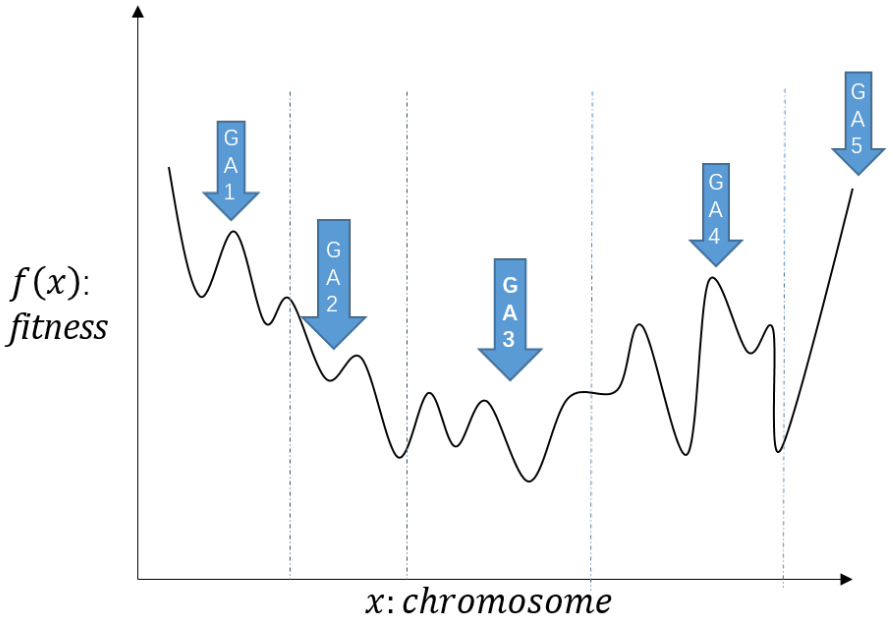


Figure 2.7: A Parallel GA Example.
Five GA instances starting from different search spaces

exist coarse grained multiple population pGAs and fine grained single population pGAs (see Figure 2.8) depending on the computation to communication ratio within the GA instances [AT99, CP98]. A representative coarse grained pGA is the distributed GA (dGA) or island GA with its own sub-population, in which the communication between GA instances is conducted by migration.

GA instances in fine grained pGAs communicate with each other more than in coarse grained pGAs. Master-slave pGA is a popular fine-grained pGA, containing a master GA instance and multiple evaluators valuing individuals in parallel. The master GA unit runs all other operations and sends the individuals to evaluators, while the slave evaluators return the fitness results to the master GA unit. The Master-slave GA is very useful when evaluation is the most time-consuming part during the GA process. For example, Erick used Master-Slave model to search the weights of the connections of the neural network, while the fitness function in the system took 3.68 seconds to evaluate every candidate [CP97]. Using Master-Slave model can reduce the total evaluation time.

Another type of fine grained pGA is called a cellular GA (cGA), when every GA instance maintains a small amount of individuals and communicates with its neighbours in a grid ar-

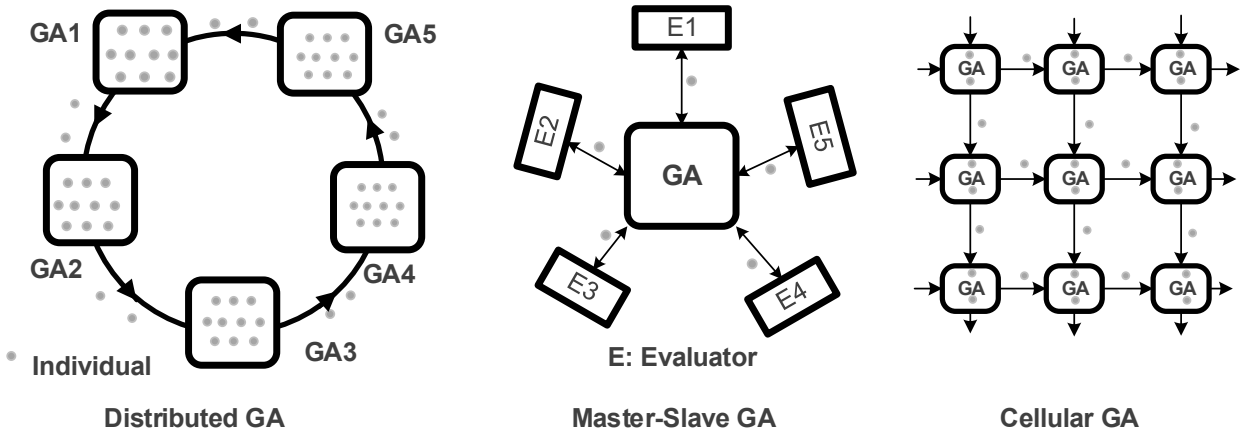


Figure 2.8: Distributed, Master-slave and Cellular pGAs.
Distributed GA has multiple GA instances evolving multiple sub-populations with migration; Master-Slave pGA has main GA module and parallel evaluators; Cellular pGA has multiple GA instances which frequently communicate with each other to evolve a single population

chitecture. As seen in Figure 2.8, every GA instance keeps sending its individuals out and also receiving incoming individuals from its neighbours GA instances. For example, Nebro, et.al. used cellular GA to solve 12 multiple-objective optimisation problems, and they found that the cGA achieved better results than two popular optimisation tools, namely NSGA-II and SPEA2 [NDL+09] .

All these pGAs have great potential for hardware acceleration due to their natural parallelism. The next subsection discusses all the potential hardware platforms, specifically FPGAs. Compared with the single-instance GA, pGA is not easy to implemented on software due to the synchronisation and communication between GA instances.

2.3 Reconfigurable Computing

This section first compares the hardware platforms such as Microprocessors, GPUs, ASICs and FPGAs, showing their advantages and disadvantages. Then, the section focuses on FPGAs, and describes the methods bringing high performance, the flexibility in FPGAs and challenges of current devices.

Table 2.1: Comparison of Microprocessors, ASICs, GPU, FPGA on Parallelism, Programmability, Flexibility, Clock Rates, Memory and Compilation Speeds

	Microprocessors	ASIC	GPU	FPGA
Parallelism	low (tens)		High(hundreds)	configurable
Programmability	high-level languages	HDL	CUDA, etc	HDL
Flexibility	high	low	high	high
Clock Rate	several GHz	MHz-GHz	several GHz	hundreds MHz
Memory	large DDR	-	large DDR	small BRAM + DDR
Compilation Speed	fast	slow	fast	slow

2.3.1 Overview of Hardware Platforms

Many applications require the computation platform to have high computing power and high performance, also request the platform to have high flexibility to tune GA parameters. Current platform solutions come from Microprocessors, Application-Specific Integrated Circuits (ASICs), Graphics Processing Unit (GPU) and FPGAs [CH02], and they all have advantages and drawbacks. A comparison of these hardware platforms is shown in Table 2.1, where we compare the parallelism, programmability, flexibility, clock rates, memory and compilation speed of them.

Application-specific integrated circuits (ASICs) are designed for a specific application, and normally produce high performance with a clock rate from MHz to GHz. However, ASICs cannot be changed after manufacture, and have a very high cost in design, development and verification. The programming of ASICs also needs a deep understanding of hardware and compilation is slow. The memory on the ASICs depends on the size of the applications.

Microprocessors provide high flexibility, and support high-level programming language such as C++, Java and so on. Although CPU is fast and the compilation is short, the program written in high-level language has to be transferred to the low-level instructions from a fixed instructions sets such as X86 or AMD64, and then the instructions follow fetch-decode-execute process, which may reduce performance. Multi-Core microprocessors accelerate an application by running processes in parallel, however it still results in an overhead in communication between cores and synchronisation between local memories.

Graphics Processing Units (GPUs) have many more parallel processing elements than multi-

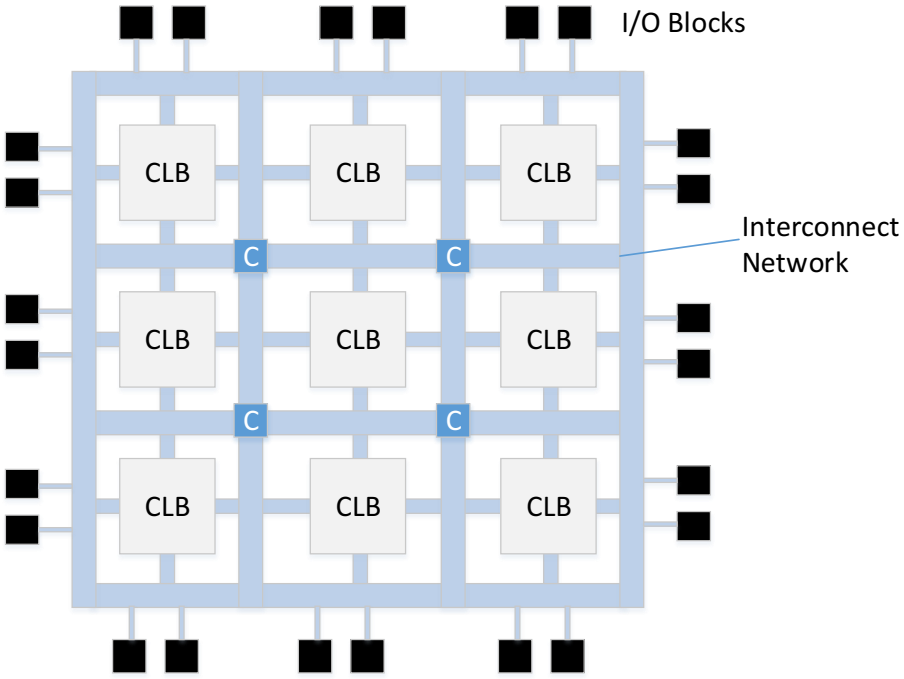


Figure 2.9: A Common FPGA Architecture.
An FPGA contains configurable logic blocks (CLB), connection blocks(C), and I/O blocks

core microprocessors, which processes the data in a very high degree of parallelism [TCW+05]. GPUs were originally designed to accelerate the image processing and computer graphics, but recently are applied to general problems, for example, Bolz, et. al. solved sparse matrix problem on the GPU [BFGS03]. However, the overhead of communication and synchronisation between processing elements is high [OHL+08]. For example, a pGA system with frequent migration is not suitable for this platform.

Field-Programmable Gate Arrays (FPGAs) combine the flexibility of microprocessors and efficiency of ASICs, as they can be configured as fast and flexible application-specific architecture [TCW+05]. The flexibility of an FPGA allows users to design the communication and synchronisation methods for the target application to reduce the overhead. Although FPGAs have limited resources and slower clock frequency compared with other platforms, FPGAs can use pipeline technique and parallel design to gain a high performance. The FPGA platform is the focus of the thesis, the details of it is described as follows.

FPGAs contain an array of logic blocks, a hierarchy of interconnections and Input/Output (I/O) blocks [CH02]. Figure 2.9 shows a typical architecture of an FPGA. A configurable logic

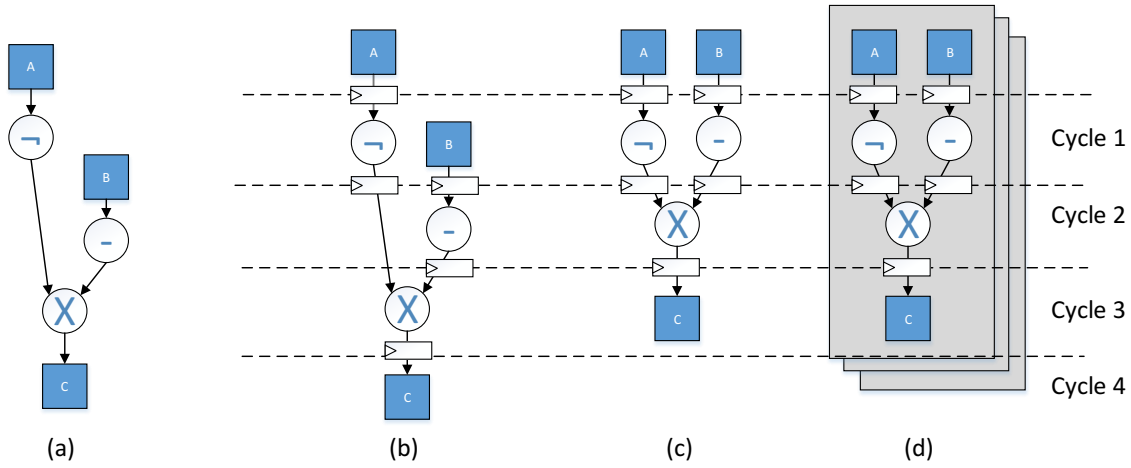


Figure 2.10: Pipelined and Parallelised Design Examples.
(a) equivalent control flow, (b) data-flow pipeline, (c) well behaved pipeline, (d) parallel well-behaved pipeline

block (CLB) or logic array block (LAB) can be programmed to be combinational functions or logic gates, the interconnection network is configured to wire these logic blocks, while I/O blocks connect the on-chip resources with the external devices. Modern FPGAs also have random-access memories (RAMs) to store data, and digital signal processor (DSP) blocks to accelerate the specific applications.

A configurable logic block consists of logical cells, each of which contains Look-Up Tables (LUTs), Flip-Flops (FFs), and multiplexers. The LUTs are used to create arbitrary combinational logic functions; the FFs are used as the clocked storage elements, while the multiplexers can route the logic inside or outside. The resources in FPGAs accelerate applications by using pipelined and parallel approaches, which are discussed in subsection 2.3.2.

2.3.2 High Performance: Dataflow, Pipeline and Parallelism

The data-flow programming model has become more popular in the hardware-based application designs [PA12]. The data-flow model splits the computational process into stages, which are placed in the available spaces. The data-flow computation process runs in a pipeline, and all these stages in the different places execute concurrently. Dataflow is also called stream processing or reactive programming.

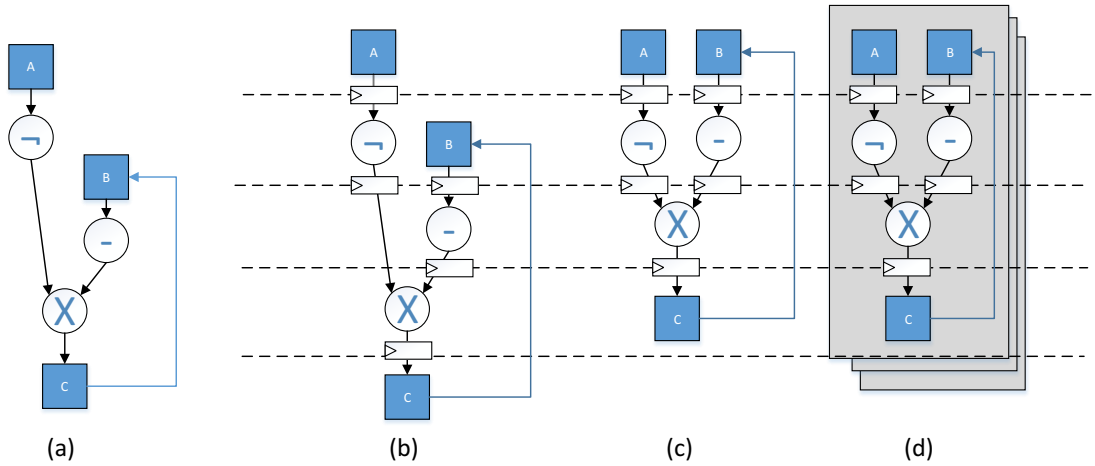


Figure 2.11: Pipelined and Parallelised Design with Feedback Examples.
(a) equivalent control flow, (b) pipeline with feedback, (c) well behaved pipeline with feedback, (d) parallel well behaved pipeline with feedback

By using the dataflow model, the program makes full use of hardware resources and space in the FPGAs. As seen in (b) of Figure 2.10, the pipelined data-flow model reads the inputs from different stages and generates the outputs every hardware cycle. If we place all the inputs at the same stage like (c) in Figure 2.10, the delay of the input will be removed, which is called well-behaved data-flow or fully pipelined [Sha92]. If it is a for-loop running for k times with a pipeline latency N, the control flow program takes $k \times N$ to finish the computation. If we use fully pipeline technique, it only takes $k + N - 1$, including the time for filling the pipeline. If the pipeline depth N is large, it makes a great difference in the execution time, and the ratio between $k \times N$ and $k + N - 1$ ends up as k times. For example, in Figure 2.10 with $N = 4$, we get the valid output every single cycle after the first 4 cycles of filling the pipeline. We also could add parallelism to pipeline if there is no dependence between input data, as shown as (d) in Figure 2.10, which will help increase performance. For example, if we increase the degree of parallelism to p, the time needed for k times computation will be $(k + N - 1)/p$.

However, a pipeline may stall due to a feedback, because the next-step execution has to wait for the updated data to reach the stage. For example in Figure 2.11, the data in the block B come from the result of block C. It takes 3 cycles for the new data from block C to reach the block B, that means before that the output data is invalid. In this case, we only get two valid results in 6 cycles, reducing performance per cycle. When designing a high-performance

pipeline for an application, users need to minimise the number of the pipeline stalls.

Pipeline structures can be replicated and placed in any available space in an FPGA. Unlike microprocessors or GPUs, FPGAs enable users to define the communication method and control the data flowing in the pipeline cycle by cycle, as well as maximise the user's ability to make full use of the parallelism. When the resources of an FPGA are not sufficient, they can be extended by connecting several FPGAs together [ZSHD02]. For example, [DKT+95] uses multiple FPGA architectures for real-time processing of low-level machine vision functions without requiring any additional memory. Even though most FPGAs have lower clock rates than that of microprocessors and GPUs, a well-designed pipeline and high parallelism help to beat them in terms of performance for many applications.

2.3.3 High Flexibility: Full and Partial Reconfiguration

FPGAs can be configured many times for different applications though the compilation process, which consists of synthesis, map, place and route. In the synthesis step, the register transfer level (RTL) program in a hardware description language (HDL) such as Verilog or VHDL, is compiled into a netlist of logic gates; then the netlist is mapped into logic blocks and interconnection. After that, these logic blocks and interconnection are placed in the available space of the target FPGA. Finally the routing process connects them together. The place-and-route process ensures to meet all the resource requirement and timing constraints. A user also needs to spend weeks or months to verify the functions of hardware design, which is very time-consuming. So compared with software systems, it is not practical to frequently change the hardware designs in an FPGA system.

The concept of reconfigurable computing dates to the 1960s, Estrin proposed a computer containing both a standard processor and reconfigurable hardware, which could be tailored to solve specific tasks [Est60]. The first modern era FPGA containing reconfigurable logic blocks appeared in 1984 by Xilinx [CDF+86]. After that, FPGAs platforms start to become popular in both academia and industry [KTR08].

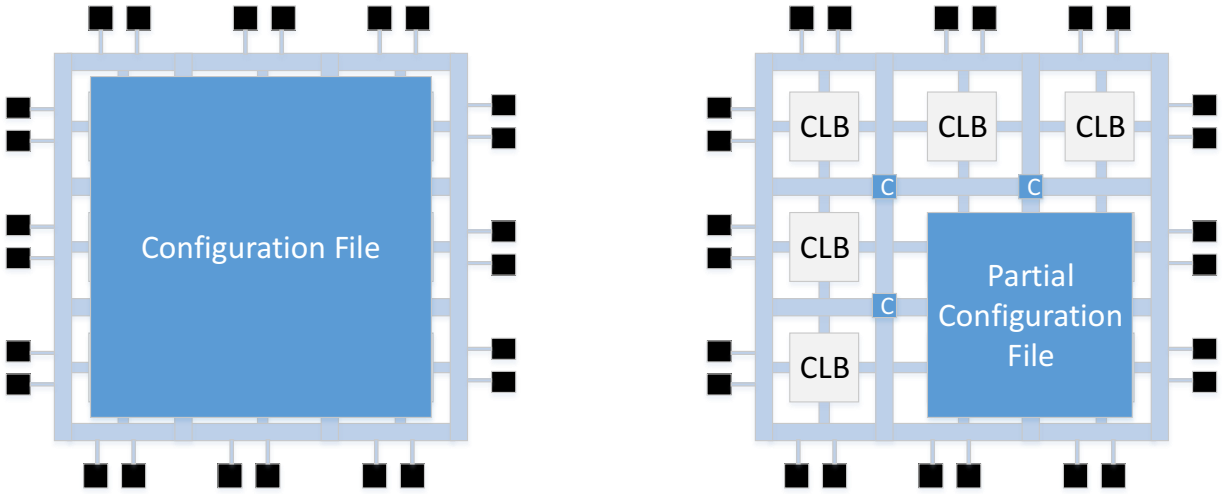


Figure 2.12: Full Reconfiguration (FR) and Partial Reconfiguration (PR). FR configures the whole FPGA, while PR configures part of the FPGA

Normally, an FPGA device is completely reconfigured, so the configuration file is loaded from an external device to overwrite the current configuration, this process is called Full Reconfiguration (FR). This means that for most FPGAs, a user has to save the states from previous design and reload the data for new design, even only a part of design is modified. Researchers have also proposed run-time partial reconfiguration (PR) to allow the users to configure a part of design [HH95]. Figure 2.12 shows the basic difference between full reconfiguration and partial reconfiguration. Normally, when using FR, the overhead is reconfiguration time and the corresponding memory needs to be reloaded after that. The partial reconfiguration only affects the PR region, shortens the reconfiguration time according to the size of PR region. It also saves reloading data time because the data can be kept in the memory [Kao05]. In addition, although partial reconfiguration uses additional on-chip resources for PR function, but it saves total storage space because it supports a system with different configurations, rather than having several separate systems. Some devices support dynamic partial reconfiguration, which allows the the rest of the device to keep running in the configuration process, while some devices only support static partial reconfiguration, making the rest of the device inactive when the device carries out partial reconfiguration [WBT06].

Many of the hardware acceleration solutions use full-reconfiguration (FR) to switch at run-time

between multiple separate FPGA designs. This FR approach saves on-chip area but it is slower in practice when a large amount of data needs to be processed and additional large memory needs to be used. However, PR is not easy to implement because users have to find a PR region to put the design in and the toolchain does not support PR very well.

2.3.4 Challenges and Development

An FPGA can gain high performance by designing good pipeline structure and increasing parallelism, as well as support high flexibility by using run-time full or partial reconfiguration. However, FPGA systems also have drawbacks and challenges:

- Low-level programmability. Programming FPGAs is difficult for non-expert users, as algorithms need to be written in a low-level hardware description language (HDL) such as VHDL or Verilog. HDL descriptions are based on registers and logic gates, which means a developer needs a very deep understanding of hardware. It also needs the ability from users to map an algorithm level program to the register-transfer level (RTL) hardware architecture.

- Time-consuming compilation and verification. Unlike software on microprocessors or GPUs, the compilation process in FPGAs contains synthesis, map, place and route, which are much more time-consuming and often take hours to finish. After modification, developers also need to verifiy the functions in new design, which takes additional time. As a result, it is not practical to frequently modify hardware designs on FPGAs.

- Limited resources. The memory resources in an FPGA are limited, especially the on-chip memory. Maximising the usage of the limited resources is a very big challenge of FPGA development.

In the recent years, researchers have proposed many solutions to meet these challenges [KTR08], which are as follows:

- High-level synthesis tools. High-level synthesis (HLS) tools are proposed to reduce the programming effort [CDW10]. These tools transfer high-level programming languages such as C, Java to low-level hardware description languages, and also significantly simplify the verification parts from users. For example, MaxCompiler [Tec13] transfers dataflow program in Max-Java language to the codes in HDL, and automatically verifies the hardware design against software for the users [Tec13].

  HLS tools have to consider the timing, address access and communication control in the translation. The steps they perform include code analysis, loop unrolling, resource allocation, resource binding, and scheduling [Bai15]. The coding style in HLS is a little different with general C/C++/Java code, in order to map the coding to high-performance dataflow. For example, Maxcompiler requires the bounds of for-loop should be known, in order to unroll the loop [Tec13], while Vivado High-Level Synthesis requires the C constructs must be of a fixed or bounded size [Des12]. HLS provides an easy way for developers to use the low-level registers and memory. However, it is still difficult for a non-expert user to create fast and efficient FPGA-based systems. These tools can provide a good intermediate-level target for customisable frameworks, but an easy tool to use is still needed to remove the programming barrier.

- Run-time parameters and run-time reconfiguration. Run-time parameters are changeable even when a design is still executing in an FPGA, which reduces the frequency of recompilation and verification. Especially for applications with different inputs, the run-time scheme enables only one design to support a series of applications. Run-time reconfiguration is another solution, especially the dynamic partial reconfiguration, it reduces the recompilation area and shorten the recompilation time.

- Large external memory and multiple FPGAs network. Large external memory provides several GBs such as DDR for resource limited FPGAs. Although the access speed is not as fast as on-chip memory, FPGAs can read and write a large amount of data via multiple I/O ports with a speed of hundreds of GB per second. If one FPGA is still not sufficient for an application, multiple FPGAs can be connected together to meet the

requirement. For example, a digital signal processor (DSP) for an electronic still camera was implemented on multiple FPGAs, and the system was faster than a ASIC system [WCT94].

Even with these solutions, an FPGA-based system needs to be carefully designed to maximise performance. This thesis considers all these challenges and designs flexible and fast systems for genetic algorithms.

2.4 Previous Hardware-based GAs

This section reviews the previous hardware-based GA systems and pGA systems. It then analyses the limitations of them: low-level programmability, lack of both run-time flexibility and architectural flexibility.

2.4.1 Hardware-based GA Systems

With the increasing demand for high performance of genetic algorithms and high flexibility to adjust parameters, researchers investigate to use hardware platforms such as microprocessors, FPGAs and GPUs, to accelerate the evolutionary process. Microprocessors provide a flexible platform for GA implementation. Goldberg proposed a Simple GA (SGA) system, which was used as a basic template by further researchers [Gol89]. There exist many GA tools based on SGA, including GALOPPS, GALib [Wal96], GA toolbox in Matlab, Genetic Algorithm Utility Library (GAUL). GALib is often used in comparison [AST09, BB04], which supports both simple GA and pGA with migration. We choose GALib as the reference too for performance in the thesis.

A GPU can be used to accelerate GA with multiple processing cores. For example, a generational GPU-based GA system combined with local search was proposed to solve the the maximum satisfiability (MAX-SAT) problem [MWMA09]. They obtained a 25 times speedup

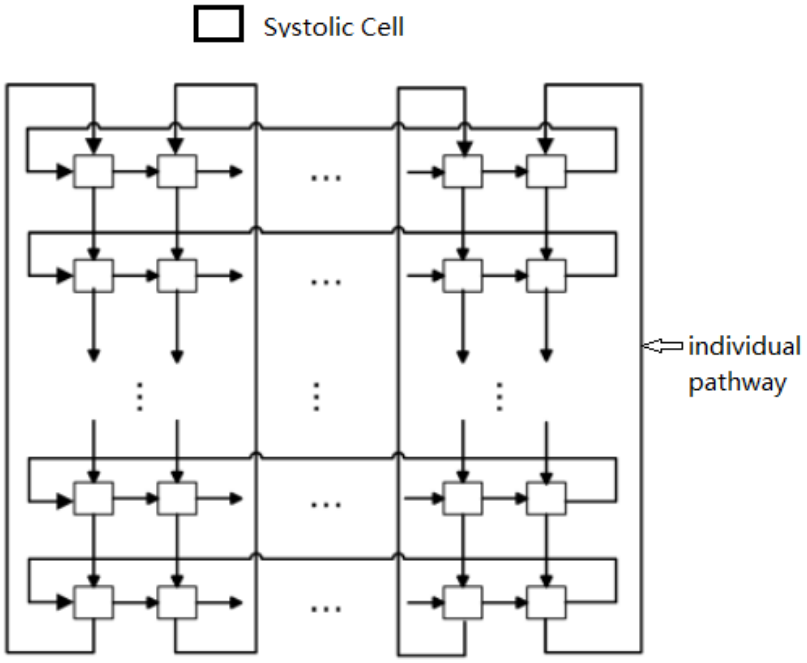


Figure 2.13: A GPU-based GA: Systolic Genetic Search Adapted from [PAL12]. The individuals move though a grid of systolic cells in fixed directions, while a systolic cell carries out a genetic operation

over some problems, but for others only had 6 times when compared with a GA system running on Intel Core i7 2.67 GHz with 4 cores. Pedemonte et al. proposed a non-generational GA system called systolic genetic search, which places the genetic operators into a fixed grid [AV11, PAL12]. As shown in Figure 2.13, the individuals in the system move through the network in a fixed direction, and are modified by genetic units equivalent to systolic cells. This GPU-based GA system was 5 to 35 times faster than software, depending on the scale of applications. The small applications were likely to have a small speedup but they did not identify the reason, I observe that it is probably because the communication and synchronisation cost between processing kernels contributes much in the execution time.

FPGAs are one of the most promising platforms to improve performance of genetic algorithms. Existing FPGA-based GA systems are effective in many applications [SSC+01, AC01, VPP09, FSK+08, SSS95, TY04]. There are two types of FPGA-based GA systems: an *application-specific* system tailored to one specific application with the fixed chromosome and specific genetic operators, while a *general-purpose* system supporting a wide range of chromosomes and genetic operators for different applications. For example, a GA system for Unmanned-Aerial-

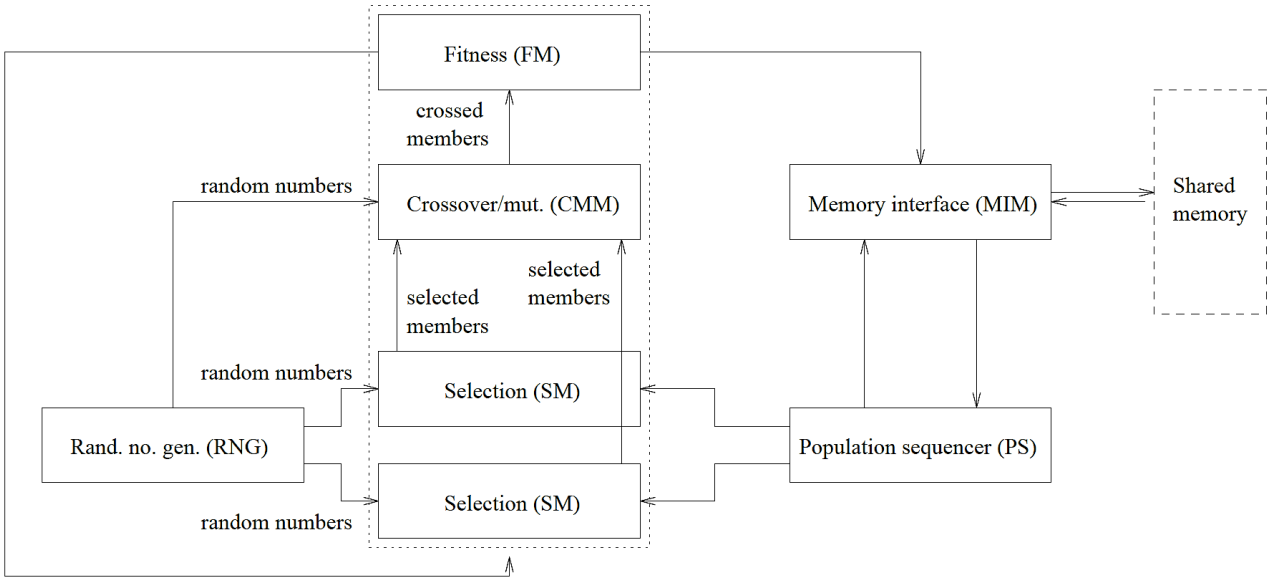


Figure 2.14: HGA System Proposed by Scott, et. al. Adapted from [SSS95]. GA process is controlled by the memory module

Vehicle Real-Time Path Planning was demonstrated with problem-specific settings [ATLF08]. General-purpose GA systems are the focus of this thesis, because they can be applied to general optimisation problems.

The first hardware-based Genetic Algorithm (HGA) was proposed in 1995 [SSS95]. They designed the system based on a typical software GA using the VHDL language and divided the software GA's process into several hardware modules. As shown in Figure 2.14, the HGA system reads the parameters such as population size and operator rates from a shared memory module via a memory interface. The process is also controlled by the status data stored in the memory module. However, these GA modules are not fully piplined, so they read new inputs only after they finish the operation for previous inputs. Parallelism is not well supported as only selection modules are parallel. They still reduced 96% of the execution time compared with a software based GA and attracted much attention from other researchers, who improved HGA with parallelism and pipeline technique [YY99, FZS10].

A general-purpose GA engine was demonstrated in [FZS10] and it can be embedded in other systems. The work allows users to change a limited number of run-time parameters, and supports multiple fitness functions for binary chromosomes. The overview of the system is shown in Figure 2.15, the system places the GA module and application module separate,

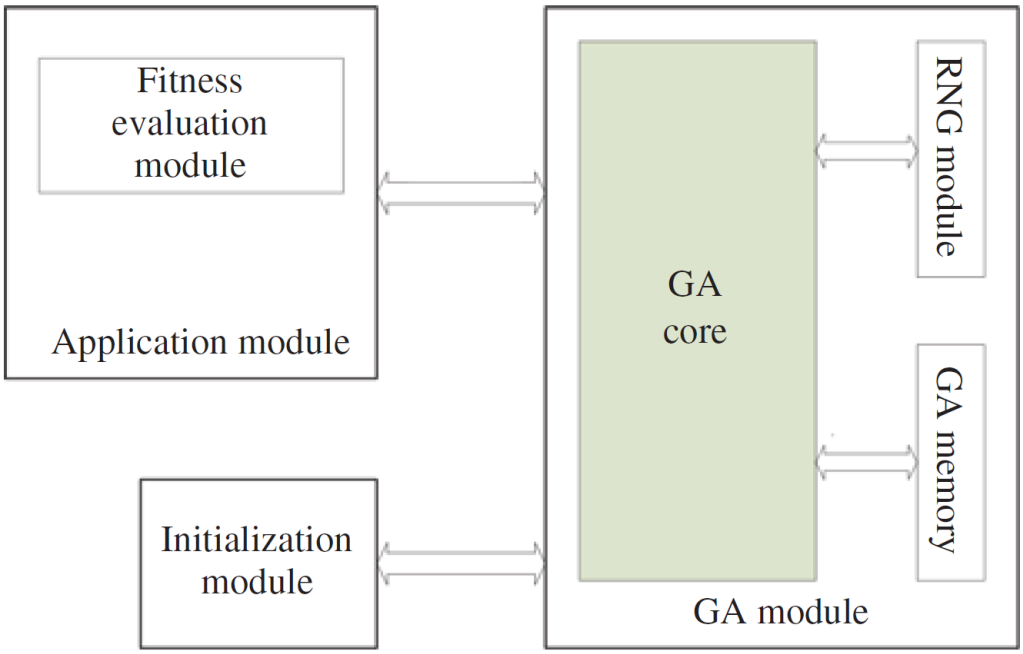


Figure 2.15: A Custom GA Engine Adapted from [FZS10].
The GA module and application module are separate, initialisation module is used to tailor the GA module

and uses an initialisation module to tailor the GA core. However, the tuning of parameters still needs hardware experience and speed-up is only 5.16 times compared to a microprocessor, probably because it uses a large piece of memory and less parallelism inside the GA core.

2.4.2 Hardware-based Parallel GA Systems

In addition to a considerable number of conventional FPGA-based GA systems, there exist FPGA-based GA varieties, especially the parallel GA systems. FPGA-based master-slave GAs and dGAs are demonstrated in [YY99, CC00, JKFA06, TMSY06, KK12], while FPGA-based cGAs are proposed in [JC08, SAF13]. Master-slave GAs [JKFA06] and dGAs are easier for FPGA place and route because of the simple and few connections between GA kernels, so there seem to be more instances of them in FPGAs.

The popular topology architectures in hardware-based parallel GA includes chain, ring, grid and island, as shown in Figure 2.16. For example, [YY99, CC00, KK12] used ring topology, [TMSY06] used chain topology, and [JKFA06, SAF13] applied grid topology. Some researchers try different topologies in one system, for example [JKFA06] uses both master-slave model and

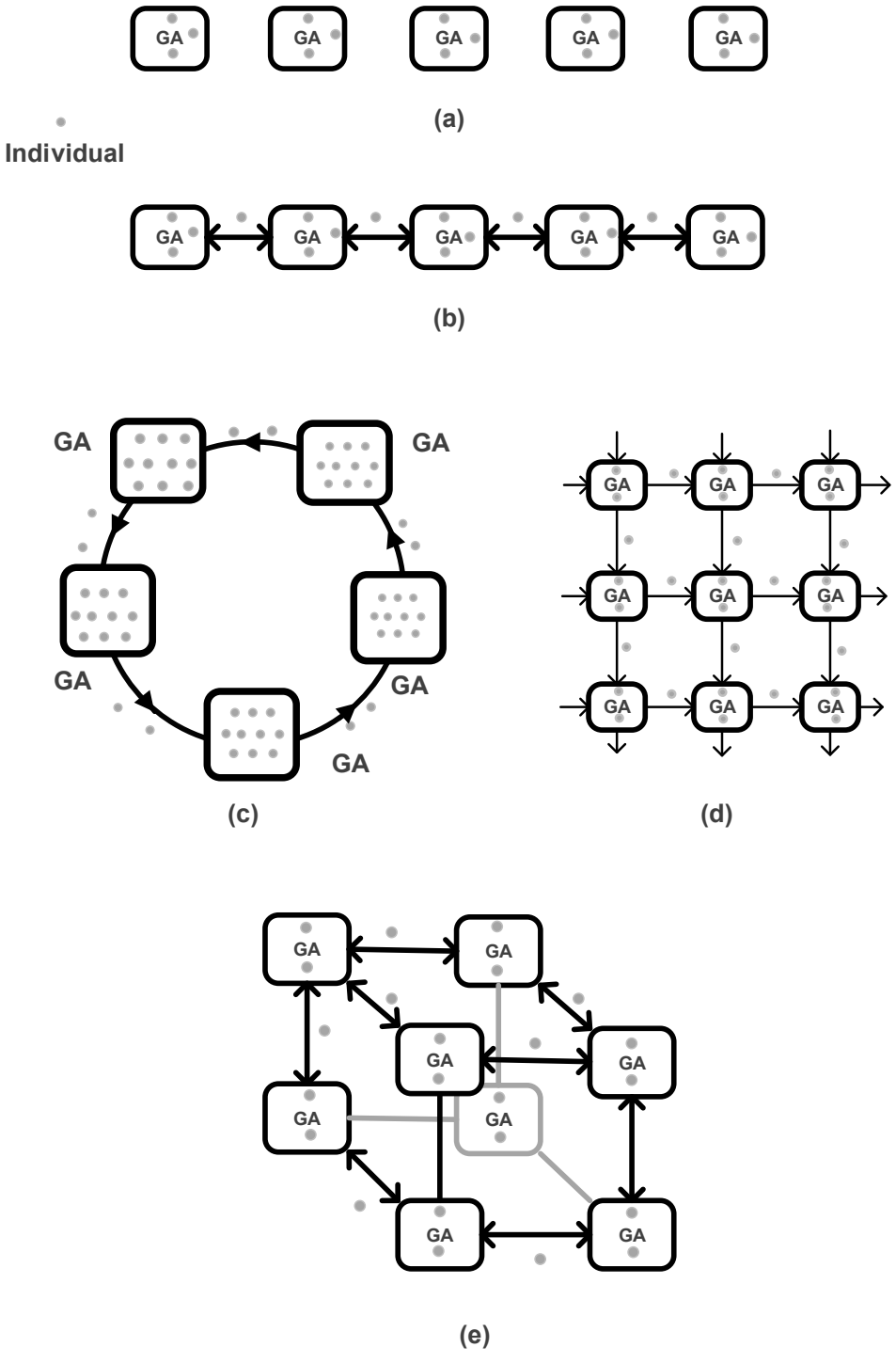


Figure 2.16: Popular Migration Topologies.
a) Island: no connection between parallel GAs; b) Chain: GA instances connect to its neighbours; c) Ring: Chain connection plus the head GA instance connecting the tail GA instance; d) Grid: GA instance connects with its four neighbours; e) Cube: GA instances connect with each other in a cube

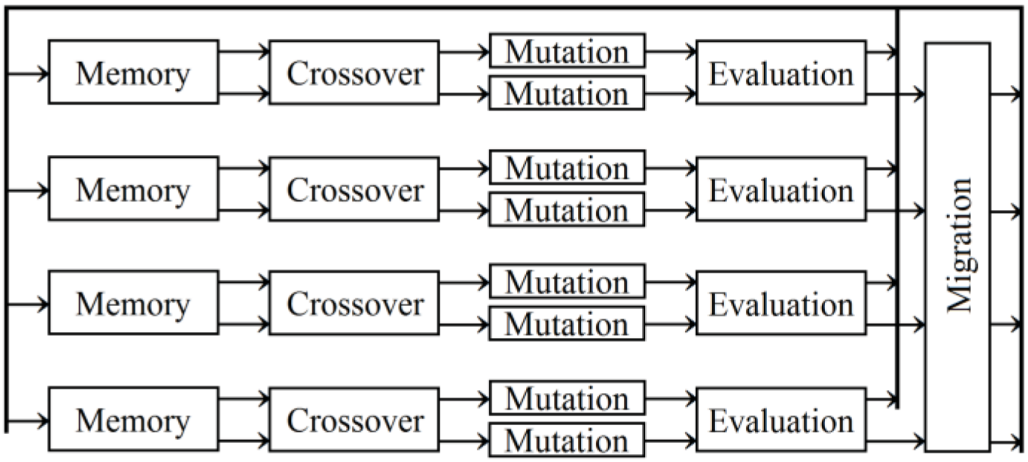


Figure 2.17: A dGA System [KK12].
Four GA instances exchange individuals via one shared migration unit

ring. The previous FPGA-based work do not use more complex topologies such as cube or 3 dimensional structure shown in Figure 2.16, probably because a complex topology is not easy to implement on FPGAs.

Kamimura and Kanasugi implemented a dGA processor in VHDL. As shown in Figure 2.17, the system consists of one migration unit and four GA instances [KK12]. Their system is applied to three numeric functions with binary chromosomes. However, the paper did not provide performance results, it only gave error rate results by simulation.

[SAF13] proposed a cGA hardware system with a scalable framework. As shown in Figure 2.18, the GA kernels contact with other kernels though the shared memory. Every processing element (PE) maintains a very small population with only four individuals and two of them are shared with other PEs. This work splits the large memory into small pieces, shows a speedup of 22 times over a CPU and 2200 times faster than a software NIOS processor in FPGAs. One issue with this method is the sub-population is so small thus it cannot carry out much selection.

Due to the insufficient resources of one single FPGA, the scale of the problems solved and the degree of parallelism are limited. To address the problem, researchers adapt pGAs to multiple FPGAs [NS05], [YY99]. [NS05] delivered a 2 times speedup from five chips over one chip, but did not provide an easy way to modify migration policy.

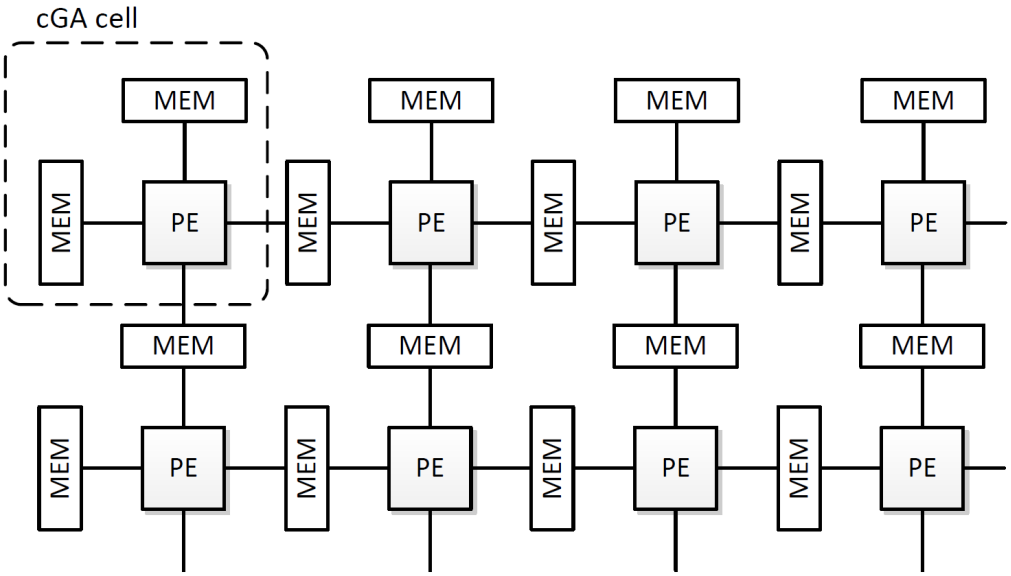


Figure 2.18: A cGA System [SAF13].
PE: processing element. MEM: shared memory. PE exchanges individuals via shared memory. A cGA cell contains two MEM and one PE.

2.4.3 Random Number Generator (RNG)

The Random Number Generator (RNG) plays an important role in many GA steps, including initial population generation, selection, crossover and mutation operations. Therefore, the quality of random numbers may affect the convergence of the algorithm. Some FPGA-based systems use pre-generated random numbers from CPU, while some use hardware-based random number generators [SSS95]. There are two types of RNGs in hardware: "True" random numbers generator (TRNG) which uses a non-deterministic source such as clock jitter in digital circuits; and Pseudo-random number generator (PRNG), which uses a deterministic algorithm to generate random numbers on hardware [THL09].

2.4.4 Limitation of Existing GA Systems

Although some GA systems provide high performance and some also support high flexibility, these existing FPGA-based systems suffer from one or more limitations:

1. Low-level programmability. The FPGA-based GA systems require a user to have significant hardware architecture knowledge before applying these systems to an application.

This limitation restricts the usage of FPGAs in the GA acceleration.

2. Lack of architectural flexibility. Most system are not general purpose, and support only one type of chromosome, which is not suitable for different applications; It is also hard to tune the structure to meet the requirement of resource usage, even for an expert, because most of the architecture is fixed;

3. Lack of run-time flexibility. Modifying either GA parameters or application parameters requires time-consuming recompilation and verification, which usually needs many hours to complete.

4. Low performance. The pipeline and parallelism techniques are not fully exploited in the systems, resulting in a low performance. Many modules in the systems are not fully pipelined or parallel at all.

Previous FPGA-based pGA systems also suffer from one or more additional limitations:

1. Inefficient communication. The links between multiple GA instances on one FPGA or multiple FPGAs are complex in design and synchronisation, reducing performance gained by parallel GAs;

2. Lack of run-time flexibility on migration. Changing migration policy needs time-consuming recompilation and verification. Because of that, researchers have not investigated the impact of migration settings on FPGAs, and so do not provide any setting guidance to users.

These limitations of current GA systems leave significant space for new research. When designing our system, this thesis considers all the limitations, and aims to build easy, fast and flexible GA systems.

2.5 Summary

Optimisation problems are common in many fields where they aim to find the best solutions or near-optimum from many candidates. In the existing algorithms to solve these problems, heuristics are good at finding good solutions in short time. As a popular heuristic algorithm, the genetic algorithm is inspired by natural evolution and uses selection, crossover and mutation to evolve candidate solutions. Genetic algorithms have been applied to a large number of applications. For complex problems, GAs may need much more time to produce the results and GA parameters may also need to be tailored frequently. To meet the requirements for performance and flexibility, hardware platforms such as microprocessors, GPUs and FPGAs are used as the execution platforms for genetic algorithms.

FPGAs are hardware devices which combine the high performance of ASICs and the flexibility of microprocessors. The platform gains high performance from parallel and pipeline techniques, while it also has the ability to reconfigure the resources fully or partially at run-time. Although an FPGA has limited resources, a group of FPGAs can be connected to support large scale applications. Thus, FPGAs become a promising platform for the acceleration of GAs. This thesis uses FPGAs as the main platform to accelerate GAs, and provides an easy, fast and flexible tool.

There exist many FPGA-based GA systems but they all suffer from one or more problems: 1) low-level programmability, because applying these GA systems to an application needs a deep understanding of hardware and users need to write GA using low-level hardware description; 2) lack of architectural flexibility, because most units of the previous GA systems are fixed thus limiting the tune for resource constrains; 3) lack of run-time flexibility, because any modification will result in long-time recompilation and verification. They always take hours or days to finish, making it impossible to frequently tune GA parameters in these GA systems; 4) low performance, because pipeline and parallelism techniques are not fully exploited in these GA systems.

To address these problems of current FPGA-based GA systems, the next chapter proposes an

automatic framework to generate and execute an FPGA-based GA system. The framework reads high-level input from users, achieves high performance for GA benchmarks and supports both architectural and run-time flexibility.

Chapter 3

Framework for the Generic GA

As shown in the background and related work in chapter 2, genetic algorithms are very useful optimisation algorithms and can be applied to both continuous and combinatorial optimisation problems. Genetic algorithms require sufficient computational power to evolve a group of individuals, and a high flexibility to tune GA parameters. Software GA cannot provide enough performance, and genetic algorithms have a great potential to be fast on FPGAs, because evolutionary process in an FPGA can be parallelised and pipelined. In comparison to ASICs, FPGAs also have a high flexibility to support GAs' tuning. However, the existing FPGA-based GA systems do not make full use of FPGA technology in terms of reconfiguration and high performance, nor well support run-time GA tuning. This thesis designs an automated framework under the following goals:

- High performance: genetic algorithms require sufficient performance to solve complex problems with many local optima, and it may need many generations to produce good solutions. High performance is the first requirement for a GA system.
- Run-time flexibility: genetic algorithms are non-deterministic and have many parameters. Modification of parameters should not be time-consuming in order to find a good configuration in a short time.
- Architectural flexibility: the resources in one FPGA are limited, thus customisation of

the architecture is necessary in order to fit problems with different scales and make full use of the hardware resources.

- High-level programmability: The FPGA is a platform demanding deep hardware knowledge, which is a barrier for non-expert users. High-level input for generating hardware is required to non-expert users.

Most existing systems try to meet the requirement of high performance, while our proposed work also supports other requirements. Unlike most previous GA systems, a user without hardware design experience can easily create an FPGA-based GA system for an application using a high level description, which brings high programmability to the GA system. After reading the user input, the proposed framework produces a pipelined Generic GA system executing on hardware, providing high performance while retaining architectural flexibility. Meanwhile, both GA and application parameters are run-time changeable without time-consuming recompilation and verification, which can take hours or days to finish in most of the previous GA systems. This chapter describes the proposed GA system as follows:

- Section 3.1 provides the overview of the automated framework, including design flow and execution flow, genetic operator library, and reports generated by framework.

- Section 3.2 describes the architecture of the proposed Generic GA in details, including all internal units.

- Section 3.3 describes the hardware pipeline and internal parallelism of the proposed Generic GA.

- Section 3.4 demonstrates the high-level user-defined inputs, showing that the framework is an easy tool for the non-expert users.

- Section 3.5 describes several classical optimisation problems, and shows how to describe them in a high-level language.

- Section 3.6 demonstrates the qualitative comparisons between the proposed system and the previous FPGA-based work, showing our system is more flexible in both architecture and run-time tuning.

- Section 3.7 demonstrates the quantitative comparisons between the proposed work and the previous FPGA-based work, showing our work covers both high performance and high flexibility.

3.1 Automated Framework

This section describes the underlying principals of the framework, which contains problem-independent library, hardware and software generators. It also describes the design flow and execution flow, as well as the compilation and execution reports from the framework. The framework provides a general purpose platform for genetic algorithms on the FPGAs, as well as removes the barrier to hardware for non-expert users.

3.1.1 Design Flow and Execution Flow

Figure 3.1 provides an overview of hardware/software generation process in the proposed automated framework, showing compile-time inputs with hardware on the left, and run-time inputs with software on the right. The compile-time inputs control the scale of architecture in the hardware, while the run-time inputs carry the application parameters and GA parameters. The GA parameters include crossover rate, mutation rate, random number seed and so on. Application parameters are the input for the applications, which are defined in section 2.2.2. For example, in function $f(x) = a \cdot x + b$, a and b are the application parameters.

The *framework* requires no hardware programming by users, who only need to provide the high-level application and GA inputs. The hardware and software generators, written in python, automatically combine the inputs and a number of existing templates from the library into the following:

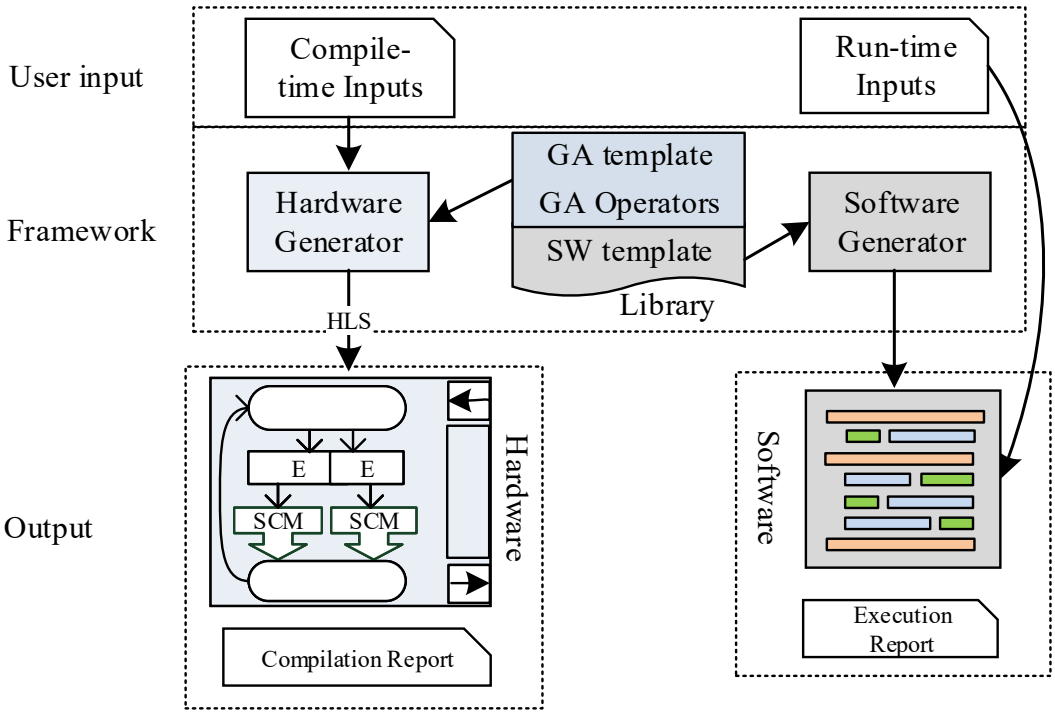


Figure 3.1: Hardware/Software Generation Process in the Framework. Framework outputs hardware, host software and reports based on the compile-time user-inputs and library; HLS: high-level synthesis; Library: a library containing genetic operators, pre-defined hardware GA template and software (SW) template; Run-time inputs: parameters changeable at run-time; Compile-time inputs: the GA architectural and application settings; Compilation Report: hardware implementation result; Execution Report: the record of execution

- Hardware files with embedded compile-time parameters;
- Compilation report with results of the hardware implementation;
- A software file which sends the run-time parameters to hardware and manages GA execution.
- Execution report, which is used to record the execution results under different configuration in run-time inputs.

In the hardware generation process, a high-level synthesis (HLS) tool such as MaxCompiler compiles the hardware code to a low-level implementation [Tec13]. In the software generation process, the generator configures in the host software ready for run-time inputs.

In the execution flow, Figure 3.2 and Figure 3.3 shows the two main stages, namely initialisation

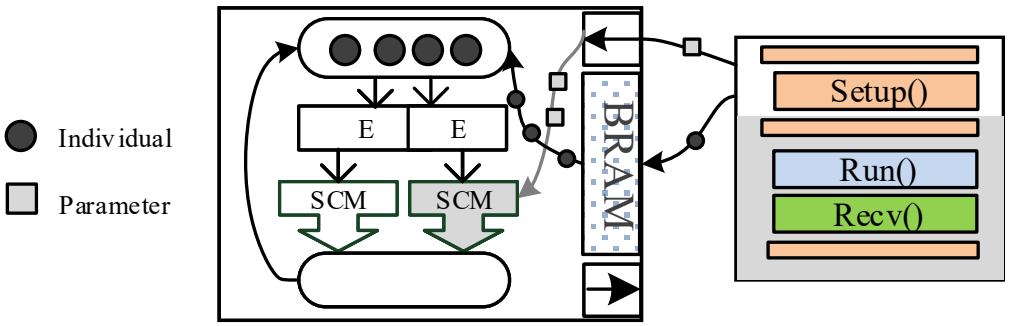


Figure 3.2: Stage 1: Initialisation.
Host Software (*setup*) sends initial population containing individuals and application parameters to FPGA via I/O ports and on-chip memory

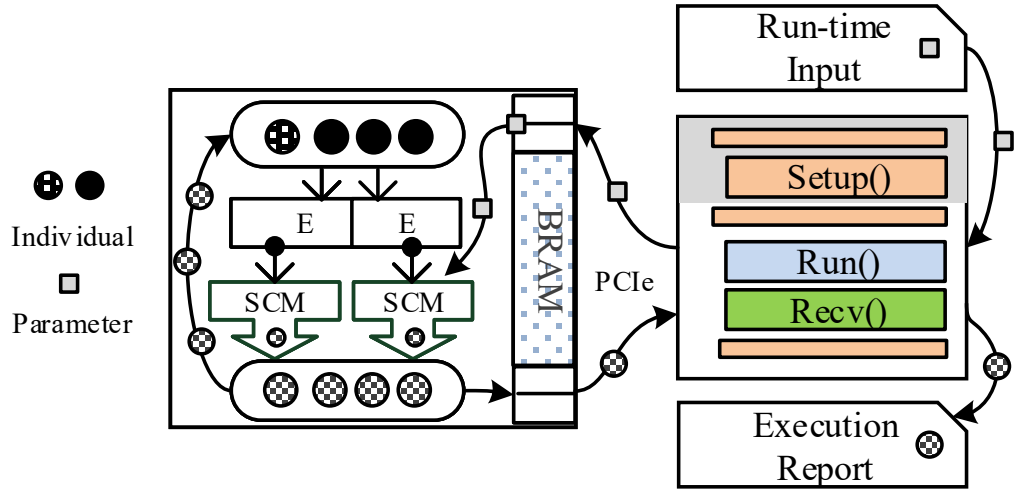


Figure 3.3: Stage 2: Evolution.
GA evolves the candidate solutions generation by generation, and sends results and chromosomes to the host software. The host software controls the execution of the system by sending commends or run-time parameters and records the results in execution report (*Run* and *Recv*)

and evolution. In the initialisation stage, the software transfers GA and application parameters to the FPGA using *setup*(), via CPU-to-FPGA input streams and on-chip memories though PCIe link. For example, the framework can send the initial population pre-generated to BRAM and application parameters to FPGA. In the evolution stage when executing the system, the FPGA-based GA receives run-time inputs such as crossover rate, and evolves candidate solutions generation by generation and outputs results to the host software. The software controls the hardware FPGA by sending commends and run-time parameters, as well as records the results in the execution report. In this way the requirement of high-level programmability identified earlier in the chapter is met.

3.1.2 Problem-Independent Library

As mentioned in section 2.2.4, the genetic operators are essentially problem independent, so it is possible to implement them in advance, before knowing the specific application a user will use them in. The library in the framework consists of different genetic operators, a parametrised GA template, and host software. Various methods of performing genetic operations are available for use in different circumstances, selectable by users. For selection, random, elitism, roulette wheel and tournament methods are offered.

- The 'random' method randomly selects individuals from one population.
- 'elitism' selects a number of best individuals from the current population.
- 'roulette wheel' selects based on the fitness values, and is likely to choose the individuals with high fitness.
- 'tournament' method chooses the best from several groups of individuals.

The details of these methods are presented in section 2.2.4. For mutation, the framework supports:

- 'bit-flip' randomly inverts bits within a chromosome, which is suitable for binary chromosomes.
- 'swap' exchanges items within a permutation chromosome, which is suitable for the chromosomes of sequential ordering problems
- 'constraint' generates random numbers within pre-set bounds, which is suitable for real-valued problems.

For the crossover operation, the framework supports:

- 'one-point' recombines two parents from a specific point.

- 'multi-point' exchanges genes of two parents at multiple points.
- 'blending' generates a new value based on a linear mixture of two parents [HH04], which is suitable for real-valued chromosome.
- 'reordering' rearranges the items in a permutations chromosome.

By using the library, the framework offers support for both combinatorial and numerical optimisation.

The hardware implementation of genetic operators significantly differs from software implementation, as it requires deep understanding of the hardware by the developer; subsection 3.3.2 discusses these differences in detail. The library removes the barrier for the non-expert users, because it includes sufficient designs for different applications. The users can choose the specific operators from the library, and the hardware generator embeds the corresponding optimised hardware into the output.

3.1.3 Compilation and Execution Reports

During the compilation and execution stages, our framework automatically produces two reports, including:

The *compilation report* presents to users the results of the hardware implementation, such as overall resource usage, clock frequency and any errors. Errors may be related to the syntax of the input specification, or due to resource exhaustion. Based on this report, the user can decide whether to change the number of the GA internal units, depending on whether there are free resources.

During execution, the FPGA platform uses an FPGA-to-CPU stream to output the best solutions together with their fitness found, making the current best solution immediately available to the end user. The *execution report* shows how different configurations and parameters affect performance. This report also contains recommendations suggesting the best configuration for

future executions of the problems with different application parameters, thus being useful when solving a series of problems.

3.2 Architecture of Generic GA

This thesis designs a scalable hardware architecture for FPGA-based GA systems, called a "Generic GA". Unlike most previous hardware-based GAs, this Generic GA has high architectural flexibility and supports run-time flexibility. The units of the Generic GA are customisable, and the library in the framework provides specific genetic operators used to customise it. Each Generic GA contains all the hardware and configuration settings needed to solve instances specific to a user's problem. Once compiled, the Generic GA can be repeatedly executed with different application parameters without time-consuming recompilation and verification, and GA parameters, such as crossover rate, mutation rate and random seed, can be varied during execution. In most previous FPGA-based systems, modifying these parameters needs hardware recompilation and verification, which takes hours or even days to finish.

While the units of other GA systems are fixed, this Generic GA is functionally flexible and scalable, allowing the user to easily customise the architecture for hardware resource constraints. In the Generic GA, the steps of a generational GA are converted into hardware functional units, which are pipelined and parallelised. The Generic GA also can be extended, such as adding migration unit designed in the next chapter to communicate with other Generic GA instances.

The design map of a Generic GA from a generational GA is shown in Figure 3.4. The Generic GA is labelled by the parameters listed in Table 3.1. The following subsections describe all the units of a Generic GA in detail, including:

- Chromosome configuration (Subsection 3.2.1). A chromosome represents the structure of a candidate solution for the target problem, the configuration is a must before using a GA system.

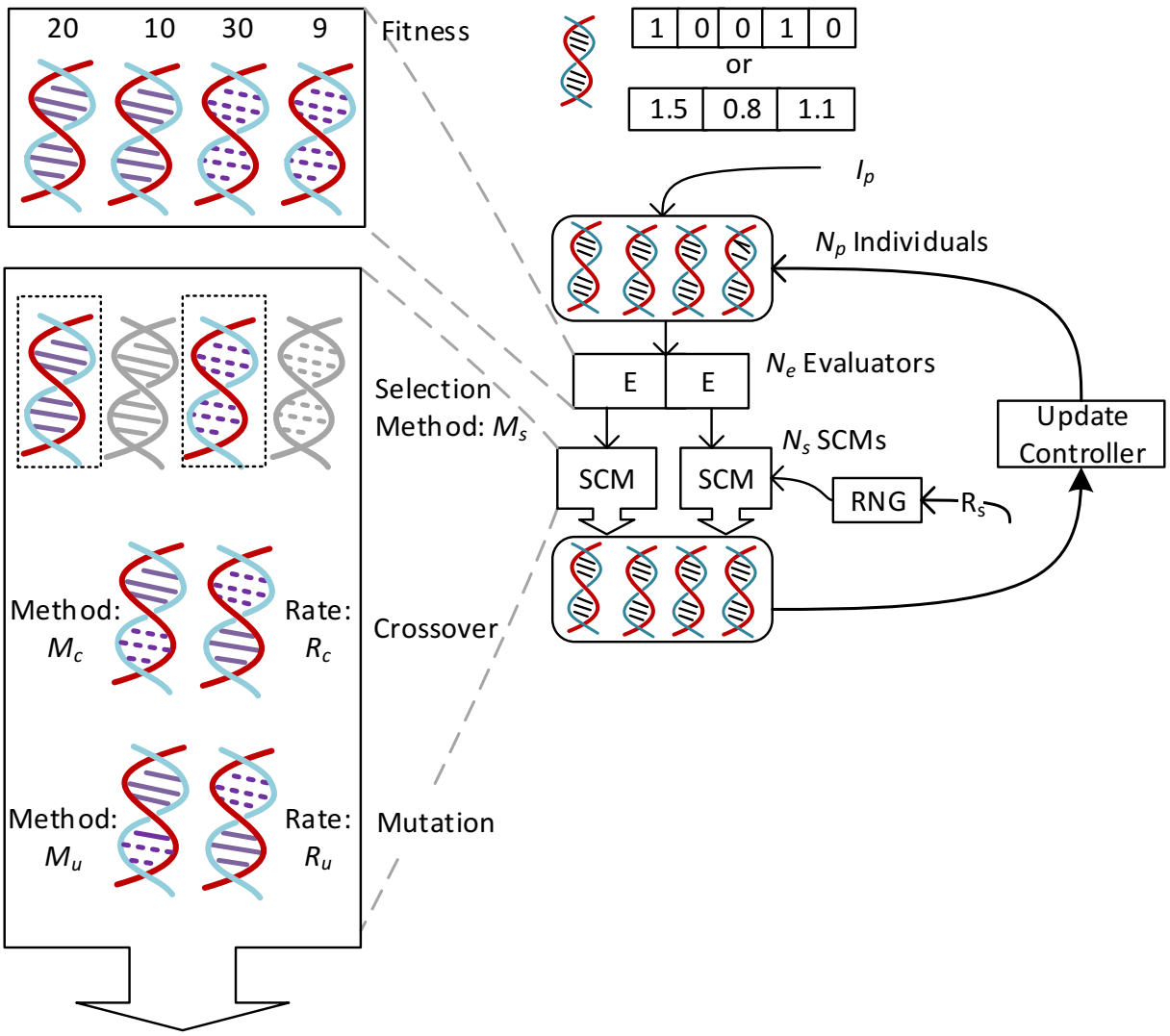


Figure 3.4: Hardware Architecture of the Generic GA Mapped from Figure 2.3. SCM: a unit combining selection, crossover and mutation; E: evaluator; all parameters are listed in Table 3.1. RNG: random number generator

- Population initialisation (Subsection 3.2.2). The initial population represents the start of the evolution.

- Fitness evaluator (Subsection 3.2.3). The evaluator values an individual and returns a fitness value.

- SCM unit (Subsection 3.2.4). A SCM unit combines selection, crossover and mutation together as one unit.

- Update controller (Subsection 3.2.5). The update controller decides the updating of the population.

Table 3.1: User Modifiable Parameters

	Compile-time Architectural Parameters		Run-time GA Parameters
N_e	number of evaluators	N_g	maximal generation
N_s	number of SCM units	R_u	mutation rate
M_c	crossover method: "one-point","multi-point", "blending","permutation"	R_c	crossover rate
		R_s	random seed
M_s	selection method: "roulette wheel", "random" "elitism","tournament"	I_p	initial population
		N_p	population size
M_u	mutation method: "bit-flip","constraint","swap"		

- Random number generator (Subsection 3.2.6). The generator produces the random numbers for the evolutionary process.

3.2.1 Chromosome Configuration

To create a Generic GA for a new application, a user needs to define the chromosome for an individual in the population, which can be constructed in many different forms for various applications, such as: integers, floating-point representations, permutations, or bit-strings, as mentioned in section 2.2.4.

Unlike most previous FPGA-based systems which support only one type of chromosome, our Generic GA system supports different kinds of chromosomes. The chromosome can be configured as a set of Booleans, permutations, integers, real components or a combination. Although someone could use fixed point numbers to represent a real valued, it might be better to use a floating-point encoding for specific precision. Our Generic GA provides different genetic operators for the selected chromosome type.

3.2.2 Population Initialisation

The initial population represents the start points of the search space in a genetic algorithm, and may need many generations to produce high fitness individuals from low fitness individuals. In our Generic GA, there are two approaches to generate the initial population:

- via randomisation, where the initial fitness depends on the random seed;
- by loading solutions from previous heuristics on a CPU or other platforms, which normally leads to a higher starting fitness.

The initial populations are loaded into the on-chip RAM set up from software before the Generic GA execution begins. The Generic GA reads the initial population from the pre-defined area. By loading the initial population from outside, the GA system not only reads the good solutions generated by other systems, it can also continue the work from a previous snapshot if the system is restarted. This method makes the system easy to connect to other platforms and continues their work any time. For example, if the FPGA is connected to the other platform such as CPU, GPU or even another FPGA, the result of the platform can be cloned in the Generic GA.

3.2.3 Fitness Evaluator

The evaluator unit returns a fitness value for an individual, which guides the exploring processes. In the Generic GA, there are parallel evaluators to reduce execution time, based on the concept of Master-Slave parallel GA described in section 2.2.5. To simplify the hardware design effort, the fitness function is written in a high-level description language according to simple rules. The rules are related to the pipeline and parallel architecture, described in section 3.4. Some working examples are also shown in section 3.5 for different functions.

3.2.4 Selection, Crossover and Mutation (SCM)

The selection, crossover and mutation units used in our generic GA system, are linked together into an SCM unit. This linkage is efficient due to their close data coupling, thus making it easy to instantiate replicas for spatial parallelism. Our automated platform provides an extensive library which contains different methods for performing the selection, crossover and mutation. This library allows the user to customise the genetic operators for a specific application, as shown in section 3.1.2. As shown in Table 3.1, a user can choose a selection method from the

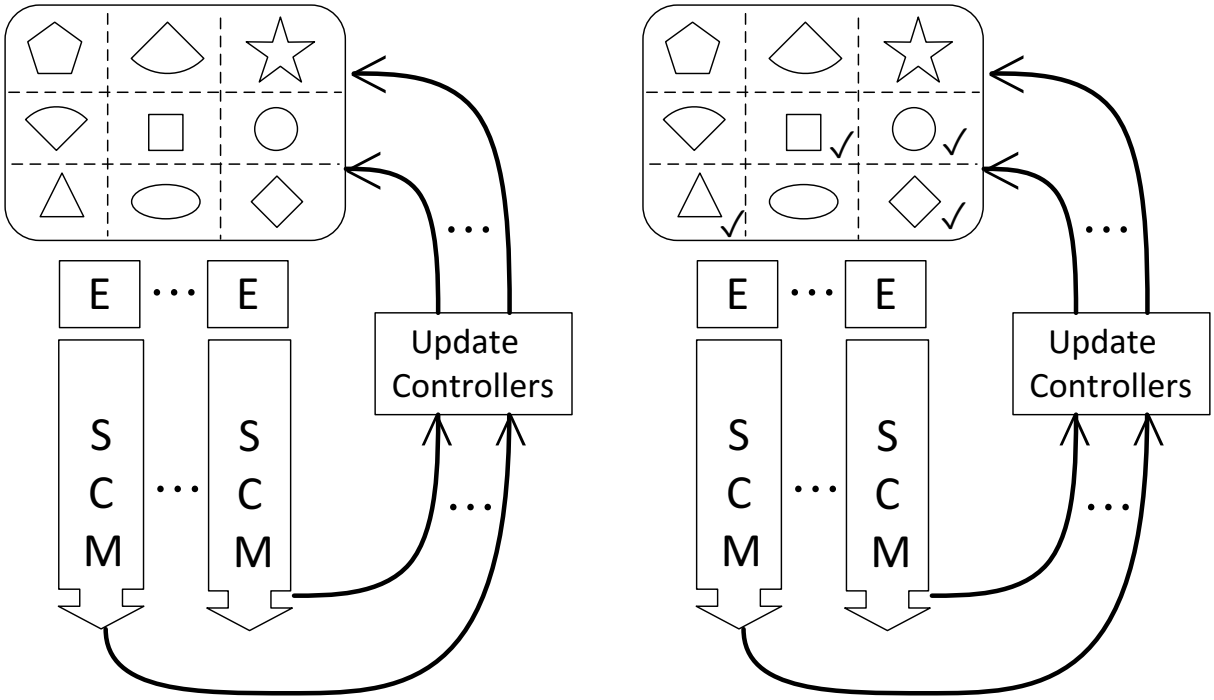
available methods in the library, such as elitism, random, roulette wheel or tournament selection. For *binary chromosomes*, one-point or multi-point crossovers are available for combining different parts of parents. The user can then choose binary mutation which inverts bits inside the chromosome. For *real-valued chromosomes*, a user can select the blending crossover and a constraint real-valued mutation. For *permutation chromosomes*, a user can apply the permutation crossover and the swap mutation, thus ensuring only correct individuals are generated.

It is usually necessary to tune crossover and mutation rates for high convergence speed (see subsection 2.2.4), but in most previously-created FPGA-based GA systems the adjustment requires hardware modification and recompilation, which takes many hours to complete. In contrast, our framework allows users to modify these rates at run-time without recompilation (see the examples in subsection 3.5).

3.2.5 Update Controller

The Generic GA enables users to configure the population updating by supporting both generational GA updating and steady state GA updating. The generational GAs normally update all individuals, while the steady-state GAs only update a pair of new individuals. Our Generic GA supports the two methods by using an update controller.

The update controller manages the update of the populations. Users also configure update controller with different replacement methods, such as elite replacement, or randomised replacement. When the update controller receives the offspring, it produces a random address for the individuals to decide the location the offspring will be placed. If users decide to use elitism updating, when the fitness of the incoming individual is higher than the current individual in that address, the individual replaces the current one. Otherwise, the lower fitness individual is thrown away. For example, in the updating stage of Figure 3.5, the four new offspring only replace three existing individuals because one offspring has lower fitness value than an existing individual.



(a) Evaluating: Individuals are evaluated

(b) Selecting: Select four individuals based on the fitness

(c) Evolving: Selected individuals are evolved

(d) Updating: Place individuals into the population

Figure 3.5: GA Pipeline Stages and Update Controller.
Four stages in GA: Evaluating, Selecting, Evolving and Updating. Update Controller controls the way of updating

3.2.6 Random Number Generator (RNG)

As mentioned in section 2.4.3, it is common to use a PRNG on FPGA-based genetic algorithms as it is faster, smaller and repeatable compared with a TRNG [FZS10]. Ref. [TL07] defines a uniform generator, which provides a simple but effective way to generate pseudo-random numbers on the FPGAs. The random seeds can be configured by users at run-time, to explore different number sequences dynamically.

3.3 Pipelining and Parallelism in the Generic GA

The Generic GA system has a pipelined structure and its internal units can also be parallelised. The parallelism and pipelining both increase performance in the Genetic GA, which will discussed in the following subsections.

3.3.1 Pipeline Stages in the Generic GA

The whole pipeline stage of the GA is shown in Figure 3.5. The system has four stages: evaluating, selecting, evolving and updating. In the evaluating stage, offspring are evaluated and assigned the fitness values. In the selection stage, individuals in different locations are chosen by a selection method and pairs of the individuals enter the SCM units. In this evolving stage, the SCM and evaluators will crossover and mutate each of the pairs in parallel, finally producing new offspring. In Figure 3.5, the shapes in grey are the new offspring. The updating depends on the users' choice, Figure 3.5 uses elitism updating.

3.3.2 Hardware Construction of the SCM Unit

When the Generic GA is configured with application specific settings, the framework configures the GA instance with pre-defined units from the library. For example, if a user chooses 'random selection', 'one-point crossover' and 'exchange mutation', the pre-defined hardware units are

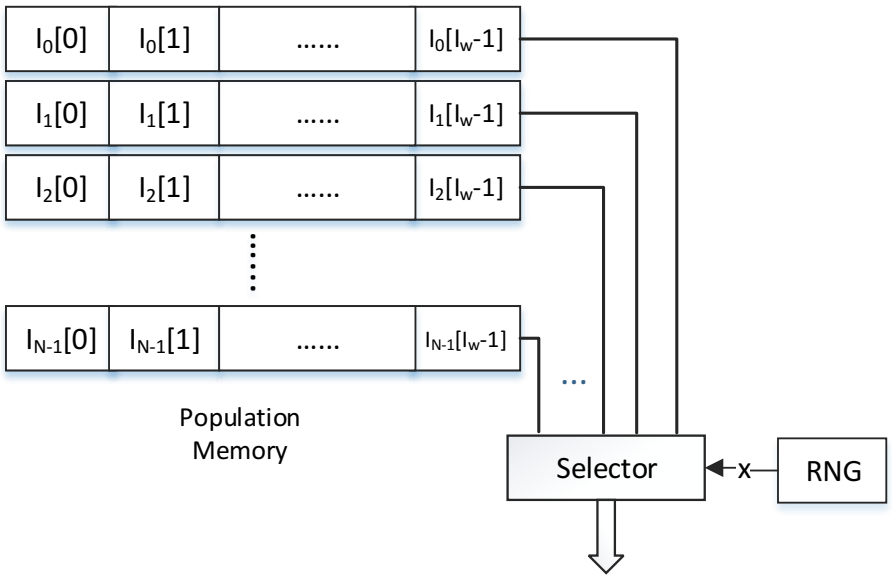


Figure 3.6: Hardware Construction of Random Selection.
W: width of an individual; $I_m[n]$: n-th bit in the m-th individual; x: a random number in the range of $[0, N-1]$; RNG: Random Number Generator; N: population size

selected from the library and then embedded into the Generic GA instance. This subsection shows the pipelined hardware construction for these three units, and also demonstrates the differences between hardware and software implementation.

Random selection is a simple selection method but is still used in practise. First the random number generator produces a random number, and uses the first $\log_2 N_p$ bits to generate a selection signal x. The unit uses signal x as the access address of the individual pool (population), as shown in Figure 3.6.

In one-point crossover, the unit keeps one part of an individual (parent 0) and combines it with the remaining part of the other one (parent 1). On hardware, a random number is first generated from a random number generator and its first $\log_2 W$ bits are used to represent the position of crossover, where W is the width of an individual. So the value of number x is in a range of $[0, W-1]$. Finally we use a bit-vector as the mask for the recombination, as shown in Figure 3.7.

In the swap mutation, the unit exchanges the two items from different positions of an individual. In hardware, we first generate two random positions x and $W-y$ using the same method mentioned in the one-point crossover, and their values are between 0 and $W-1$. Then the

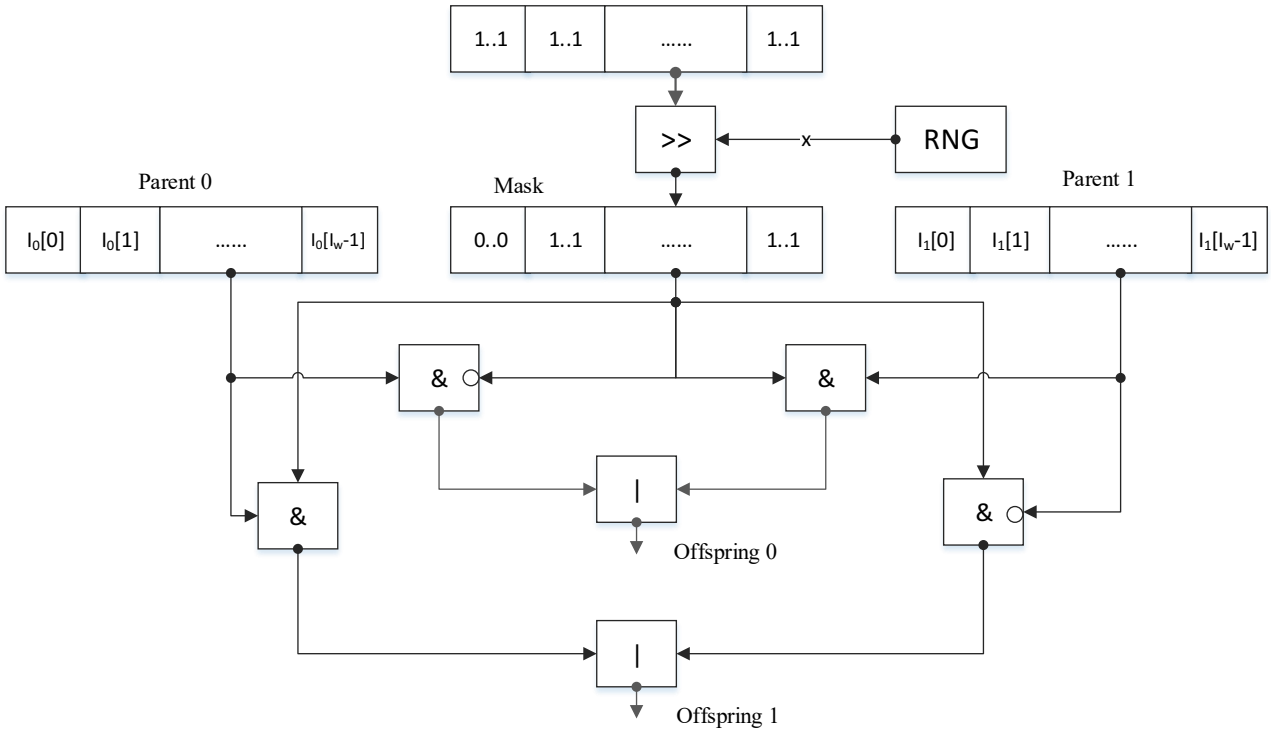


Figure 3.7: Hardware Construction of One-point Crossover.
Two Offspring are generated from two parents. W: width of an individual; $I_m[n]$: n-th bit in the m-th individual; x: a random number in the range of [0, W - 1]; RNG: Random Number Generator

unit uses two bit-vectors as the masks, and performs the logic computation for the exchange, as shown in Figure 3.8.

The hardware implementation differs significantly from software systems because the design has to consider the pipelines, so the construction in this subsection demonstrates the complicated hardware specific designs needed from developers. By providing sufficient pre-defined hardware units, the library really removes barriers for the non-experts users.

3.3.3 Parallelism in Generic GA

The execution time of the Generic GA can be reduced by increasing the parallelism of the architecture. As shown in Figure 3.9, the number of parallel evaluators is controlled by N_E, while the number of parallel SCM units is changed by N_S. A user can adjust the amount of parallelism by easily modifying the two parameters in the input, and the proposed framework

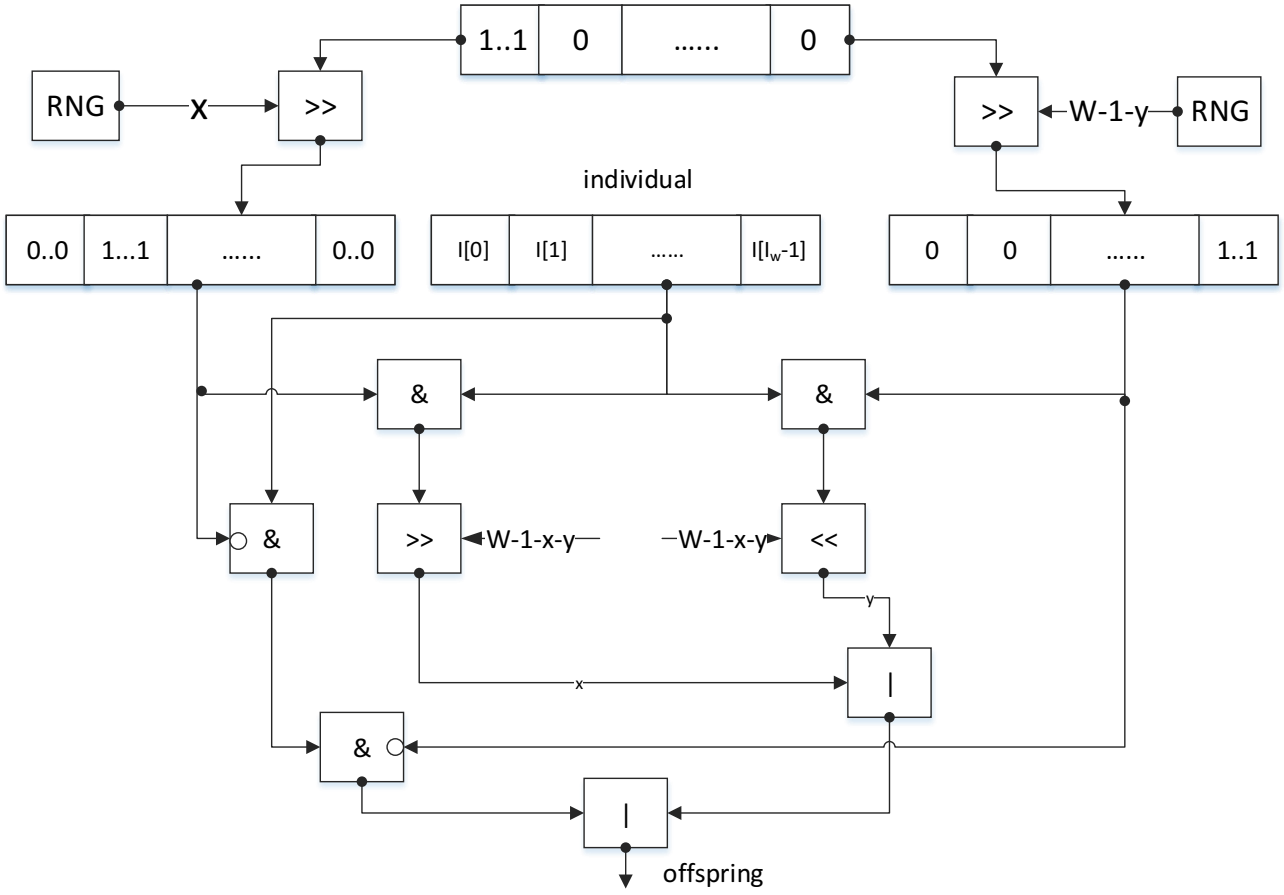


Figure 3.8: Hardware Construction of Swap Mutation.
W: width of an individual; $I_m[n]$: n-th bit in the m-th individual; x: a random number in the range of $[0, W-1]$; y: a random number in the range of $[0, W-1]$; RNG: Random Number Generator

automatically compiles them to an appropriate hardware design.

In Figure 3.9, N_p is the population size. If all the individuals in a population are evaluated in only one cycle after filling the pipelines, there are $N_E = N_p$ parallel evaluator instances. The feedback latency is $L = L_E + L_S$, which is labelled on the left of Figure 3.9. In this case, the resource usage will be very large when N_p is large.

We allow an evaluator to process a population in n cycles after filling the pipeline, in order to solve the resources utilisation problem. Therefore, N_E is reduced to N_p/n, and the feedback latency for one generation becomes $L' = L_E + L_S + n - 1$, which slightly decreases performance if $(n-1)$ is far smaller than $L_E + L_S$. For example, if L_E is 100 and L_S is 50, $n = 5$, the ratio between L and L' will be 150/154.

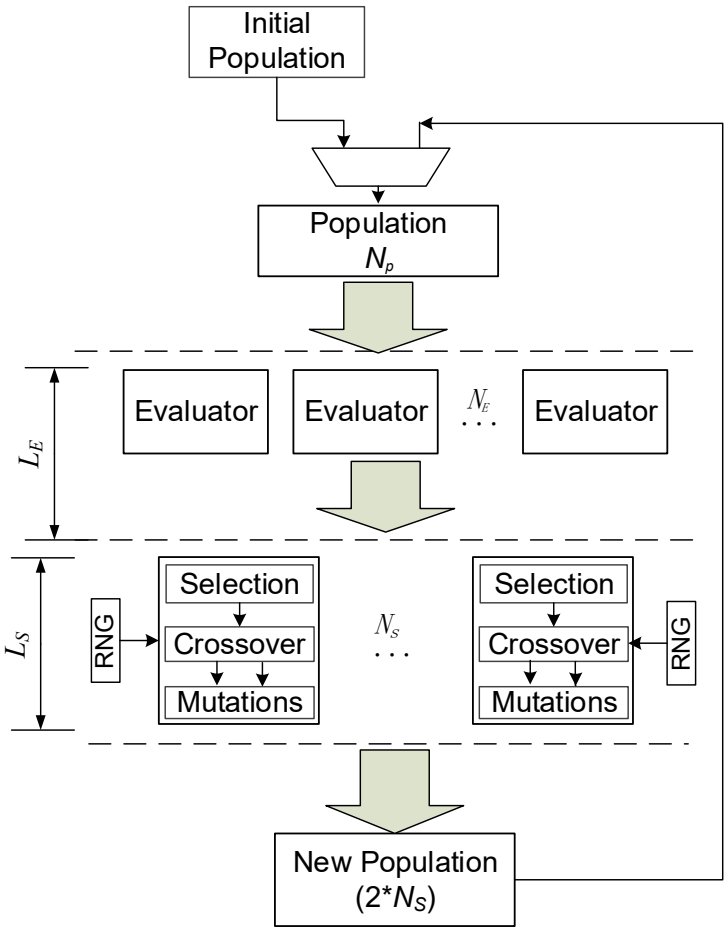


Figure 3.9: Dataflow of the Generic GA.
L_E: latency of evaluator, L_S: latency of SCM units, RNG: Random Number Generator, N_E: Number of Evaluator, N_S: Number of SCM Unit (Selection, Crossover and Mutation)

In the same way, we can also adjust N_S to balance resource usage of SCM units with their performance. By tuning N_E and N_S according to the complexity of the evaluator and SCM units, our platform supports different scales of applications. An example provided in section 3.7 shows how this flexibility affects resources and performance.

3.4 User Defined Input

To promote the framework to non-expert users, the framework only requires them to provide high-level specifications and parameters for the target applications and genetic algorithm, as shown in Table 3.1. The inputs are represented in a high-level domain specific language, which

uses a sub-set of Java. The thesis describes the compile-time and run-time inputs in this section, and presents examples in section 3.5. These high-level inputs demonstrate the high programmability in the framework.

3.4.1 Compile-time Inputs

The compile-time inputs contain two parts, namely application description and architectural parameters.

(1) Application Description

There exist two compile-time sections in the application description:

- $CHROMOSOME$ which contains the names and types of the data elements making up a chromosome. This section is declarative, describing data structures.
- $FITNESS$, which describes the fitness function used to evaluate individuals. This section contains imperative code, which uses input parameters and a chromosomes to calculate a fitness value.

The fitness function uses a sub-set of Java which is then automatically compiled into a hardware implementation. The fitness function contains all standard arithmetic operators (add, mul, etc.) as well as mathematical functions such as sin, log and exp. All components of the chromosome and application parameters are available as implicitly declared variables, which are read either directly or as array look-ups, depending on their types.

A user can declare additional temporary variables within the fitness function of any type, and convert expressions between types, for example from fixed-point (int/uint) to floating-point (float/double). A user can also customise the width of an integer for optimisation, for example, uint8 means unsigned 8-bit width integer.

The fitness function contains multiple statements, which are executed sequentially. The statements can be simple assignment, if-else, or for-loops. Due to the underlying compilation strategy in the tool used by our framework [Tec13], we require that for-loop bounds are statically determined, so that they can later be easily converted into a pipelined representation. These restrictions mean that certain behaviour cannot be expressed such as using a hardware variable as a for-loop bound, but in section 3.5 we show that the high-level description captures various common problems. As mentioned in section 2.3.4, these rules make sure a statement is transferred to a pipelined and parallel architecture, which increases performance in the hardware.

(2) Architecture Parameters

Our framework supports a set of compile-time parameters for the Generic GA. At compile-time, the user can balance resource usage of evaluators and SCM units, by changing N_E and N_S. Those architecture parameters can make the system easier to fit different scales of problems, for example, if the evaluator occupies a large number of resources, we could reduce N_E. For the methods of selection, crossover and mutation, M_s, M_c and M_u can be configured for binary, permutation or real-valued chromosomes.

3.4.2 Run-time Inputs

Run-time inputs make it possible to change the GA system quickly without a long recompilation. There are also two parts in run-time inputs, one being application parameters, the other being GA parameters.

(1) Application Parameters

Most applications have parameters passed to the fitness function, which represents a specific instance of the problem, but in the previous FPGA-based GA systems, a user has to recompile the design to change any of them. To save the long compilation time, our Generic GA can

support all input problems for an application by changing the APP_PARAM section at run-time. For example, assume the target function is $f(x) = a \cdot x + b$, we could define a and b as the application parameters, and change them to support a series of $f(x)$.

(2) GA Parameters

As seen from Table 3.1, we have a series of run-time parameters for the GA. At run-time, N_g controls the number of generations generated, while N_p controls the size of one population.

The framework can also try multiple combinations of run-time parameters, for example trying a list of R_c and R_u to maximise the convergence rate. Changing I_p is also useful if a good prior population exists. These GA parameters can help users find a good configuration for their specific type of application, and they all have sensible default values if a user does not specify them.

3.5 Application to Optimisation Problems

The proposed general purpose framework can be used to solve many problems. In our case, we choose ten standard benchmarks, including the locating problem, the set covering problem, maximum satisfiability problem, the travelling salesman problem, and six other benchmarks to prove our system's functionality and behaviour. This section also shows all the configurations for these problems, demonstrating how the framework provides significant programmability and removes the programming barriers for non-expert users.

3.5.1 Locating Problem

The locating problem attempts to find the best location for an emergency response unit in a city, using the response time to any other location as the target metric. The study presented in [HH04] provides a complex example with a 10 × 10 km city divided into 100 sections.

Compile-time Input

```
Chromosome {float xf,yf;}
Fitness{
  float cost = 0.0;
  for ( int i = 0; i <= 9; i ++ )
  {
    for(int j = 0; j <= 9; j ++)
    {
      float xn = i + 0.5;
      float yn = j + 0.5;
      cost += W[i][j]*sqrt((xn - xf)*
              (xn - xf)+(yn - yf)*(yn - yf));
    }
  }
  return cost;
}
Arch_Param
{
  Ne = 2;
  Ns = 8;
  Ms:  "tournament selection"
  Mc:  "blending crossover"
  Mu:  "constraint mutation"
}
```

Figure 3.10: The Compile-time User Input for the Locating Problem

The response unit can be put at any place in the city, so a solution (x_f, y_f) is a real valued coordinate.

The cost function is:

$$cost = \sum_{n=1}^{100} w_n \sqrt{(x_n - x_f)^2 + (y_n - y_f)^2} \tag{3.1}$$

where (x_n, y_n) is the coordinate of the emergency centre of square n and w_n is emergency frequency in square n.

We use floating-point chromosome, and define the high-level specification and parameters in Figure 3.10 and Figure 3.11. We choose selection, crossover and mutation methods based on the chromosome type, and define the run-time rates according to the features of the problem. All these definitions follow the rules described in section 3.4.

Run-time Input

```
App_Param{ int8 W[10][10] =
          {{0,6,...} {4...}...};
}
GA_Param{
  Ng = 1,000,000
  Np = 32
  Rc = 0.6, 0.5, ...
  Ru = 0.01, 0.02, ...
  Rs = 0x1234, 0xffff
  Ip = {(3.1, 4.5),(2.3, 4.8)...}
}
```

Figure 3.11: The Run-time User Input for Locating Problem

3.5.2 Set Covering Problem

The set covering problem (SCP) is a classic NP-hard combinatorial optimisation problem which has many practical applications [Bal82]. For example in hardware verification, a suite with M tests can test or cover N functions. The relation between test suite and functional points can be represented by an $N \times M$ matrix, as shown in Figure 3.12. If every test has an execution cost, the problem becomes a weighted set covering problem. The aim of the weighted SCP is to find the lowest cost sub-suite of tests which cover as many functional points as possible.

For a candidate suite, the fitness is as follows:

$$Fitness = p \times coverage(suite) - cost(suite) \tag{3.2}$$

where p adjusts the fitness scale, *coverage* represents the number of functions covered by the *suite* and *cost* calculates the total weight of the tests in the *suite*.

As shown in Figure 3.13, in our case, the chromosome is designed as an m-bit binary, and the i-th bit of the binary determines the existence of i-th program in the suite. The run-time input is shown in Figure 3.14, we define a set of GA parameters and application parameters such as cost array and coverage array, which can be changed after one-time compilation.

	t_0	t_1	t_2	t_3	t_4	t_5
p_0	1	1	1	0	0	1
p_1	0	0	0	1	0	0
p_2	0	0	0	0	1	1
p_3	0	1	0	1	0	1
p_4	0	0	1	0	0	1
p_5	0	0	0	0	1	0

Cost[] = {1, 3, 4, 8, 5, 12}

Cov[] = {0x01, 0x09, 0x11, 0x06, 0x24, 0x1e}

Figure 3.12: A Set Covering Problem
$p_i \in \{p\}$ represents the i-th functional point, $t_j \in \{t\}$ means j-th test, '1' in the matrix means if a test covers the point. Cost[j]: cost of t_j. Cov[j]: coverage of $t[j]$. The aim of this problem is finding a set of $\{t_j\} \subset \{t\}$ with smallest cost to cover as many points p as possible

3.5.3 Maximum Satisfiability Problem

The Maximum Satisfiability Problem (MAXSAT) is another classic NP-hard problem used to determine an optimal assignment of a set of boolean variables $\mathbb{V} = \{V_1, V_2, \ldots, V_u\}$. A literal is a boolean variable or its negation, i.e. $L \in \{V, \neg V\}$. A clause C_k is the disjunction (logic "*or*") of m_k literals in the form

$$C_k = \bigvee_{i=1}^{m_k} L_{ki} \tag{3.3}$$

A formula F in the conjunctive normal form is defined as the conjunction ("*and*") of M clauses, i.e.

$$F = \bigwedge_{k=1}^{M} C_k = \bigwedge_{k=1}^{M} (\bigvee_{i=1}^{m_k} L_{ki}) \tag{3.4}$$

Let $\Xi = (V_1/v_1, V_2/v_2, ..., V_u/v_u)$ be an assignment of $\mathbb{V}$. Then the optimisation target of the MAX-SAT problem is

$$g(\Xi) = \max_{\Xi} \sum_{k=1}^{M} \delta(C_k|_{\Xi}) \tag{3.5}$$

where $\delta(C_k) = \begin{cases} 1 & \text{if } C_k = \text{true} \\ 0 & \text{otherwise} \end{cases}$

Compile-time Input

```
Chromosome { uintM suite; }
Fitness{
  uint cost = 0;
  uintN cov = 0;
  for ( int i = 0; i <= M-1; i ++ )
  {
    cov |= suite[i] ?  covs[i]:  0;
    cost += suite[i] ?  costs[i] :  0;
  }
  uint covs_n = 0;
  for (int i = 0; i < N; i ++)
    covs_n += cov[i];
  return (p × covs_n - cost);
}
Arch_Param {
  Mc:  "multi-point crossover"
  Ms:  "roulette wheel selection"
  Mu:  "bit-flip mutation"
}
```

Figure 3.13: The Compile-time User Inputs for the Set Covering Problem

Run-time Inputs

```
APP_PARAM {
  int p = 2,
  uintN covs[M]={...},
  uint cost[M]={...}
}
GA_PARAM{
  Np = 128;
  ...
  Ip = {0xffff, 0x1, ...}
}
```

Figure 3.14: The Run-time User Inputs for the Set Covering Problem

Compile-time Inputs

```
Chromesome { bool v[100]; }
Fitness{
  bool c[430]={false};
  uint count=0;
  c[0] = v[25]|(!v[98])|v(6);
  ...    // remove for space
  c[429] = v[59]|v[91]|(!v(72));
  for ( int i=0; i < 430; i++)
    count += c[i] ?  1:0;
  return count;
}
Arch_Param {
  Ne=8;
  Ns=8;
  Mc:"multi-point crossover";
  Ms:"roulette wheel select";
  Mu:"bit-flip mutation"; }
}
```

Figure 3.15: The Compile-time User Inputs for MAXSAT

Run-time Inputs

```
App_Param{ }
GA_Param {
  Np = 32;
  Ng = 1000000;
  Ru = 0.065;
  Rc = 0.65;
  Rs = 0xffffff;
  Ip = {b10...1
    ,...}
}
```

Figure 3.16: The Run-time User Inputs for MAXSAT

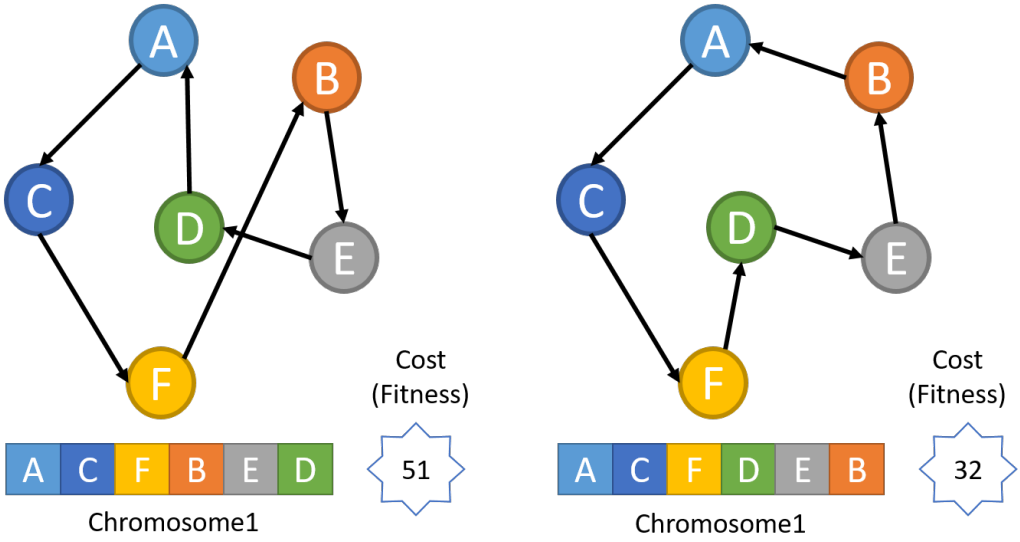


Figure 3.17: The Chromosomes and Their Fitness Values of a Travelling Salesman Problem. Every item in the chromosome represents a city index

We define Ξ as a binary chromosome containing multiple booleans and describe the high-level inputs in Figure 3.15 and Figure 3.16, from the input, we could quickly describe the problem in our framework.

3.5.4 Travelling Salesman Problem

The Travelling Salesman Problem (TSP) is a well-known NP-hard combinatorial optimisation problem: given a list of cities and the distances between each pair of cities we try to find the shortest possible route that visits each city exactly once and returns to the origin city.

Figure 3.17 shows an example of two chromosomes for TSP. To solve this problem, we use permutation chromosomes with a fixed number of items, each being a city index. The sequence of the items in a chromosome represents the order of visits, the two individuals have different fitness values. In our case we test a TSP with 64 cities. We use 'swap' mutation and 'permutation' crossover, as shown in Figure 3.18, to secure the correctness of offspring solutions. We also define the distance array as the application parameters in Figure. 3.19.

Compile-time Input

```
Chromosome { uintM city[64];
}
Fitness{
  int totaldis = 0;
  for (int i = 0 ; i < 63; i ++)
  //add distances of visits
  {
    totaldis += Dist[city[i]][city[i+1]]
  }
  totaldis += Dist[city[0]][city[63]]
  return totaldis
}
Arch_Param{
  Ns = 1;
  Ne = 1;
  Mc:  "permutation crossover"
  Ms:  "roulette wheel selection"
  Mu:  "swap mutation"
}
```

Figure 3.18: The Compile-time User Inputs for TSP

Run-time Input

```
App_Param{
  int Dist[64][64] = {{...}}
}
GA_Param{
...
}
```

Figure 3.19: The Run-time User Inputs for TSP

Table 3.2: GA Benchmarks and Their Parameters

Name	Fitness Function	Parameters
RB	$\sum_{i=1}^{n-1}\left[100\left(x_{i+1}-x_i^2\right)^2+\left(x_i-1\right)^2\right]$	$-10 \le x_i \le 10$
BF6	$4096+\{[(x^2+x)\times cos(x)]/220\}$	$0 \le x \le 65535$
BF7	$32768+\{56\times[x\times sin(4x)+1.25\times y\times sin(2y)]\}$	$0 \le x, y \le 255$
2DS	$65535-174\times\left(150+\{\prod_{k=1}^{2}\sum_{i=1}^{5} i\times cos[(i+1)\times x_k+i]\}\right)$	$0 \le x_1, x_2 \le 255$
F11	$1+\sum_{n=1}^{N} x_n^2/4000-\prod_{n=1}^{N} cos(x_n)$	$-10 \le x_n \le 10.0$
F7	$x\times sin(4\times y)+1.1\times y\times sin(2\times y)$	$0 \le x, y \le 10.0$

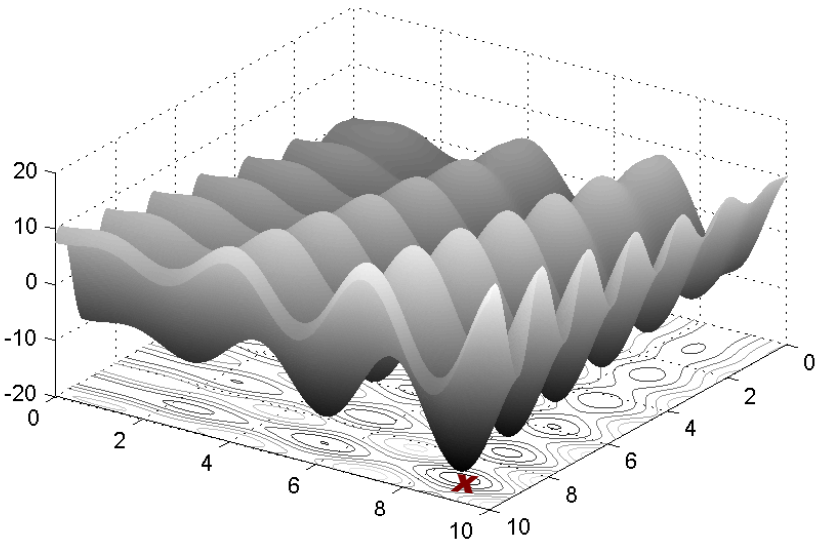


Figure 3.20: Search Space of F7.
F7 has a number of local optima, and red cross shows the best fitness

3.5.5 GA Benchmarks

To test the ability of solving numeric problems in our Generic GA, we use the non-convex Rosenbrock function (RF) [KS09], four GA benchmarks from [FZS10], including binary F6 (BF6), binary F7 (BF7) and 2-D Shubert function (2DS), and two benchmarks called F7 and F11 from [HH04]. As shown in Table 3.2, these functions have one or more parameters, with real-valued or binary variables and application parameters. They are popular performance test functions for optimisation algorithms, because searching the optimum for them is hard. For example, the global minimum of RF hides in a long narrow valley. Figure 3.20 shows the solution search space of F7. The search space of F7 contains many local optima that may trap the search algorithm. The red cross in the figure shows the theoretical best solution. These problems have different parameters according to their features, for example the run-time input for the BF7 problem is shown in Figure 3.21.

Table 3.3 summarises the problems in this section, and shows that our framework covers many different kinds of problems. From these high-level inputs for all the optimisation problems and benchmarks, we can see the framework increases the programmability of FPGA-based GAs, and also removes the barrier of hardware for non-expert users.

Run-time Inputs

```
App_Param{

}
GA_Param
{
    Np = 32;
    Ng = 1000000;
    Ru = 0.0625;
    Rc = 0.625;
    Rs = 0xfffeee;
    Ip = {(10,10), (0,220)
    ,...}
}
```

Figure 3.21: The Run-time User Inputs for BF7

Table 3.3: Problem Summary of Chromosome Type, Gene Number, Application Parameters and Fitness

Application	Chromosome Type	Gene Number	Application Parameters	Fitness
Locating	real-value	2	emergency array	real-value
SCP	binary	test number	cost, cov, p	integer
MaxSAT	binary	clause number		integer
TSP	permutation	city number	city distance	integer
BF6	binary	1		integer
BF7	binary	1		integer
2DS	binary	2		integer
F11	real-value	4		real-value
RB	real-value	4		real-value
F7	real-value	2		real-value

Table 3.4: Qualitative Comparisons of FPGA-based GAs.
Chrome: Chromosome, Pop.: population, App: application, Spec: specification. Param. Parameters

First Author	Year	Chrome.	Run-time GA param.	App. param.	Parallel param.	Initial pop.	App. Spec.	Platform
Scott	1995	binary	-	fixed	-	rand	low	BORDG
Yoshida	1999	binary	-	fixed	-	rand	low	SFL
Shackleford	2001	binary	-	fixed	-	rand	low	AXB-MP3
Aporntewanet	2001	binary	-	fixed	-	rand	low	Xilinx V1000
Tang	2004	binary	N_p, N_g, R_c, R_u	fixed	-	rand	low	PCI System
Vavouras	2009	binary	N_p	fixed	-	rand	low	Virtex2 Pro
Fernando	2010	binary	N_p, N_g, R_u, R_c, R_s	fixed	-	rand,	low	Virtex2 Pro
Our work		binary, real permutation	$N_p, N_g, R_u, R_s, R_c, I_p$	run -time	N_E, N_S	rand, I_p	high	MAX3 (V6-SXT475)

3.6 Qualitative Comparison

We compare the features of previous FPGA-based GAs to our framework in Table 3.4. Our Generic GA is more flexible and easier to use than other FPGA-based GA systems, with the following advantages:

1. Our framework provides a unified platform for binary, permutation, and real-valued chromosomes on a flexible structure. The selection, crossover and mutation methods (M_s, M_c, M_u) can be changed for different kinds of problems. There exists a set of parameters changeable at run-time, which are crossover and mutation rates (R_c, R_u), the population size(N_p), maximal generation number (N_g), the random seed (R_s), in particular application parameters and initial population (I_p), which are run-time changeable only in our framework. Specifically, the changeable application parameters make it possible to execute different inputs for an application without recompilation. Some approaches also support several run-time parameters, but require users to have hardware knowledge to change them [FZS10, TY04].

2. The framework gains the significant flexibility and parallelism in the architecture, as the degrees of internal parallelism can be tuned by non-expert users. The framework allows a user to decide the number of parallel evaluation (N_E) and SCM units (N_S)

at a high level to adjust resource usage, without any manual modification of hardware code. Furthermore, as described in subsection 3.3.3, the customisable parallelism makes it possible to support different scale applications.

3. The chromosome and the fitness function of a new application are defined in a high level description language, making it easy for a user to apply our Generic GA, without writing any low-level hardware code in VHDL or Verilog.

3.7 Evaluation

This section will use our framework, a CPU-based GA and a GPU-based GA to solve the problems from section 3.5. The experiments demonstrate the high performance, high architectural flexibility and run-time flexibility in our framework.

3.7.1 Hardware Implementations

To test the optimisation problems shown in section 3.5, we select an FPGA-based acceleration platform containing Virtex 6-SX475T for hardware implementation [Tec13]. We compare our FPGA-based system with software and GPU-based solutions quantitatively. We implement software counterparts based on the third-party GA [Col03] and GALib on an 2.67GHz Dual Intel Xeon X5650 CPU system, which has 12 physical cores and 24 threads in total. The number of threads used is optimised based on the communication cost. The CPU code is well tuned using multi-threading techniques including Pthreads and SIMD, and the code is compiled by Intel C compiler with the highest optimisation level.

For the maximum satisfiability problem (MAXSAT), we also compare our Generic GA with a third-party GPU-based work on an nVidia Tesla C1060 [MWMA09]. We have compared our system with other FPGA-based systems qualitatively in section 3.6, but it is very difficult to compare quantitatively, because they use different platforms and do not provide enough details

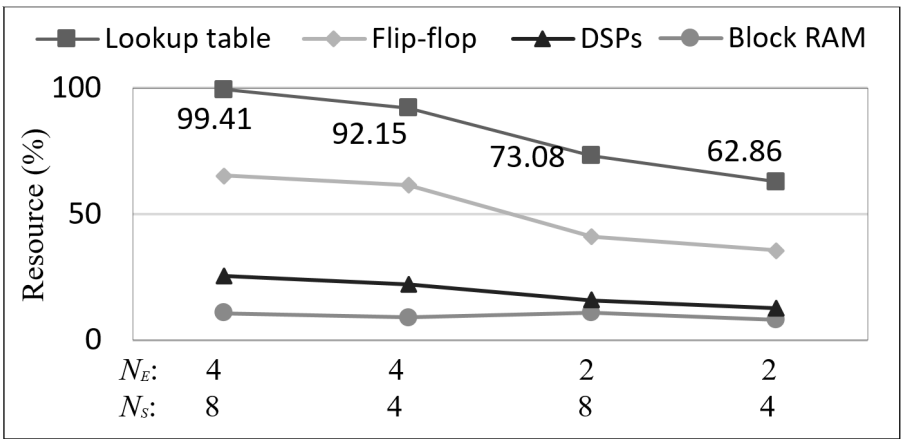


Figure 3.22: Resources for Different N_E and N_S in a Locating Problem.
N_E: number of evaluators, N_S: number of SCM units

about execution time. Because some problems share the similar features, this section selects a subset of the problems to test the following three aspects:

- Architectural flexibility: location problem
- Run-time flexibility: the set covering problem, MAXSAT, benchmarks
- High performance: all the problems

3.7.2 Architectural Flexibility

The framework allows a user to configure GA parameters, including the internal parallelism and configuration of genetic operators. The number of parallel evaluators and SCM units can be changed via N_E and N_S. As shown in Figure 3.22, we can tune these parameters for the locating problem in order to adjust resource usage and performance. For example, if we reduce N_E from 4 to 2, we would obtain a slightly decreased performance (3% slower) due to the pipelined structure.

3.7.3 Run-time Flexibility

The framework allows a user to configure the application at run-time for different instances, to avoid recompilation. As shown in Figure 3.11, in the run-time parameters for the locating

problem, weight array W is defined as an application parameter. By changing W, we can use the same Generic GA to solve multiple input problems without recompilation, which would always needs several hours to finish. We also define a series of run-time parameters to tune convergence speed. Our framework will try a full combination of them, including lists of mutation rates (R_u), crossover rates (R_c), and random seeds (R_s). The execution report helps users to find the best configuration for the problem.

For the set covering problem, we also choose different methods of selection, crossover and mutation from the locating problem. The coverage array ($cov[M]$), cost array ($cost[M]$) and p are supplied at run-time, as shown in Figure 3.14. We test an instance of Steiner triple systems (STS_{27}), which is considered as a hard SCP with a matrix of 117×27 [PP02]. For the MAX-SAT problem, we apply the Generic GA system to a hard instance uf100-430, which contains 100 variables and 430 clauses [LA11].

3.7.4 High Performance

To compare execution time with the multi-core CPU, we let the Generic GA run 1,000,000 generations for different population sizes (N_p). We choose 1,000,000 generations because an GA always needs to run for a large number of generations before it reaches a desirable result. As shown in Figure 3.23, our Generic GA is on average 28 times faster than the multi-core CPU. Although the initial compilation of the Generic GA is slow and takes hours to finish, future executions of the same Generic GA system with different parameters require no compilation, and start evolution immediately.

Our platform outputs the results from the FPGA to the CPU for comparison. As shown in Table 3.5, our platform effectively solves different types of applications with a geometric mean speed up of 28 times, including discrete combinatorics, numeric, real-valued and permutation optimisation. The results also show our framework has a lwo latency.

Our Generic GA finds a location within 96% of the best fitness according to [HH04] for the locating problem, as well as finding good solutions for all other applications. The clock frequency

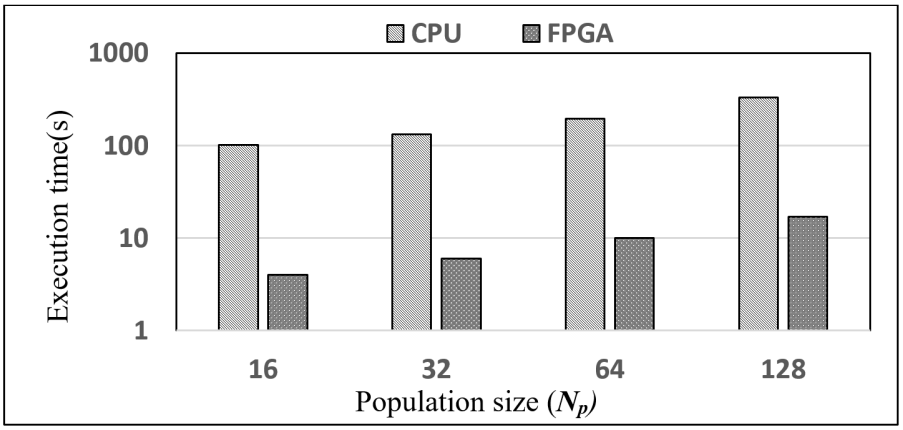


Figure 3.23: The Execution Time for Different N_p in a Locating Problem.
N_p: number of individuals in one population

Table 3.5: The Configurations and Performance Results of the Generic GA Systems. Res.: Resources. Mean: Geometric Mean

App.	N_E	N_S	Res.%	Speed-up	Quality	Description
Locating	2	8	73.08	24	96%	real-valued computation
STS_{27}	16	16	65.12	45	100%	discrete combinatorics
MAX-SAT	8	8	74.96	22	100 %	discrete combinatorics
TSP	1	1	75.85	28	90 %	permutation chromosomes
BF6	8	8	26.45	26	100%	numeric computation
BF7	8	8	21.48	25	100%	numeric computation
2DS	8	8	68.89	31	100%	numeric computation
F11	16	16	60.05	27	100%	real valued computation
F7	8	8	36.21	28	96%	real valued computation
RF	8	8	60.05	29	98%	real valued computation
Mean	-	-	-	28	-	

of our Generic GA is set to a conservative value between 75MHz and 100MHz, so with longer compile times higher speed-ups are possible. In the Table 3.5, the resource usages vary with the complexity of an application, the number of evaluation units (N_E) and SCM units (N_S).

Reference [FZS10] achieved a speed-up of around 5 times over BF6, BF7 and 2DS without reporting the exact execution times, thus making it difficult for us to directly compare their system's performance with ours. However, despite using high-level inputs, our automated framework still achieves high performance (28 times faster than a multi-core CPU) while retaining flexibility. The performance is compared for the same evaluation and same results. For the MAX-SAT problem, compared with the GPU-based work [MWMA09] for the same problem, our system obtains a 5 times speedup.

3.8 Summary

Genetic algorithms need FPGA-based acceleration because of their long execution time. To provide an easy way for users to create and execute FPGA-based GAs with binary, real-valued and permutation chromosomes, this chapter has presented an automated unified framework for general-purpose GA systems.

Our framework contains a scalable and customisable generic GA, with sufficient programmability to allow the user to tune resource usage, without directly modifying hardware design. At compile-time, the user just needs to define a high-level specification of an application, including chromosome and fitness function, without writing any hardware code using VHDL or Verilog. At run-time, the user can change the GA and application parameters without waiting for recompilation.

When compared with existing FPGA-based GAs, the generic GA has more architectural flexibility, making it much easier for users to take advantage of FPGA resources. Compared with a multi-core CPU over ten applications, the geometric average speed-up of the Generic GA is 28 times, showing the high performance of designed framework.

To make full use of all the resources in the FPGA and to avoid getting stuck by having only one search process, the framework will be extended in the next chapter to support multiple Generic GA instances in one FPGA. If the resources in one FPGA are still not enough, involving more FPGAs is a possible solution. A multiple-level parallel framework will be presented in the next chapter, to explore both internal and external parallelism of the Generic GA.

Chapter 4

pGA: Adding Parallelism to the Generic GA

As shown in chapter 2 and chapter 3, FPGA-based Generic GAs have been effective in solving optimisation problems in less time than CPUs. This chapter now explores methods for increasing the parallelism of the Generic GA, namely the parallel GA, in order to increase performance. Parallel GA (pGA) is a variety of GA in which a number of GA instances evolve in parallel and exchange information with each other. As the search processes of GA instances are conducted in different areas of the search space, pGAs are likely to decrease the execution time needed to find high quality solutions [LA11]. There have been previous attempts to adapt pGAs to FPGAs, either for acceleration or reduced power consumption, which are listed in chapter 2. However, designing an FPGA-based pGA is not as easy as adding multiple GA threads in a microprocessor, or simply replicating blocks in the hardware, the developer must also consider communication and synchronisation between GA instances.

This chapter extends the framework in chapter 3, providing an easy tool for parallel GAs not only in one FPGA (section 4.1), but also in multiple FPGAs (section 4.3). The arrangement of this chapter is as follows:

- Section 4.1 explores Generic GAs in one FPGA, by adding a migration unit to the Generic

GA. The migration unit controls all the communication between the parallel GA instances.

- Section 4.2 compares our pGA system with existing pGA systems in one FPGA, and also evaluates the performance of pGA systems.
- Section 4.3 explores multi-level parallel Generic GAs in more than one FPGA, with an inter-FPGA migration unit controlling the communication.
- Section 4.4 compares our pGA system with existing pGA systems in multiple FPGAs, and also evaluates the performance of pGA systems.
- Section 4.5 discusses how the migration parameters affect performance, resulting in guidance for tailoring migration.

4.1 Intra-FPGA Parallelism: pGA in one FPGA

The Genetic GA in chapter 3 provides a flexible architecture that can be extended. However, adding parallelism to the Generic GA is not as straightforward as doing so in a software GA where we can add threads. The major challenges behind a good hardware pGA design include:

- Communication. The migration between GA instances requires communication, so the design of the migration scheme must be closely related to hardware resource consumption as well as pGA performance.
- Parallelism. pGA systems are more complex than single-thread ones because the user needs to control the parallelism in addition to the evolutionary process.
- Additional run-time flexibility. It might be necessary to modify the system including communication and migration parameters. As recompilation may take a long time, a useful pGA system should avoid unnecessary recompilation when changing these parameters.

To address these challenges and make the pGA designs accessible to non-expert users, the framework proposed in chapter 3 is extended to be:

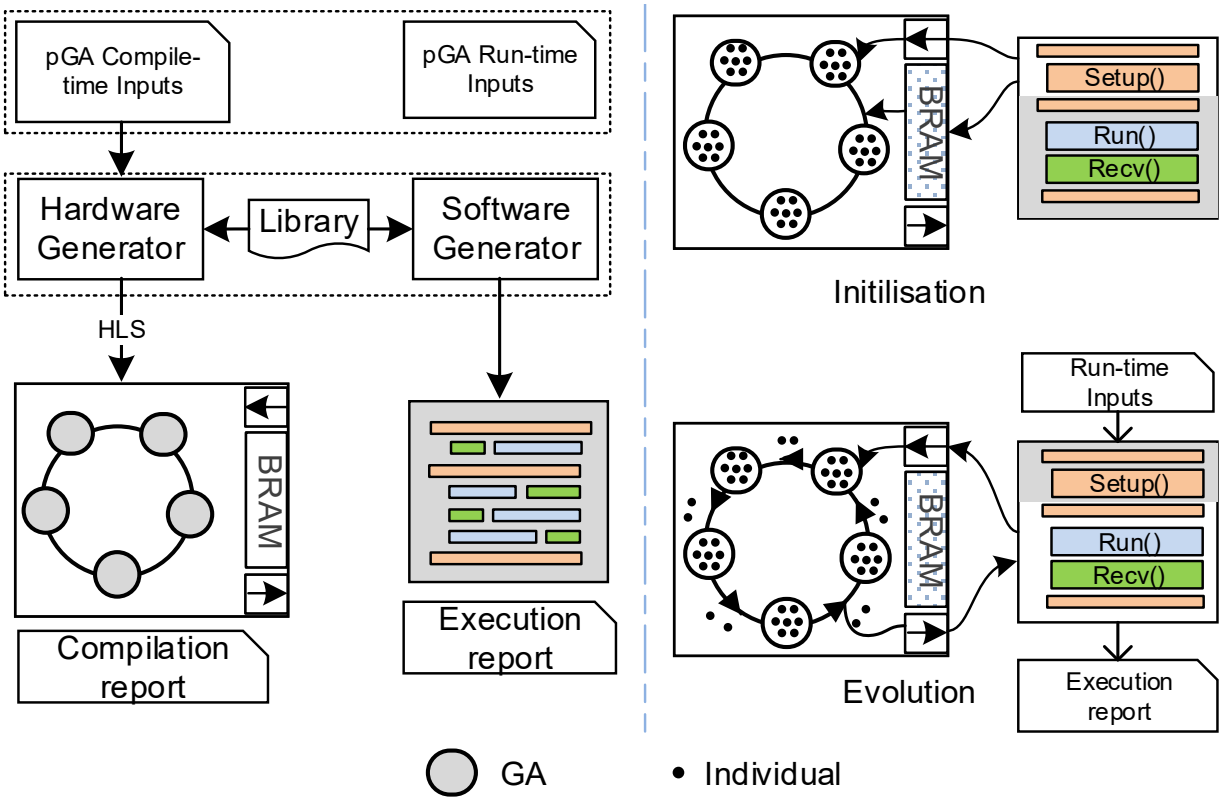


Figure 4.1: The Design Flow (Left) and Execution Flow (right) of pGA Framework Adapted from Figure 3.1, Figure 3.2 and Figure 3.3
HLS: High Level Synthesis

- A general-purpose pGA framework for creating and running multiple GA instances with an effective communication and migration scheme.
- A customisable pGA architecture which exploits both inter-FPGA and intra-FPGA parallelism, allowing non-expert users to specify problem settings and architecture options at a high level without hardware design expertise.
- A fine-tuning mechanism for the pGA architecture, enabling users to tune a set of run-time parameters including communication behaviour without recompiling the design. The user can test the pGA systems with different parameters for each GA instance.

The design and execution flow of framework also becomes Figure 4.1. As can be seen, there are more than one GA kernel on one FPGA.

After a qualitative comparison with existing FPGA-based pGA systems, this work shows improved flexibility in architecture and better run-time tuning in communication. We compare

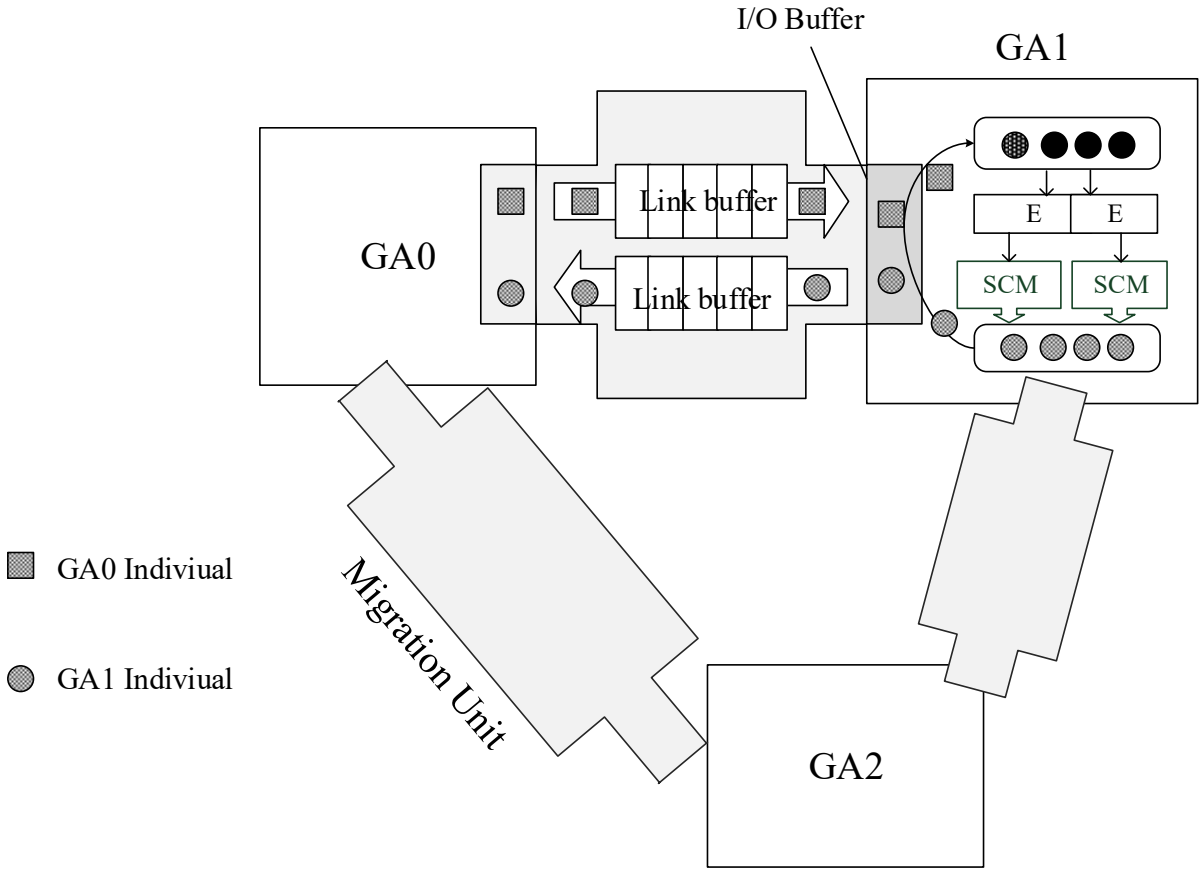


Figure 4.2: Migration Unit and Migration Between GA Instances in One FPGA. Migration Unit contains Input/Output (I/O) buffer and link buffer for sending and receiving individuals. E: Evaluator, SCM: Selection, Crossover and Mutation

our work with single-instance GA, microprocessor, and GPU-based work, the results show the benefits of parallelising GAs in FPGAs. We reach these goals by adding a migration unit (subsection 4.1.1) and architectural parameters to the Generic GA in chapter 3, which are described in the following subsections.

4.1.1 Migration Unit

Migration units control how Generic GA instances exchange individuals. A migration unit has an input buffer and an output buffer for receiving and sending individuals, and link buffers to connect other GA instance. These buffers may contain multiple channels when connected to more than one neighbour. A channel transfers a part of a GA's population to one of the

neighbouring GAs. Several parameters control the behaviour of migration, including migration interval, migration topology, and migration rate [CP98]. However, in the previous pGA systems in FPGA, only one or two parameters are supported. We define all the parameters in our framework to decide the migration policy as follows:

- Migration topology. The topology (T_m) determines how GA instances are connected. The popular topologies are chain, ring, grid, island and cube, as shown in Figure 2.16. The island topology uses isolated GA instances without any communication; the chain topology connects all the GA instances in a chain; the ring topology is the chain topology but with the head and the tail connected together. The grid topology puts all the GA instances in a grid structure, with all GA instances connecting to its four neighbours. To reduce the complexity of links, our framework supports all four topologies except the complex ones such as cube structure.

- Migration interval. The migration interval (I_m) controls the frequency of data exchange. Some papers use a similar parameter called migration period, which controls the generation gap between two migrations [Pit95].

- Migration rate. The parameter R_m determines the percentage of individuals to be exchanged during one migration. Some papers use number of migrants to indicate the rate [Bel95].

By setting these parameters, we can control the migration between GAs. For example, if we assume $I_m = 20$, R_m is 10% and $N_p = 100$, 10 individuals in a GA instance will be sent to neighbour GAs every 20 generations.

As shown in Figure 4.2, the length of the link buffers are different, the length between GA0 and GA1 is larger than that between GA1 and GA2. To avoid waiting for data from neighbour kernels, the migration unit uses a non-blocking protocol which continually streams data over the channels and receives the incoming data. For example in Figure 4.2, GA1 place I_m individuals in output buffers every I_m generation, and then keeps sending them into its link buffers to GA0 and GA2. At the same time, GA1 also receives the GA0's individuals, and update them into

Table 4.1: The Parameters of User Input for Parallel GA

	Architecture Parameters		GA Parameters
N_k	**number of GA kernels**	N_p	sub-population size
N_E	number of evaluator	N_g	maximal generation
N_S	number of SCM units	R_u	mutation rate
M_c	crossover method: "one-point","multi-point", "blending","reordering"	R_c	crossover rate
		R_s	random seed
M_s	selection method: "roulette wheel" "elitism", "tournament"	I_p	initial population
		R_m	**migration rate**
M_u	mutation method: "bit-flip","constraint","swap"	I_m	**migration interval**
T_m	**migration topology: "island", "chain", "ring", "grid"**		

local population. The migration topology in Figure 4.2 is ring, and can be configured as chain, grid and cube in one FPGA.

4.1.2 Enhanced Flexibility

The architecture can be easily configured by the user, as we add two more compile-time architectural parameters to those already seen in chapter 3, including the degree of parallelism through the numbers of parallel GA kernels (N_k) and migration topology (T_m). As shown in Table 4.1, we now have seven architectural parameters in total. The framework provides the users high level programmability to modify these sttings.

In this way, users gain the ability to control the hardware layout via the parametrisation of external and internal parallelism, without needing to understand how to modify the FPGA architecture directly. In a kernel, $N_E = N_p$ and $N_S = N_p/2$ imply maximal parallelism and resource utilisation, but minimal execution time per generation, as discussed in section 3.3.3. Naturally, if the whole population size $N_P = N_k \times N_p$ exceeds the available hardware resources then the combination of N_k, N_S and N_E must be chosen carefully. Our framework enables a user to select suitable values of these parameters to maximise FPGA resource utilisation or performance. We also add two more run-time parameters R_m and I_m to control migration rate and migration interval respectively without time-consuming recompilation.

Table 4.2: Qualitative Comparisons of FPGA-based pGAs. Param.: Parameters

Work	Year	Migration Topology	Migration Param.	Kernels' Param.	Paral Evaluator	Paral SCM	Platform
[YY99]	1999	ring	fixed	-	yes	no	SFL
[CC00]	2000	ring	fixed	-	yes	no	PCIGEN10K
[JKFA06]	2006	grid	fixed	-	–	yes	Stratix EPS1S10
[TMSY06]	2006	chain	fixed	identical	no	no	Cyclone FPGA
[KK12]	2012	ring, island	fixed	identical	no	no	Virtex 4 (XC4VLX25)
[SAF13]	2013	net	fixed	identical	no	no	Virtex 6 (XC6VLX240T)
Ours		ring, grid island, chain	R_m T_m, I_m	identical, different	N_E	N_S	Virtex 6 (SX475T)

4.1.3 Qualitative Comparison

We summarise the features of our framework and compare with other systems mentioned in chapter 2. In Table 4.2, it is clear that our pGA system makes it easier to explore parameters and is more flexible than others because:

- Many parameters including the migration parameters shown in Table 4.1 are modifiable at run-time, enabling a user to effectively tune them in any kernel without time-consuming recompilation. The parameters can be configured either identically or differently for each kernel, allowing users to try different configurations.
- We gain the maximal flexibility and parallelism in the architecture, because the degrees of both external and internal parallelism as well as migration topology can be tuned by non-expert users.
- We support four types of topologies, namely chain, ring, grid and island, while the previous systems support only one or two topologies.

4.2 Evaluation of pGA on One FPGA

Global optimisation can be applied to many real-world applications, both of discrete and non-discrete nature. We know that a general global optimisation algorithm is usually less efficient

than specific versions tuned to the problem at hand, however it is still valuable to gauge the baseline performance of a global optimisation scheme using benchmark problems.

Therefore, here we choose five problems with different features, including an NP-hard combinatorial optimisation problem and four numeric problems from section 3.5. We select an FPGA-based acceleration platform containing a Virtex 6-SX475T for hardware implementation [Tec13]. A direct fair comparison with other FPGA-based pGA systems is still very difficult as they use different platforms and do not provide enough details of their execution time, but we can still compare the pGA system with the single-kernel Generic GA in chapter 3. In addition, we compare our FPGA-based pGA with the software GALib mentioned in chapter 2 and GPU implementation quantitatively. We implement software counterparts on a dual core Intel Xeon X5650 CPU, which is optimised using the same configuration mentioned in chapter 3 but uses parallel GA instances.

4.2.1 Maximum Satisfiability Problem

MAXSAT is a classic NP-hard problem to determine an optimised assignment of a set of boolean variables $\mathbb{V} = \{V_1, V_2, \ldots, V_u\}$, as described in section 3.5. Following the definition in section 3.5, we still define the same Ξ as binary chromosomes and but describe a different high-level input in Table 4.3, then our framework creates a hardware solution with four kernels. As can be seen in Tables 4.3 and 4.4, the compile-time and run-time parameters are different for every kernel. We start each kernel from different initial populations, and also specify different GA parameters for them. Here we apply the pGA system to a hard instant uf100-430 [LA11]. Due to the stochastic nature of meta heuristics, we run a number of independent experiments to gain sufficient data.

(1) Resource Usage and Compilation Time

We use our framework to generate FPGA-based pGA systems with different compile-time configurations. We use pGA_n^m to represent a parallel GA with m independent islands, each of

Table 4.3: The Compile-Time User Inputs of pGA for MAXSAT

Compile-time Inputs of pGA for MAX-SAT

```
Chromosome { bool v[100]; }
Fitness{
  uint count=0;
  bool c[430]={false};
  c[0] = v[25]|(!v[98])|v(6);
  ...    // remove for space
  c[429] = v[59]|v[91]|(!v(72));
  for ( int i=0; i <= 429; i++);
    count += c[i] ?  1:0;
  return count;
}
Arch_Param {
  Nk=4;
  Tm ="ring";
  K1{ Ne=4;
    Ns=4;
    Mc:"multi-point crossover";
    Ms:"roulette wheel select";
    Mu:"bit-flip mutation";
  }
  K2{ Ms:"elitist select"...}
  ...
  K4{ Mc:"one-point crossover"...}
}
```

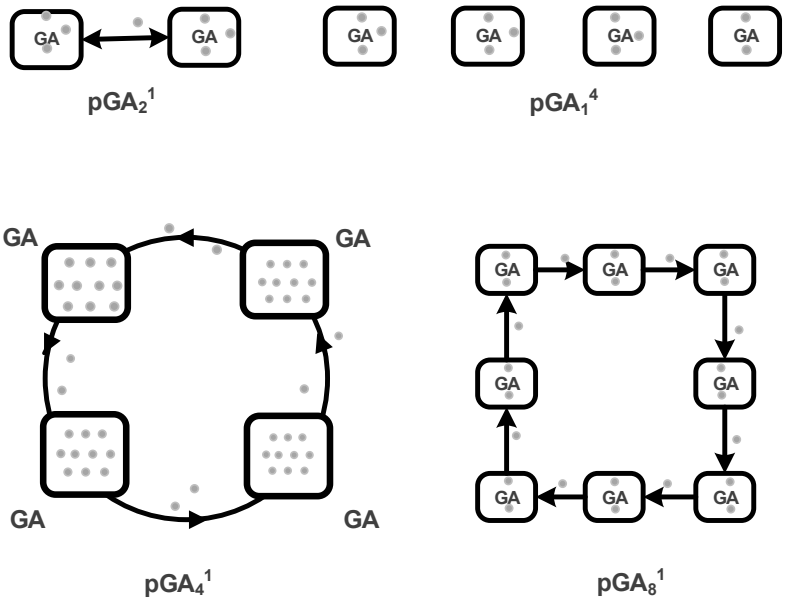


Figure 4.3: Configurations of pGAs: pGA_2^1, pGA_4^1, pGA_1^4, pGA_8^1

Table 4.4: The Run-Time User Inputs of pGA for MAXSAT

Run-time Inputs

App_Param{ }
GA_Param{
K1{Np = 32;
Ng = 1000000;
Ru = 0.065;
Rc = 0.65;
Sr = 0xffffff;
Rm = 0.2;
Im = 8;
Ip ={b1010...1,...}
}
K2{
Ip ={b1110...0,...}...
Rc = 0.75; } //kernel 2
...
K4{
Ip ={b1011...1,...}...
Ru = 0.055; } //kernel Nk
}

which contains n GA instances connected. As shown in Figure 4.3, the configurations for these pGA systems are:

- pGA_1^1: 1 isolated pGA, equivalent to the single kernel GA from previous chapter.
- pGA_2^1: 2 parallel GA instances connected with chain topology.
- pGA_4^1: 4 parallel GA instances connected with grid topology.
- pGA_1^4: 4 islands, each contains one GA inside.
- pGA_8^1: 8 parallel GA instances connected with ring topology

The different resource usages and compilation times are shown in Table 4.5 with various numbers of kernels (N_k), internal evaluators (N_E) and SCM units (N_S). As can be seen, when maintaining the same total population, pGA systems can not only have smaller resource usage, probably because of reduced fan-out, but also have shorter compilation time due to replication

Table 4.5: Resource Usages of Virtex-6 SX475T and Compilation Times of Five pGA Systems for MAX-SAT

Configuration	pGA_1^1	pGA_2^1	pGA_4^1	pGA_8^1	pGA_1^4
kernel number N_k	1	2	4	8	4
N_E of a kernel	8	4	2	1	2
N_S of a kernel	8	4	2	1	2
Resource Usage(%)	74.96	50.46	41.44	32.46	39.89
Compilation Time	5h38m	4h02m	2h28m	1h49m	2h10m

of the same structure in place and route. The change in resource usage is non-linear, as the streaming links also occupy resources, as can be seen in the table.

(2) Performance

We record the execution time of FPGA-based pGA designs and their CPU-based counterparts, i.e. CPU4 is the counterpart of pGA_4^1. Our systems have an average speedup of 24 times over the multi-core CPU. Moreover, when finding the global optimum, the pGA_4^1 system gains 6.8 times speedup compared with the GPU-based system proposed in [MWMA09] on the nVidia Tesla C1060.

We also observe the best fitness found under these different configurations as a function of run time, shown in Figure 4.4. We can see the pGA systems can achieve better results than the CPU-based ones in general. We also obtain the following three observations.

First, the conventional GA_1^1 and independent pGA_1^4 reach plateaus, but the systems with multiple GA instances and migration reach better solutions. This is probably because GA_1^1 and pGA_1^4 are vulnerable to local optimal solutions while other configurations avoid similar circumstance by exploring different sub-spaces and by sharing information. The observation proves the importance of accelerating pGA with migration in terms of solution quality.

Second, FPGA-based systems generated by our framework offer good solutions in less time than CPU-based systems. This is because the FPGA-based systems compute more generations per unit time. This observation suggests the practical usefulness of the systems as a user always wants satisfactory results within shorter time.

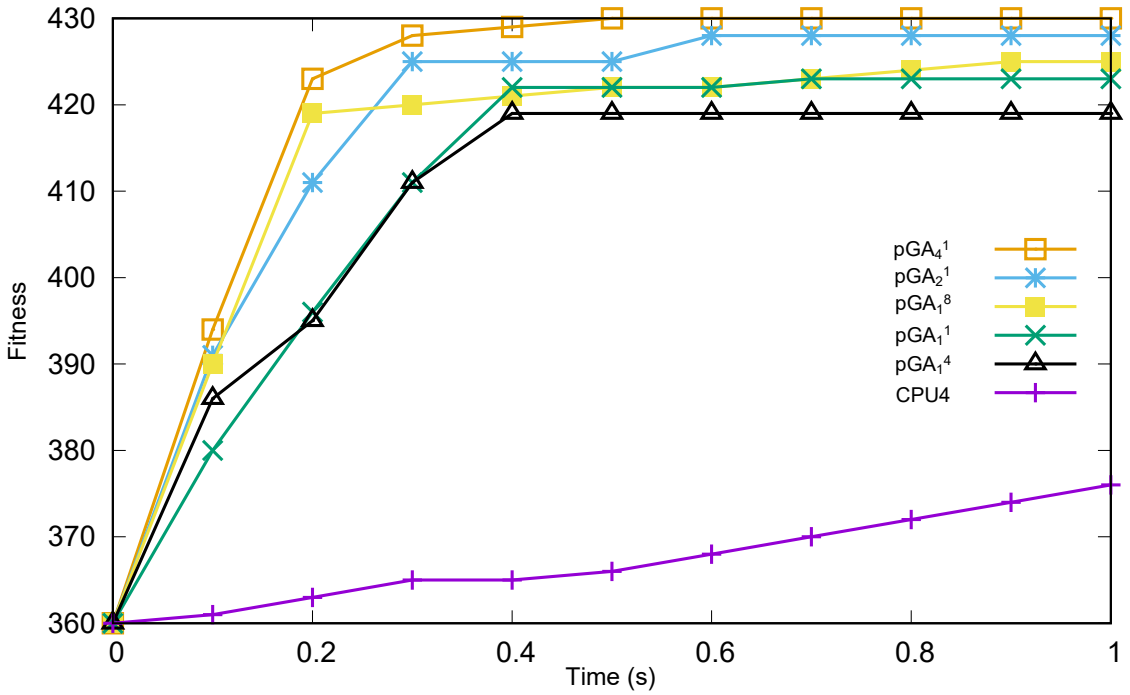


Figure 4.4: Best Fitness Found by pGA Systems for MAX-SAT.
pGA_n^m: a pGA with m independent islands, each of which contains n GA instances connected.
CPU4: CPU counterpart of pGA_4^1

Third, it is not clear which particular configuration achieves the best result. This is a common observation in GA systems. Practically, this observation suggests the need for flexibility of a GA system, because inflexible systems may prevent the users from fine-tuning them. We pay special attention to the flexibility in our framework, allowing users to specify various properties of the system to pursue better solutions.

4.2.2 Benchmarks

To test the ability of dealing with numeric computation in our pGA systems, we solve four GA benchmarks with binary and real-valued encoding: BF6, 2DS, F7 and F11 mentioned in section 3.5 [FZS10, Col03]. The resource usages and speedups of different problems are shown in Table 4.6, when the frequencies of them are between 120 MHz and 160 MHz. Based on the resources left over, we can tailor the architectures according to the complexity of an evaluator, by changing parameters such as N_E.

Table 4.6: Resources Usage (Res.) and Speedups of pGA systems.
Speedups: SP.C: Speedup over CPU-based pGA, SP.F: Speedup over single kernel GA in chapter 3. N_k: Number of GA kernels

App.	N_k	N_E	N_S	Res.%	SP.C	SP.F	Chromosome
MAXSAT	4	2	2	41.44	24	1.5	binary
BF6	4	2	2	23.74	28	1.4	binary
2DS	4	2	2	60.42	34	1.4	binary
F7	4	2	2	27.95	21	1.2	real-valued
F11	4	2	2	55.34	23	1.3	real-valued
Geometric Mean		-	-	-	26	1.4	

4.2.3 Discussion

As a variant of genetic algorithms, parallel GAs have great potential for hardware acceleration. To simplify the design and execution flow of FPGA-based pGA systems for non-expert users, this section extends the automated framework in chapter 3 to generate them based on a high-level description of a target problem. The flexible architecture created by our framework exploits two levels of parallelism, allowing users to customise the degrees of both external and internal parallelism, as well as the topology of migration. Once created, the framework also enables a user to tune run-time parameters without time-consuming recompilation, including migration parameters. Our system is more flexible compared with existing pGA work; when compared with a multi-core software implementation, our system also shows an average speedup of 26 times for an NP-hard problem and four benchmarks; our system is also 6.8 times faster than a GPU for the NP-hard problem. We also compare the proposed work with the single kernel GA system in chapter 3, showing an average speedup of 1.4 times. It also shows a reduction in resources compared with the single kernel, probably because of distributed memory and less fan out.

4.3 Inter-FPGA Parallelism: pGA on Multiple FPGAs

In the last section we have shown that parallel GAs can solve optimisation problems quickly with a number of parallel GA instances, but the hardware resources in one FPGA are limited, so the maximal number of GA kernels is also limited. Researchers have previously proposed

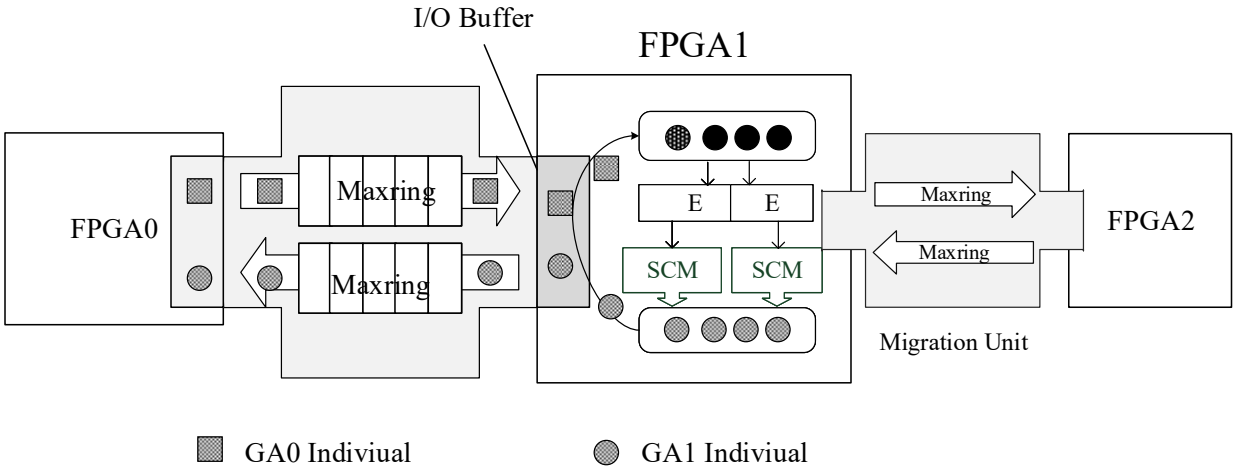


Figure 4.5: Intra-FPGA Migration Unit and Migration Between Three FPGAs. I/O: Input/Output, Maxring: a link between FPGA chips, E: Evaluator, SCM: Selection, Crossover and Mutation

implementing parallel GAs using multiple FPGAs to solve this problem, as mentioned in section 2.4.2. However, the lack of flexibility of previous systems limits its usage to non-expert users, and they do not support changing the migration policy at run-time [NS05, YY99]. Examples of previous systems are mentioned in section 2.4.2.

When designing a parallel GA system on multiple FPGAs, we need to consider the communication challenges, as working with multiple FPGAs is more difficult than working with multi-core CPUs or multiple instances in one FPGA. To deal with this complexity, a simple method is needed to abstract the hardware details.

This section proposes an inter-FPGA migration unit, which controls all migration between connected FPGAs. Compared with previous work, our approach significantly improves its pipeline performance. It supports simple communication such as chain connection between multiple FPGAs, encapsulating the complex operation of connecting and synchronising multiple FPGAs together.

Table 4.7: Qualitative Comparisons of Multiple FPGA-based pGA Systems. E: evaluator, SCM: Selection, Crossover and Mutation.

Work	Year	# of FPGAs	Runtime Param.	Migration Param.	Parallel SCM/E	Platform
[SWSS95]	1995	3	N/A	N/A	No	Armstrong III
[GN95]	1995	4	N/A	N/A	No	Splash 2
[YY99]	1999	Multiple	N/A	No	No	SFL
[NS05]	2005	Multiple	N/A	No	No	Spartan II-E
Our Work		Multiple	SCM rates	Run-time	Yes	Virtex 6 SX475T

4.3.1 Inter-FPGA Migration Unit

The inter-FPGA migration unit contains an inter-FPGA link buffer, input buffer and output buffer. The link buffer is implemented as Maxring in Maxeler system. The inter-FPGA link buffer has a bigger delay than the intra-FPGA buffer in section 4.1. For example, the cycles needed for an individual to be sent from one GA instance to another instance in one FPGA are around 70, while the cycles needed in inter-FPGA systems are 150 in the Maxeler system.

Similar to the internal migration unit in section 4.1, the inter-FPGA migration unit also has the same migration parameters, including the migration interval and number of migrants. Compared with the previous multiple FPGA-based GA systems, a user can customise all of those parameters in the inter-FPGA migration unit.

The inter-FPGA migration uses non-blocking transfer, as the same as the communication between Generic GA instances in one FPGA. As shown in Figure 4.5, there are three FPGA chips: FPGA1, FPGA2 and FPGA3. FPGA1 puts the data in the output buffer, and then keep sending the data to their neighbours via Maxring. When receiving new individuals from FPGA0, FPGA1 also places it into its local memory. Due to the limitation of Maxring, we allow the migration topology between FPGAs to be configured as chain structure.

4.3.2 Qualitative Summary

We summarise the features of our multiple-FPGA approach and compare it with other pGA approaches mentioned in section 2.4.2. From the comparison shown in Table 4.7, it is clear

Table 4.8: Resource Usage in Virtex-6 SX475T for LP, TSP and F11 Problems

Problems.	LUTs(%)	FFs(%)	BRAMs(%)	DSPs(%)
TSP	75.85	59.69	19.17	6.25
Locating	73.08	40.71	9.68	26.98
F11	60.05	38.14	17.67	32.54

that our approach is different from others, because we address limitations encountered in the current research:

1. Our approach maximises flexibility and parallelism in the architecture, including the migration topology, the degrees of both inter-FPGAs and internal SCM/Evaluator parallelism.

2. Our approach supports a simple method for the inter-FPGA link, which abstracts the complexity and the synchronisation between multiple FPGAs.

3. Our approach allows users to tune the inter-FPGA migration interval at run-time, which provides an easy and fast way to test different configurations.

4.4 Performance Evaluation

We implement the proposed solution on a workstation with four Maxeler MAX3A Vectis acceleration cards. Each card contains a Xilinx Virtex-6 SX475T FPGA. The CPU-based GA system from GALib is optimised in the same way with the section 4.2.

In our testing we choose different kinds of problems: F11, the Locating Problem and the Travelling Salesman Problem (TSP). Using multiple FPGAs makes it possible to fit a big problem which cannot be parallelised in one FPGA. As shown in Figure 4.8, TSP has a big kernel and already takes more than 70% shown in Table 3.5, cannot be parallelised in one FPGA. Based on the resources left over, we can tailor the architectures according to the complexity of our evaluation and SCM/Evaluator unit.

We have obtained good speedup for the problems we have tested our approach on (see Table

Table 4.9: Experimental Results and Speedups of pGAs on Multiple FPGAs. P: SCM/Evaluator Parallelism, Freq.: Frequency, SP.C: Speedup over CPU-based GA, SP.F: Speedup over framework with a single FPGA

Fun.	No. of FPGA	P. per FPGA	Freq.(MHz)	SP.C	SP.F
TSP	4	1	75	30	1.4
Locating	4	32	150	27	1.5
F11	4	4	150	34	1.5
Geometric Mean				30	1.5

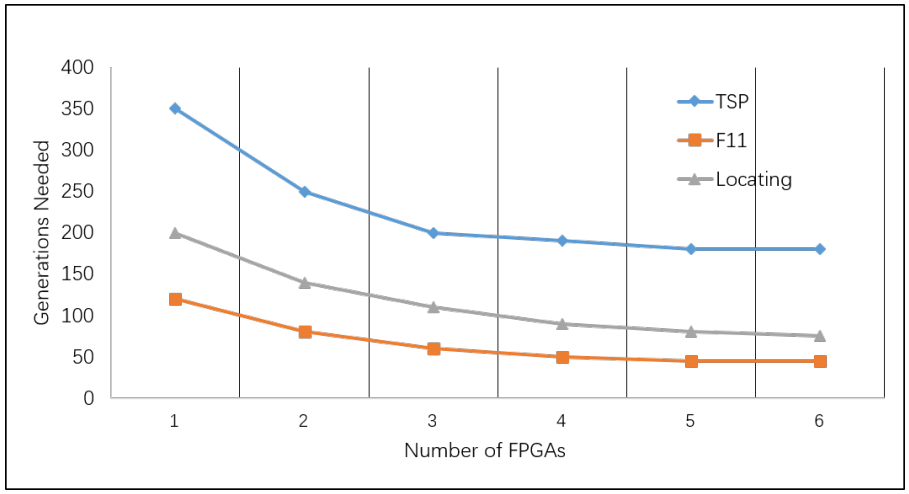


Figure 4.6: Generations Needed to Reach Desirable Solutions with Different Numbers of FPGAs

4.9). Our tool performs the best in the case of the F11 benchmark, obtaining a 34 times speedup, while using a 150 MHz clock frequency, when compared to an equivalent multi-core software implementation of the GA. We also compare the performance with the framework using one FPGA in section 3, the parallel one showing a 1.5 times speedup.

We also change the number of FPGAs for the three problems to see how the number of parallel FPGAs affects performance. From Figure 4.6, we could find increasing the degree of parallelism will raise performance to find the desirable solutions (90% of the optima), although the speedup may plateau.

In summary, we have tested our system on a location problem, a GA benchmark called F11, as well as the well-known travelling salesman problem. After running experiments on these problems, we obtain 30 times speedup on average with a 4-FPGA system compared to a multi-core software implementation and 1.5 times speedup over a single kernel FPGA-based GA.

4.5 Migration Interval and Discussion

In this section, we examine how the migration interval affects performance, and give users some guidance on setting it. We have done experiments for two problems and their results are shown in Figure 4.7. The first three graphs are the F11 under different migration intervals, while the last three are for the locating problem (LP).

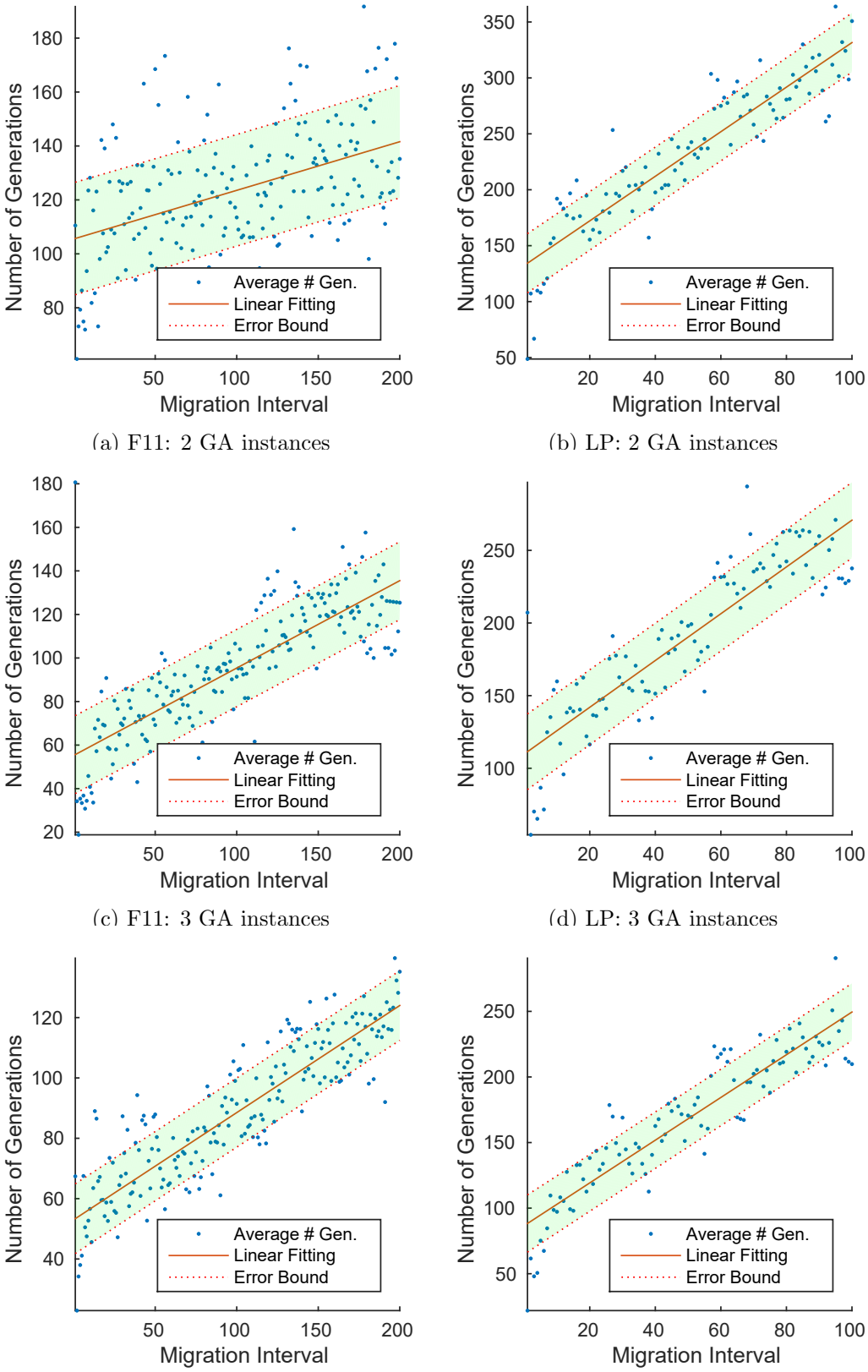
In Figure 4.7, each point represents the average number of generations to reach 90% of the best possible fitness in 100 repetitions using the corresponding migration interval. The straight line is the linear regression. This is to show the trend of the growth. The error bound is the regression result represented by one standard deviation of the regression error, showing the uncertainty of the linear regression.

Thus, as a result of our extensive experiments we can draw two major conclusions as follows:

- A system with more GA instances takes fewer generations to obtain satisfactory fitness. One explanation of this observation is that having more GA instances enables more individuals to evolve in parallel.

- The number of generations to reach the chosen fitness level grows linearly against the migration interval. We also notice that, even if we increase the degree of the regression polynomial, we still obtain a linear trend. This trend reveals the strong correlation between the migration interval and the optimisation quality, but there does not seem to be any obvious reasons for the linearity of the correlation.

The results provide practical guidance for the determination of the migration interval. The experiments suggest that the lowest achievable values of migration interval are likely to result in the highest performance. As a result, we suggest users give high priority to small values in the determination of migration intervals and high parallelism.



(a) F11: 2 GA instances

(b) LP: 2 GA instances

(c) F11: 3 GA instances

(d) LP: 3 GA instances

(e) F11: 4 GA instances

(f) LP: 4 GA instances

Figure 4.7: Generation Needed for Different Migration Intervals and Number of GA Instances. The result shows the migration interval affects the generations needed to reach the desirable results for F11 ((a), (c) and (e)) and LP ((b), (d) and (f))

4.6 Summary

Parallel GAs have significant potential for hardware acceleration in evolutionary computing, as demonstrated by previous work and this study. The parallel GAs can start from different search sub-spaces and exchange good solutions with each other, allowing them to cooperate to find the best solution. pGA not only reduces the execution time of finding a good solution, but also reduces resource usage because it reduces fan-out by using distributed memory.

This chapter extends a GA system which makes use of multiple GA kernels on one or multiple FPGAs to speed up the search for optimised solutions. Our system is simple to use, providing increased flexibility not only in picking GA components and parameters, but also in selecting the migration parameters. Our approach exploits two levels of parallelism: internal parallelism in SCM and evaluator units, and external parallelism in one FPGA and multiple FPGAs. It also enables the tuning of run-time parameters without time-consuming hardware recompilation.

However, the work in this chapter and previous chapter are all based on the classical genetic algorithms, using centralised memory to store all the individuals and with a close data dependency when updating each generation. In their pipeline structure, there exist pipeline stalls in many places, for example, GAs has to wait until all individuals update before entering the next generation. Even using steady-state GA, the pipeline stall still exists. The next chapter proposes a totally new strategy based on current GAs, but designed specifically for reconfigurable hardware. It makes full use of reconfiguration and introduces graph theory into the traditional GA, which solves the pipeline stalls effectively.

Chapter 5

Pipelined Genetic Propagation

FPGA acceleration is a promising way to speed up genetic algorithms, but the state-of-art FPGA accelerations of genetic algorithms, including the work in chapters 3 and 4, are limited to the implementation of standard or conventional genetic algorithms. Standard genetic algorithms contain a considerable number of operations without efficient hardware mappings, so the problems behind the current FPGA-based genetic algorithms are:

- Insufficient way to escape local optima. Genetic algorithms sometimes get stuck in local optima, but the traditional GA relies on only the genetic operations to escape from local optimum. In addition, when genetic operations cannot produce any diversity, changing GAs' operators needs time-consuming recompilation.

- Large centralised memory. Standard genetic algorithms use a centralised memory to store all individuals. Even in the parallel GAs, a GA instance still uses centralised memory. However, accessing the centralised memory results in large fan out and limited memory bandwidth, which is not time effective for FPGAs.

- Pipeline stall. Generic genetic algorithms have a strong data dependency between generations, due to the selection-crossover-mutation-updating link. The evolution process is performed generation by generation, which results in pipeline stalls because the system has to wait for new individuals to arrive before evolving next generation.

This chapter proposes a variant of the standard evolutionary algorithm which solves these problems in reconfigurable hardware. We arrange this chapter as:

- Section 5.1 summarises the motivation and insight when designing PGP.
- Section 5.2 describes the architecture of Pipelined Genetic Propagation (PGP), including genetic operators, links between genetic operators, and physical implementation of these links.
- Section 5.3 proposes a novel way to escape from local optima, using run-time topology switch, to reconnect the genetic operators.
- Section 5.4 presents a topology generation algorithm, which can create efficient topologies for chosen set of genetic operators.
- Section 5.5 describes the high level inputs, which allows the users to quickly configure the PGP architecture.
- Section 5.6 shows PGP can be used to solve several optimisation problems, and has better performance than previous work.

5.1 Motivation and Insight

We propose a novel strategy based on these observations from the previous chapters:

- Splitting large memories into small pieces increases memory bandwidth and decreases memory resources because of the reduced fan-out, this observation comes from the parallel GA in chapter 4.
- High parallelism and a small migration interval help to improve performance, this observation comes from the experiments in section 4.5. That means we need the GA instance to communicate with its neighbour as frequently as possible.

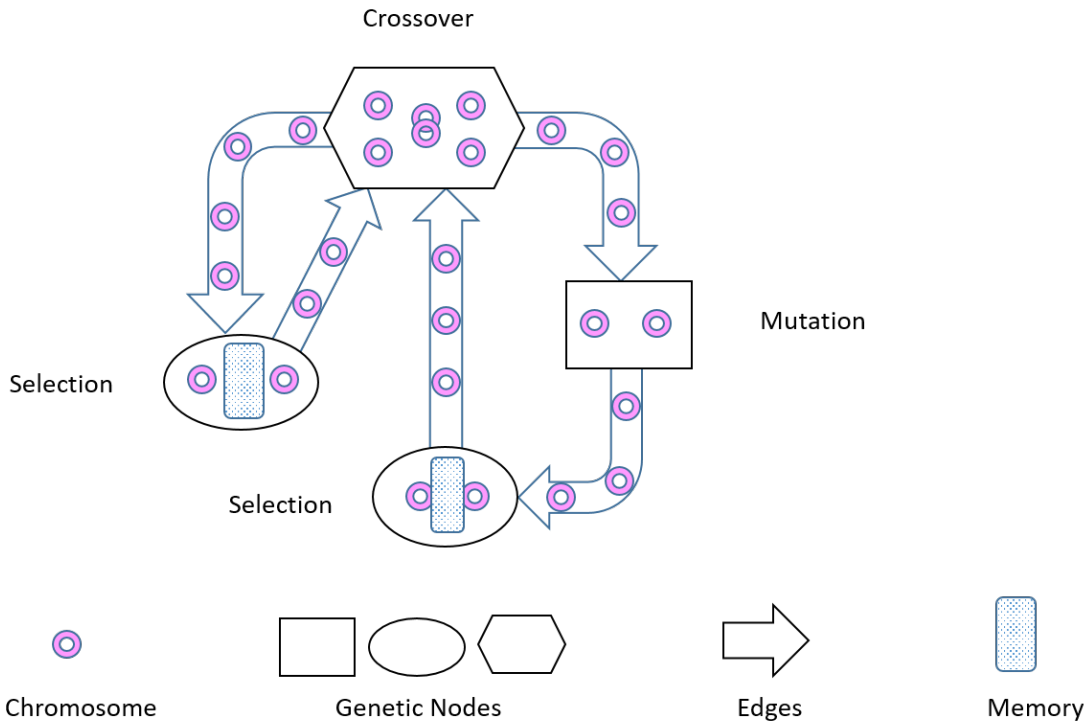


Figure 5.1: Pipelined Genetic Propagation Architecture

- Fully utilised pipelines could greatly improve performance, according to chapter 2. However, while it is possible to reduce the pipeline stall in a generational GA, it is impossible to fully eliminate the stalls. So we must redesign the evolutionary process.

- Changing the topology in the link of parallel GA affects the evolutionary process. This can be seen from section 4.2. For example, pGA_2^2 works differently to pGA_1^4. So we should consider the power of the connection of genetic nodes.

Based on these observations, we propose a novel strategy which rethinks the design of genetic algorithms, which is described in following sections.

5.2 PGP Architecture

Pipelined Genetic Propagation is an optimisation algorithm, architecture, and strategy, allowing users to create a hardware-specific graph-based GA optimiser for a chosen objective function. A PGP system optimises the objective function by propagating and circulating a group of

individuals in a directed graph structure $\mathbb{G} = (\mathbb{V}, \mathbb{E})$, where $\mathbb{V}$ is a set of genetic nodes that perform genetic operations on data, and $\mathbb{E}$ is a set of directed edges between the nodes that transport data in a pipelined or streaming manner, as shown in Figure 5.1. All nodes and edges are continually active, with no pauses or dependencies, making full utilisation of the logic.

5.2.1 Genetic Nodes

Genetic nodes are the elementary compute components in a PGP solver, and are represented by the nodes $\mathbb{V}$ in the logical structure. A primary concern in the design of the nodes is an efficient hardware implementation, so we propose the following requirements:

1. Minimise memory consumption. On-chip RAM is limited in a reconfigurable hardware platform, so we try to eliminate explicit RAM storage and rely on implicit storage in pipelines, only using on-chip memory when random accesses are needed.

2. Distributed local memories. The large centralised storage in generational GAs results in high contention and significant fan-out. We only allow local RAMs which are accessible within a genetic operator, ensuring locality and exploiting the distributed nature of FPGA storage.

3. Minimising per-node resource consumption. To get the maximum speed-up, we want as many genetic operators as possible, so we design small operators that map easily to hardware, allowing us to scale up to complex solver topologies and large FPGAs.

Based on these design disciplines, we map all the data-dependent genetic operators of genetic algorithms into hardware as distributed nodes, which are shown in Figure 5.2. We divide the genetic nodes into three classes, namely selection nodes, crossover nodes and mutation nodes. The exact behaviour and implementation of these nodes (e.g. the specific version of selection) is not specified at this level, only the overall behaviour in terms of inputs and outputs.

A **selection node** consists of an input port, an output port, some amount of local memory, a fitness evaluator, and an unspecified selection process. The local memory stores a fixed number

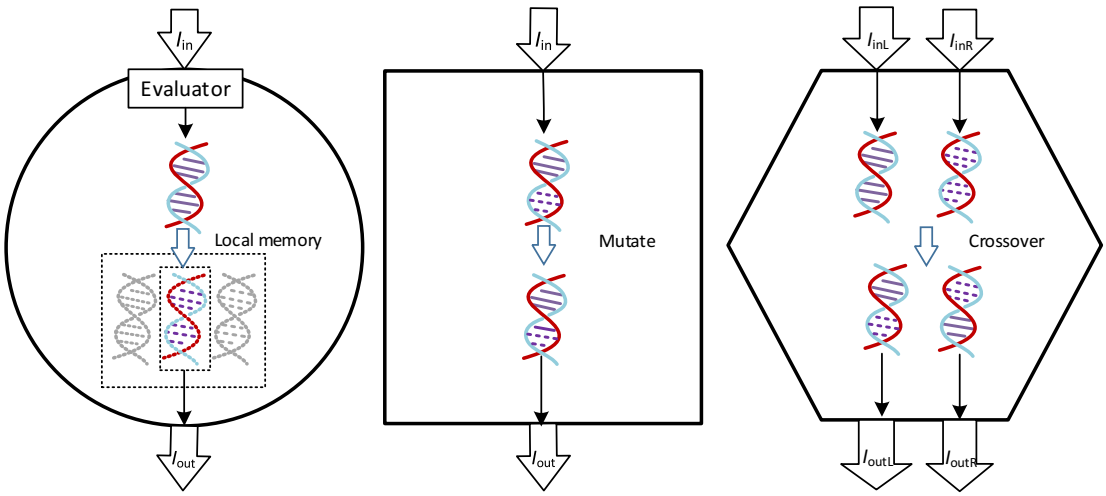


Figure 5.2: Genetic Nodes: Selection, Mutation and Crossover.
I_{in}: incoming individual, I_{out}: output individual, L: left, R: right.

N_i of individuals along with their corresponding fitness values. Every cycle, the input port receives one incoming individual I_{in} and sends it to the evaluator, while the output port sends out one individual I_{out} chosen by the selector from the local internal memory. When the fitness value of the incoming individual I_{in} becomes available, the node compares the fitness values of I_{in} and I_{out}. If the incoming individual I_{in} has higher quality than the outgoing individual I_{out}, it replaces I_{out} in the local memory. Meanwhile, its fitness value replaces that of the selected individual as well. In other words, the selection node monitors the quality of the individuals evolving and stores the high-quality individuals in the local memory.

A **mutation node** consists of an input port, an output port and some sort of mutation process, but does not contain any local memory or fitness evaluator. Every cycle, the input port receives one incoming individual I_{in}, then mutates the individual (for example using a bit-flip or constraint mutation). Finally, the output port sends out a mutated individual to other nodes.

A **crossover node** consists of two inputs, two output ports and some sort of crossover process. Similar to the mutation node, a crossover node contains neither local memory nor fitness evaluator. The crossover node exchanges the information between the incoming individual I_{inL} from one input port and I_{inR} from the other port. After receiving the two individuals, the crossover operator cuts off a piece of information from each individual, and recombines the individuals by transplanting the information taken from one individual to the other, producing two new

individuals I_{outL} and I_{outR}. The two output ports then send out I_{outL} and I_{outR} to other nodes. Note that we disallow any output of a crossover node to directly connect to its inputs, because a crossover node configured in this way might introduce a combinatorial loop.

We divide the genetic nodes into two categories, namely non-blind nodes and blind nodes. As the selection nodes have the evaluators to monitor the quality of individuals, we call them *non-blind* nodes; the mutation and crossover nodes change the individuals without considering their quality, so we call them *blind* nodes. For ease of discussion, we denote the set of all selection, mutation and crossover nodes respectively as $\mathfrak{S}, \mathfrak{M}$, and $\mathfrak{C}$. Let the set of blind nodes be $\mathfrak{B} = \mathfrak{M} \cup \mathfrak{C}$ and the set of all non-blind nodes be $\mathfrak{N} = \mathfrak{S}$. An evolutionary process should have at least one blind node and one non-blind node, because blind nodes carry out the evolutionary genetic operations which increases diversity, while the non-blind nodes ensure that there is a trend towards higher fitness individuals.

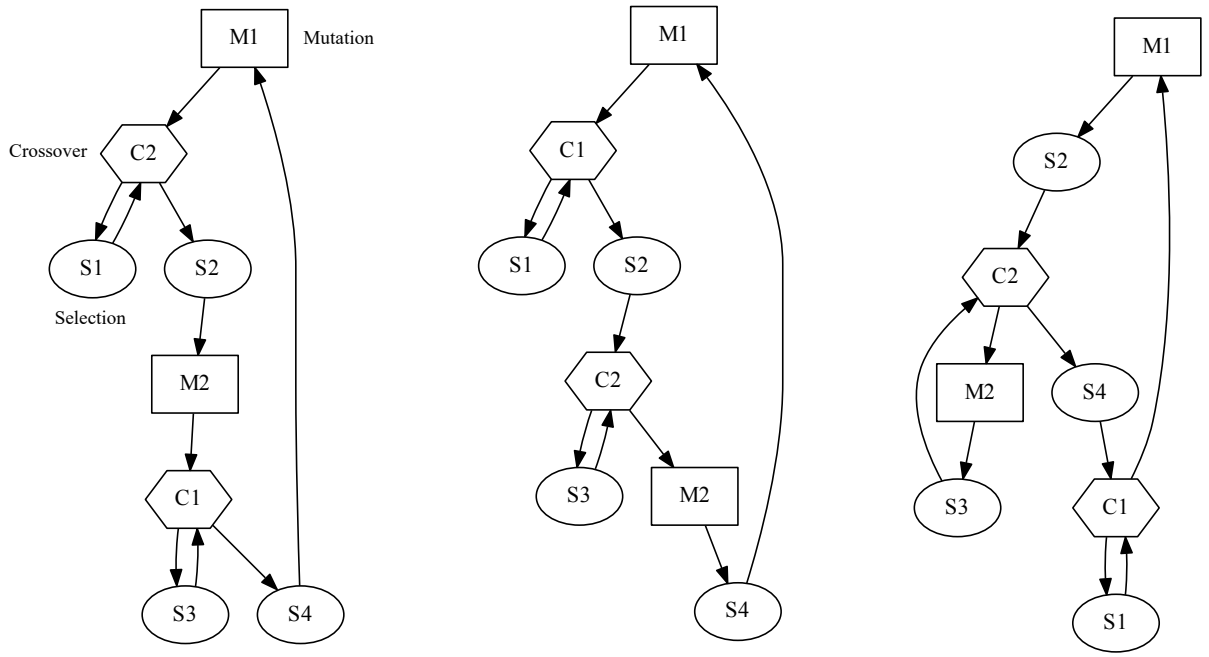
5.2.2 Logical Connection Between Nodes

We will now consider the logical connections between the nodes, which is captured as the set of edges $\mathbb{E}$ between nodes. In the design of logical connections, we keep the following considerations in mind:

1. Minimise the amount of recompilation. Hardware compilation of FPGA devices is time-consuming, and it is not always clear what the best topology is, so we support a single compilation of the architecture which allows multiple possible topologies.

2. Dynamic changes of connection topology at run-time. Run-time reconfiguration of the topological structure can enhance the optimisation power of the system, but we do not want to interrupt the current evolutionary process. During the reconfiguration of the topological structure, we keep the data in all unrelated blocks unchanged, retaining the fitness already achieved.

3. Configurable set of selection, mutation, and crossover nodes. Users can specify a set of nodes to use, and may also choose to use a sub-set in certain configurations. PGP will



(a) Topology 1 (b) Topology 2 (c) Topology 3

Figure 5.3: Three Topologies: T_1, T_2, and T_3.
They all contain two mutation nodes (M1 and M2), two crossover nodes (C1 and C2), and four selection nodes (S1, S2, S3 and S4)

then create valid topologies using all the user-specified nodes and allows exploration of different patterns of connectivity between them.

Compared with the fixed and restricted connection of genetic operators in generational genetic algorithms, the proposed connection scheme is much more flexible and powerful. Given the number and type of genetic nodes, we can create many different topologies for it. For example, given 2 mutation nodes, 4 selection nodes and 2 crossover nodes, we have 10 inputs and 10 outputs in total, a large set of possible interconnections.

However, only a subset of topologies are reasonable, and can be expected to produce good results. According to experience with genetic algorithms, we suggest to restrict the set of topologies using the following rules:

- **Rule of full connectivity**: for all pairs of nodes $(v_\dagger, v_\ddagger) \in \mathbb{V}^2$ in a topology, there exists a path from $v_\dagger$ to $v_\ddagger$ with finite length, as shown in Figure 5.4.

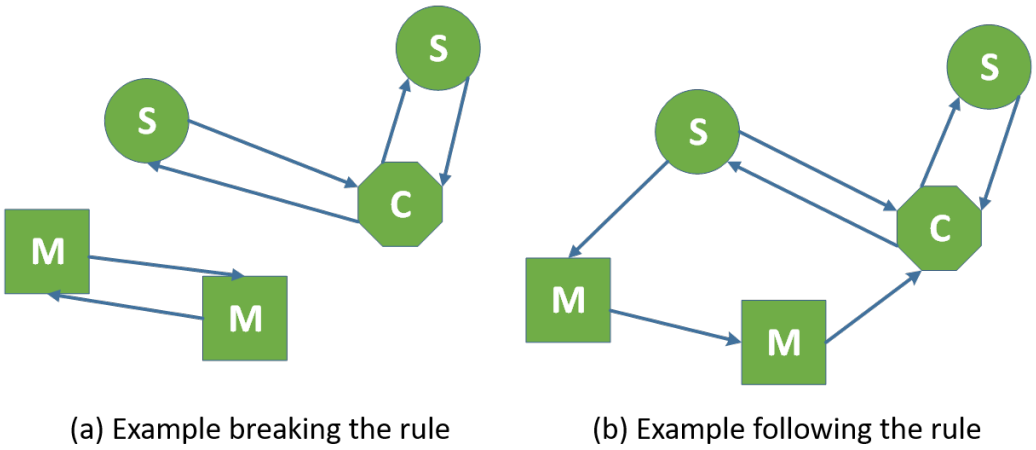


Figure 5.4: Examples for Rule of Full Connectivity.
Left diagram breaks the rule while the right obeys the rule

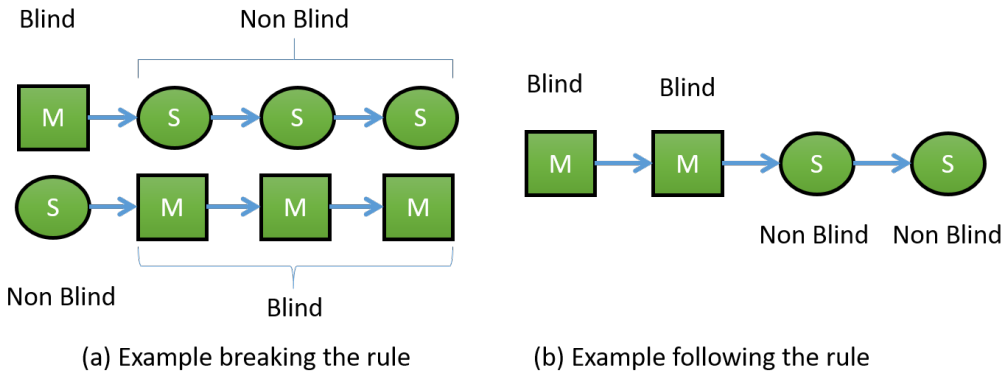


Figure 5.5: Examples for Rule of Striped Serialisation.
Left diagram breaks the rule while the right obeys the rule

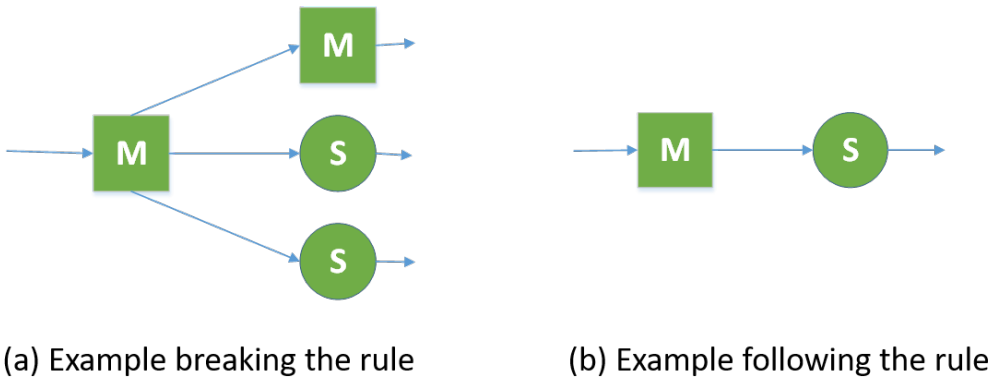


Figure 5.6: Examples for Rule of Fan-out Control.
Left diagram breaks the rule while the right obeys the rule

- **Rule of striped serialisation**: for all edges $\langle v, v' \rangle \in \mathbb{E}$, if $(v, v') \in \mathfrak{B}^2$, then $v'' \in \mathfrak{N}$ for all v'' such that $\langle v', v'' \rangle \in \mathbb{E}$; for all edges $\langle v, v' \rangle \in \mathbb{E}$, if $(v, v') \in \mathfrak{N}^2$, then $v'' \in \mathfrak{B}$ for all v'' such that $\langle v', v'' \rangle \in \mathbb{E}$, as shown in Figure 5.5.

- **Rule of fan-out control**: $\deg^+(v_i) = 1$ for all nodes $v_i \in \mathfrak{M} \cup \mathfrak{S}$; $\deg^+(v_{ii}) = 2$ for all nodes $v_{ii} \in \mathfrak{C}$ where $\deg^+(v)$ is the outdegree of node v, as shown in Figure 5.6.

These rules are based on experience and experiments: the rule of full connectivity ensures all individuals are able (though not guaranteed) to eventually reach all nodes in the topology. The rule of striped serialisation ensures at most two nodes of the same type connect directly. In particular, a chain of blind nodes may throw out or replace high-quality solutions because the blind nodes do not monitor the fitness. Similarly, a chain of non-blind nodes keep the individuals unchanged and results in extra resource consumption. The rule of fan-out control avoids duplication of individuals to retain the divergence of the population, because experiments show the duplication of individuals reduces the diversity in population. The experiment results in Figure 5.15 show the stable quality of PGP generated by the three rules. Users can propose new rules according to their own considerations.

It is a non-trivial task to generate legitimate topologies satisfying all the proposed rules, particularly with large numbers of nodes. We tackle the topology generation problem and present our solution in section 5.4. In the rest of this section, we assume the availability of legitimate topologies. For instance, Figure 5.3 shows three topologies T_1, T_2 and T_3 satisfying all the three rules. In each topology, $\mathbb{V}$ includes two mutation nodes (M1 and M2), two crossover nodes (C1 and C2), and four selection nodes (S1, S2, S3 and S4).

5.2.3 Physical Connection Between Nodes

The pipelined genetic propagation scheme has a substantial advantage over generational genetic computing architectures in terms of pipeline efficiency. In a generational system, the next generation cannot be started until the previous one has finished, forcing a synchronisation point

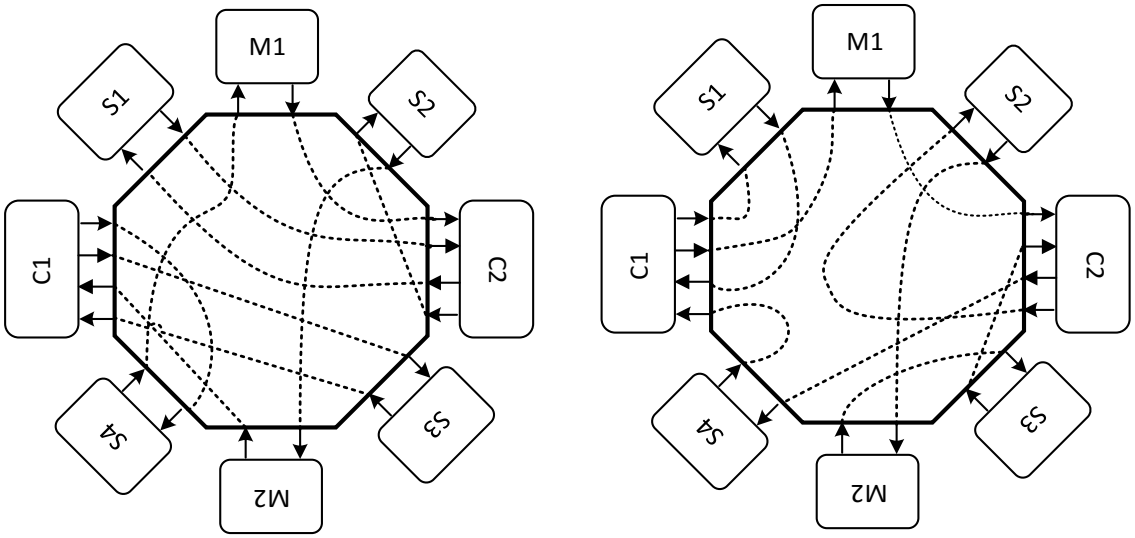


Figure 5.7: Physical-Level Connections between Genetic Nodes for T_1 and T_3 from Figure 5.3

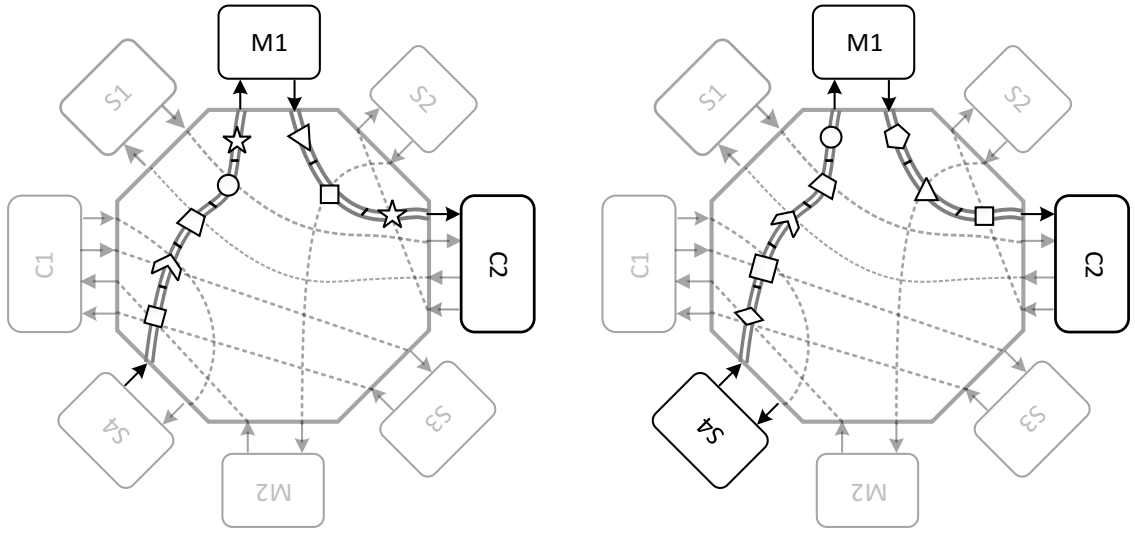


Figure 5.8: Pipeline Flow in the Physical-level Connections for T_1 from Figure 5.3

amongst all genetic operators. This data dependence results in pipelines stalls and reduces the overall efficiency of computation.

In contrast, our proposed system does not divide the evolution process into generations. Therefore, all genetic operations are able to work simultaneously in a fully pipelined manner without incurring stalls between generations. For instance, the physical connections between the genetic nodes of the two example topologies T_1 and T_3 in Figure 5.3 are shown in Figure 5.7. The two stages of the pipelined flow of first topology T_1 are demonstrated in Figure 5.8, showing the individuals flow in the pipeline as streams.

To allow the propagation and circulation of individuals in the topology, we need to map the logical structure to a physical reconfigurable device, which we solve using a *topology mapping unit* to manage the connection of genetic nodes. The mapping unit connects all the inputs and outputs of genetic nodes, using multiplexers to support dynamic reconfiguration of topology without interrupting executing. A direct and simple solution is to connect all the inputs from each genetic node to all genetic node outputs, using a *full crossbar*.

A full crossbar supports a wide range of topologies, however it is resource-consuming, especially for a large number of genetic nodes. It also supports many illegal connections of nodes according to the rules, as well as multiple realisations of what is essentially the same topology. To reduce resource consumption, we limit the inputs of each multiplexer to support only the topologies needed, which is a sub-set of all valid topologies. If a user generates M topologies for a set of genetic nodes with N Input sources, we only need at most M input sources for each multiplexer, which saves $N - M$ input sources for each multiplexer. We call this method *sparse crossbar*, which uses less resource usage than full crossbar. Some of the topologies may share a subset of inputs, the number of required inputs for a multiplexer may be even less than M, thus saves even more resource. We will discuss the empirical benefits of sparse crossbars over full crossbars in section 5.6.

5.3 Dynamic Topology Change

Generational genetic algorithms sometimes get stuck in a local optimum, or a particular configuration is found to work poorly for a specific problem. In software the algorithm and parameters can be tweaked, but in hardware any modification of the architecture will require a time-consuming recompilation. In the work in chapter 3, we can modify the crossover and mutation rates. In PGP, we also can reconnect the genetic nodes, in order to escape from the local optimum found by one topology. For instance, Figure 5.9 shows a case where we support three topologies for a problem, each of which contains two genetic operators. In this example, we assume that the two genetic operators are conventional and smooth, such that one execution of the operator slightly modifies the individual. A point in the space represents an individual, and the x axis and y axis show respectively the directions that the two genetic operators drive individuals along. The z axis stands for the fitness value of the individual. All points in the space form a surface corresponding to the three-dimensional projection of the high-dimensional individual fitness mapping. The surfaces in the three topologies are different because genetic operations in the three topologies move individuals on different planes in the high-dimensional space.

The genetic operations modify an individual slightly in each execution, resulting in small changes in the solution space. The small scale of changes brings both benefits and drawbacks. The small steps enable an individual to move smoothly in the solution space of a topology without mistakenly skipping high-quality solutions. Nevertheless, when an individual gets stuck in a local optimum, the genetic operators may have insufficient power to help the individual to escape due to the limited step size.

An occurrence of dynamic topology change corresponds to an *inter-topology crossing* for all individuals. If an individual is unable to move in the original topology due to a local optimum, it may move again in the destination topology. By moving back and forth between topologies, an individual has a better chance to escape from a local optimum.

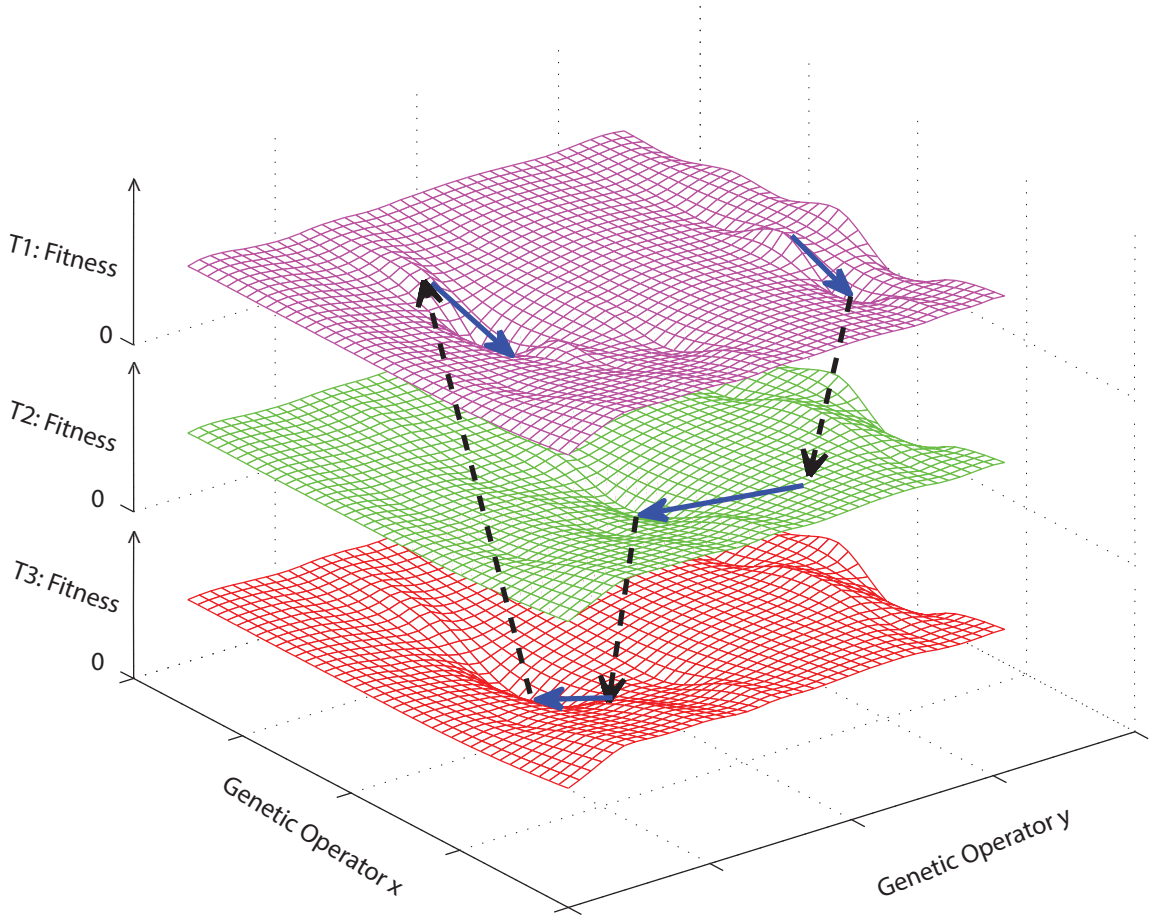


Figure 5.9: Inter-Topology Crossing between Three Topologies.
The x axis and y axis show respectively the directions that the genetic operators drive individuals along. The z axis stands for the fitness surface.

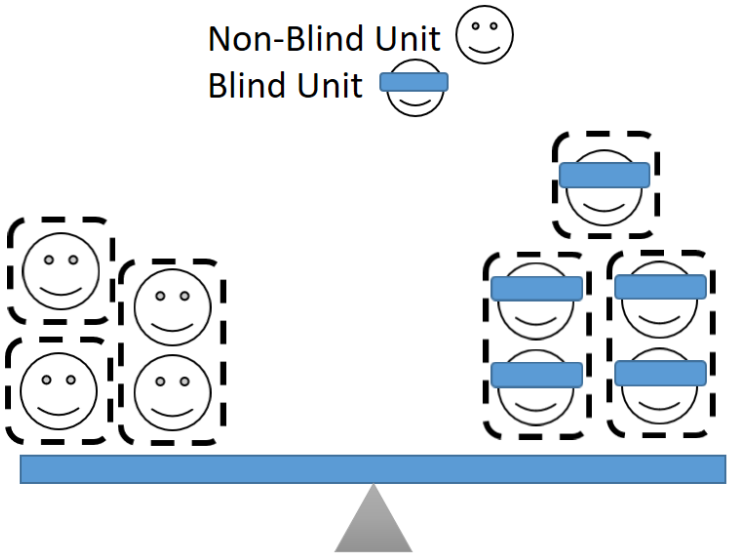


Figure 5.10: Step 1: Balancing.
The step balances the number of blind nodes and non-blind nodes. Blind nodes: crossover and mutation. Non-blind nodes: selection

5.4 Topology Generation Algorithm

We propose a randomised topology generation algorithm to produce legal topological structures for the connection problem based on the three rules in section 5.2.2. This algorithm consists of four major parts: *balancing*, *threading*, *dispatch* and *collapsing*.

5.4.1 Balancing

The balancing algorithm discovers legitimate configurations for a chosen number of blind and non-blind nodes by finding integer solutions to a set of constraints. According to Rule of striped serialisation, let x_b and x_{2b} be the number of blind nodes and blind pairs respectively; let x_{nb} and x_{2nb} be the number of non-blind nodes and non-blind pairs respectively. By definition, the following must hold:

$$x_b + 2x_{2b} = 2c + m \tag{5.1}$$

$$x_{nb} + 2x_{2nb} = s \tag{5.2}$$

Here c is the number of crossover nodes, m is the number of mutation nodes, and s is the number of selection nodes. Because blind components and non-blind component alternate in the chain, the number of blind components equals that of the non-blind ones, such that:

$$x_b + x_{2b} = x_{nb} + x_{2nb} \tag{5.3}$$

Therefore, the vector $\mathbf{x} = (x_b, x_{2b}, x_{nb}, x_{2nb})'$ must satisfy

$$\begin{pmatrix} 1 & 2 & 0 & 0 \\ 0 & 0 & 1 & 2 \\ 1 & 1 & -1 & -1 \end{pmatrix} (x_b, x_{2b}, x_{nb}, x_{2nb})“ = \begin{pmatrix} 2c+m \\ s \\ 0 \end{pmatrix} \tag{5.4}$$

The rank of the coefficient matrix is less than the number of its rows. Therefore, the linear equation has an infinite number of solutions if there are no additional constraints. However, we know that x_b, x_{2b}, x_{nb} and x_{2nb} must be all positive integers. To obtain the positive solutions, we first represent x_b, x_{nb} and x_{2nb} in terms of x_{2b}:

$$x_b = 2c + m - 2x_{2b} \tag{5.5}$$

$$x_{nb} = 4c + 2m - 2x_{2b} - s \tag{5.6}$$

$$x_{2nb} = s - 2c - m + x_{2b} \tag{5.7}$$

We then determine x_{2b} by enumerating through a number of integers in the range $[x_\perp, x_\top]$ where

$$x_\perp = \lceil 2c + m - s \rceil \tag{5.8}$$

$$x_\top = \left\lfloor \min\left(\frac{m}{2} + c, m, m + 2c - \frac{s}{2}\right) \right\rfloor \tag{5.9}$$

5.4.2 Threading

The threading program arranges the units along a directed ring, as shown in Figure 5.11. The threading algorithm first makes a local arrangement for blind components and non-blind ones

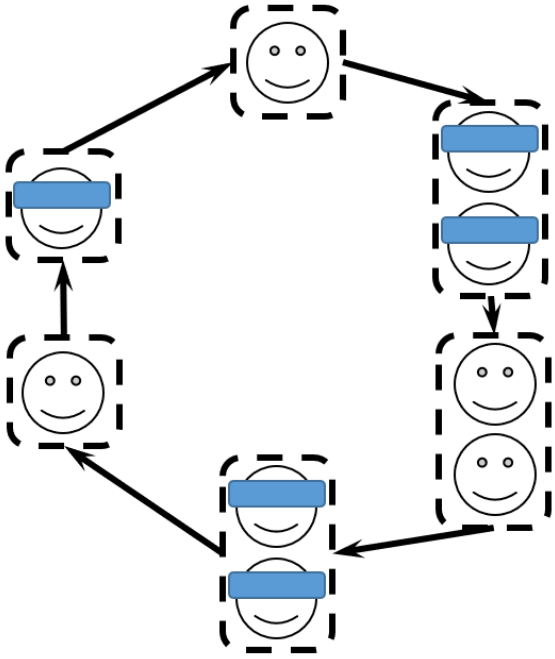


Figure 5.11: Step 2: Threading.
The threading program arranges the units along a directed ring

separately, and then merges the two sequences together. To make a local arrangement for blind components, the algorithm creates a sequence of blind components where the number of blind units and pairs follow the constraints from the previous stage. To merge the two local arrangements together, the algorithm takes entries alternately from the two local arrangements.

The algorithm first produces a sequence list of blind nodes L^B and a sequence list of non-blind ones L^N. Let $\mathcal{B}$ be an abstract blind node, which will be determined to be a crossover node or a mutation node in later stages of the generation algorithm; let $\mathcal{N}$ be a non-blind node, which in this work is always a selection node. For ease of explanation, we use $2\mathcal{B}$ and $2\mathcal{N}$ to represent respectively a combination of two blind nodes, and a combination of two non-blind nodes.

$$L^B = (\underbrace{\mathcal{B}, \mathcal{B}, \ldots, \mathcal{B}}_{x_b \text{ times}}, \underbrace{2\mathcal{B}, 2\mathcal{B}, \ldots, 2\mathcal{B}}_{x_{2b} \text{ times}}) \tag{5.10}$$

$$L^N = (\underbrace{\mathcal{N}, \mathcal{N}, \ldots, \mathcal{N}}_{x_{nb} \text{ times}}, \underbrace{2\mathcal{N}, 2\mathcal{N}, \ldots, 2\mathcal{N}}_{x_{2nb} \text{ times}}) \tag{5.11}$$

The algorithm then shuffles L^B and L^N into a random order called Λ^B and Λ^N.

Finally, the algorithm returns a sequence Λ containing all the entries in Λ^B and Λ^N. Entries

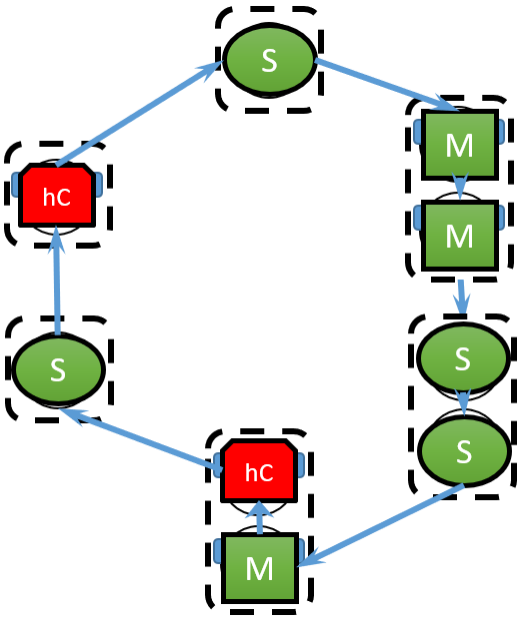


Figure 5.12: Step 3: Dispatching.
The dispatch program puts the mutation (M), half-crossover (hC) and selection (S) nodes into the blind and non-blind units

in Λ with odd indexes are entries in Λ^B in their original order, i.e.

$$\Lambda_{2k+1} = \Lambda^B_{k+1} \tag{5.12}$$

for all $k \in \mathbb{Z} \cap [0, x_b + x_{2b} - 1]$. Similarly, entries in Λ with even indexes are entries in Λ^N in their original order, i.e.

$$\Lambda_{2k} = \Lambda^N_k \tag{5.13}$$

for all $k \in \mathbb{Z} \cap [1, x_{nb} + x_{2nb}]$. By linking up the head and tail of Λ, we may build a ring satisfying all the rules discussed in the previous section 5.2.2.

5.4.3 Dispatching

The dispatch algorithm expands the sequence produced by the threading algorithm by dispatching mutation (M), half-crossover (hC), and selection (S) nodes into the units, as shown in Figure 5.12. A half-crossover node is a temporary unit to simplify the computation. It contains

one input and one output. In a later stage, two half-crossover nodes merge to a crossover node.

The algorithm first instantiates NB and 2NB units, as they must be selection nodes. Let $\mathcal{E}$ be the result of dispatching.

$$\forall i_N \in \{i : \Lambda_i = \mathcal{N}\} : \mathcal{E}_{i_N} = (S) \tag{5.14}$$

$$\forall i_{2N} \in \{i : \Lambda_i = 2\mathcal{N}\} : \mathcal{E}_{i_{2N}} = (S, S) \tag{5.15}$$

The next task is to dispatch M nodes. As we have discussed in section 5.2.1, we avoid the situation where an output of a crossover node connects to its own inputs. As a result, we avoid two hC nodes occupying a 2B unit. To achieve this, the algorithm dispatches at least one M node to each 2B unit. Therefore, each 2B unit either contains exactly one or two M nodes. In other words, the algorithm fixes a total of x_{2b} M nodes to 2B units, leaving $(m - x_{2b})$ to dispatch elsewhere.

It is trivial to assign an M node to each 2B unit, so we fix these M nodes while dispatching other nodes. To dispatch the $(m - x_{2b})$ M nodes in the ring, we first find out all possible dispatching locations by computing the index set $\mathcal{I}_{B+}$ of B and 2B nodes:

$$\mathcal{I}_{B+} = \{i : \Lambda_i = \mathcal{B} \vee \Lambda_i = 2\mathcal{B}\} \tag{5.16}$$

We randomly select a subset with $(m - x_{2b})$ members, i.e.

$$\mathcal{I}^*_{B+} = \text{randomsubset}(\mathcal{I}_{B+}, m - x_{2b}) \tag{5.17}$$

Then for each $i^*_{B+} \in \mathcal{I}^*_{B+}$, we dispatch an M node using

$$\mathcal{E}_{i^*_{B+}} = \begin{cases} (M), & \Lambda_{i^*_{B+}} = \mathcal{B} \\ (M, M), & \Lambda_{i^*_{B+}} = 2\mathcal{B} \end{cases} \tag{5.18}$$

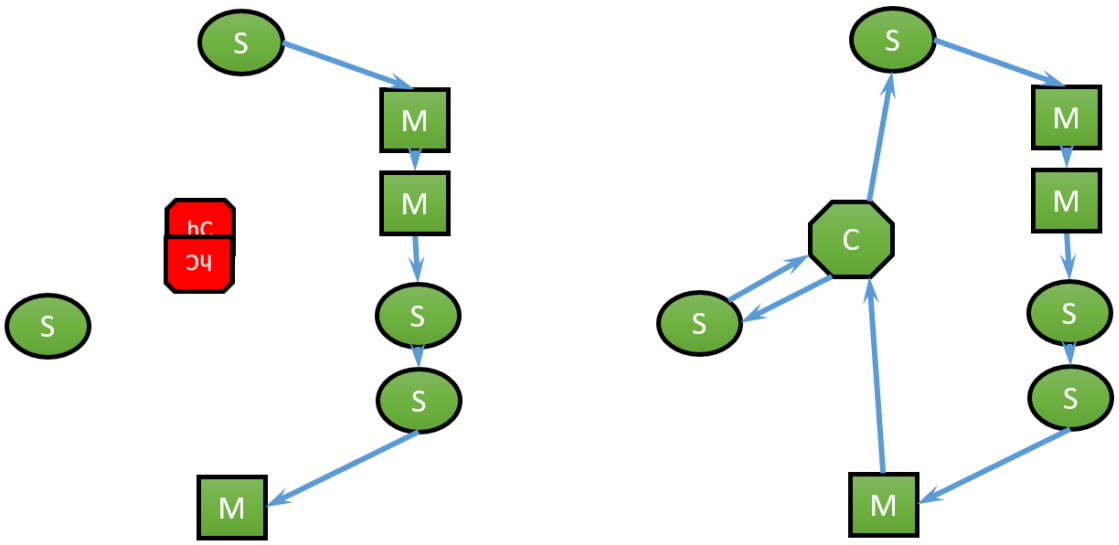


Figure 5.13: Step 4: Collapsing.
The collapsing step pairs half-crossover (hC) nodes into crossover (C)

Next, we fix the hC nodes to the remaining locations in $\mathcal{I}_{B+}$. Specifically, for all $i^{\#}_{B+} \in (\mathcal{I}_{B+} - \mathcal{I}^{*}_{B+})$

$$\mathcal{E}_{i^{\#}_{B+}} = \begin{cases} (hC), & \Lambda_{i^{\#}_{B+}} = \mathcal{B} \\ (hC, M), & \Lambda_{i^{\#}_{B+}} = 2\mathcal{B} \end{cases} \tag{5.19}$$

To introduce extra diversity to the topology, when $\Lambda_{i^{\#}_{B+}} = 2\mathcal{B}$, we randomly flip (hC, M) to (hC, M) with a probability of 0.5. The dispatch algorithm finally returns a list of nodes $\mathcal{L}$ by linking up all list in $\mathcal{E}$, i.e.

$$\mathcal{L} = \mathcal{E}_1 \parallel \mathcal{E}_2 \parallel \ \ldots \ \parallel \mathcal{E}_{|2(x_b + x_{2b})|} \tag{5.20}$$

where $\parallel$ is the list concatenating operator.

5.4.4 Collapsing

The collapsing algorithm transforms the outcome of the dispatch algorithm by randomly pairing half-crossover (hC) nodes, as shown in Figure 5.13. To achieve this goal, the algorithm first

extracts locations of hC nodes in $\mathcal{L}$, i.e.

$$\mathcal{I}_{hC} = \{i : \mathcal{L}_i = hC\} \tag{5.21}$$

Then it partitions $\mathcal{I}_{hC}$ into two subsets with equal size c. Next, the algorithm places the members in each subset into a sequence in random order, called C^L and C^R. Finally, we obtain the C nodes by combining the hC nodes in the two sequence with identical index, i.e.

$$C_i = (C_i^L, C_i^R) \tag{5.22}$$

The topology generator finally returns the ring $\mathcal{L}$ along with C. $\mathcal{L}$ contains the logical positions of all M, S and hC nodes, while C specifies the combination of hC nodes.

The user may then deploy the result of the topology generator in a reconfigurable device using multiplexers following the instructions in section 5.5. Experimental results related to the generator are available in section 5.6.

5.5 High Level Interface

To make PGP available to use for non-expert developers, we define an automatic framework based on chapter 3 to read a high level input and then generate the PGP system based on that input. As shown in Figure 5.14, there are three sections in the specification.

- Genetic nodes. The user can specify the genetic nodes included in the PGP and their methods, which comes from the pre-defined library.
- Connection topology. The algorithm in section 5.3 will automatically produce a topology, or the users can define the connection method by themselves. When switching the topology, the system will randomly choose the next one from these defined topologies.
- Check period. The user can define the check period in CPU for the inter-topology crossing. PGP transfers the fitness result to the CPU via FPGA-to-CPU (PCIe) link every

```
Genetic Nodes
Rs:  0xffff #seed for random number generator
M1:{
  method = 'bit-flip',
  rate = 0.06
} #Mutation Node 1

M2:{
  method = 'bit-flip',
  rate = 0.05
} #Mutation Node 2

C1:{
  method = 'one pint',
  rate = 0.46
} #Crossover Node 1
...
Connection Topology
#if no defined, the topology will be produced by
# the algorithm in section 5.3

Topology 1 {
  M1 -> M2
  C1L -> M1 #C1L: Left output of Crossover Node 1
  M2 -> C1R #C1R: left input of Crosover Node 1
...
}
Topology 2 {...}
...
Check Period:  check the result of PGP
Check_Period:10000
```

Figure 5.14: High Level Input for PGP

check period, and the CPU can decide if PGP switches to a new topology defined in the *connection topology*. A large check period reduces the overhead of communication but may increase the waiting time of switching, user can define the period to balance performance and overhead.

5.6 Evaluation

To evaluate the PGP approach, we apply it to two optimisation problems, the maximum satisfiability (MAX-SAT) and travelling salesman (TSP), and five numerical optimisation benchmarks [FSK+08, Col03] from section 3.5. These seven problems have different kinds of chromosomes, including fixed-point, permutation and floating-point chromosomes, and vary significantly in their scale and difficulty.

We compare the proposed work to the FPGA-based pGA accelerator in chapter 4, a GPU-based accelerator [MWMA09], and a CPU-based software based on the generational genetic algorithm [Col03] and GALib. The FPGA-based accelerator in chapter 4 targets a Virtex 6 (SX475T) FPGA. Our proposed system also targets the same platform for a fair comparison.

If the best individual fitness of a GA system remains unchanged for 10,000 generations, we consider a GA system to have reached its plateau fitness. When a CPU-based GA system reaches its plateau, we calculate the speedup by dividing CPU-based GA system time by the execution time for the FPGA-based systems to reach the same fitness.

5.6.1 Maximum Satisfiability Problem

The maximum satisfiability problem (MAX-SAT) is a classic NP-hard combinational optimisation problem. We apply three different topologies (PGP-1, PGP-2 and PGP-3) generated by the algorithm in section 5.4, and use the inter-topology crossing technology between these three topologies decoded as PGP-X. To reduce resource usage in the FPGA, we put the topology generator in the CPU, and send the new topology via CPU-to-FPGA link. To reduce the communication between CPU and FPGAs, we limit the FPGA to send out the fitness result every 10,000 cycles, that means we check the result and may switch topology at a check point. When the best fitness from PGP with one topology reaches a plateau without changing for 10,000 clock cycles, we carry out the inter-topology crossing. Inter-topology crossing never reduces the highest quality found using previous topology, because the optimum is still in local memory when switching topologies.

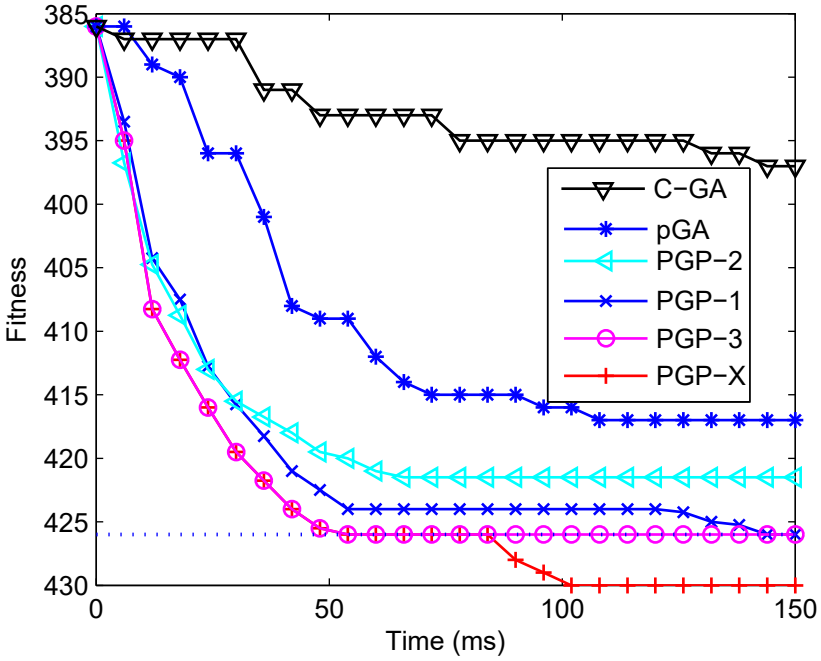


Figure 5.15: Best Fitness Found by Different Topologies for MAXSAT.
Dot line: the best fitness C-GA can find. pGA: FPGA-based pGA in chapter 4. C-GA: CPU-based GA. PGP-X: inter-topology crossing based on PGP-1, PGP-2 and PGP-3, which are generated by the algorithm in section 5.4.

(1) Performance Results

Because GA is non deterministic, we run the experiments multiple times. The average of best fitness found over time are shown in Figure 5.15. Our proposed work clearly outperforms the FPGA-based pGA system (pGA) and CPU-based system (C-GA), and when the C-GA system reaches the plateau fitness (shown as the dotted line in the figure), our proposed work has a speedup of 90 times over the multi-core C-GA, and 5 times over the pGA for the same fitness. For the same problem (uf100-430), our work is also 30 times faster than the GPU-based GA [MWMA09].

We can make the following three observation from Figure 5.15.

1. All three topologies provide significantly better results than the CPU and generational FPGA architecture in terms of convergence speed and solution fitness. This observation shows the increased optimisation power of the PGP approach.

2. The gaps in terms of the optimisation power between three different topologies are small.

This observation demonstrates the stable quality of topological structures produced by the proposed topology generator with the three rules.

3. The results for PGP-X with inter-topology crossing are better than all fixed topologies. This observation shows the usefulness of dynamic switching between topologies.

(2) Resource Optimisation

We also test the reduction of resource consumption using sparse crossbars. We build three systems consisting of 5, 10 and 20 genetic nodes respectively. In each system, we generate five different topologies using the generator proposed in section 5.4. The resource usages are shown in Table 5.1, and in all cases the sparse crossbars save resources, sometimes significantly. For example, full crossbar in MAX-SAT uses 26.38%, while the sparse crossbar only uses 21.50%, saving 18.50%.

5.6.2 Travelling Salesman Problem and Benchmarks

We also apply our proposed work to the travelling salesman problem (TSP) and five different functional benchmarks, including BF6, BF7, F11, 2DS and RF from section 3.5. After running multiple times, we compare the average of the best fitness versus time for the following three approaches: the FPGA-based generational pGA system (pGA) from chapter 4, pipelined genetic propagation using inter-topology crossing (PGP-X), and the CPU-based GA (C-GA). Table. 5.2 and Figure 5.16 show the combined results and PGP-X information, including clock frequency, resources and the numbers of genetic nodes such as selection, crossover and mutation. After the CPU-based work reaches plateau labelled as dotted line, our proposed work has an average speedup of around 5 times over the generational FPGA-based pGA system, and an average speedup of 90 times over the multi-core CPU-based GA. The blank in the table means pGA with generic structure sometimes cannot reach the fitness plateau of C-GA.

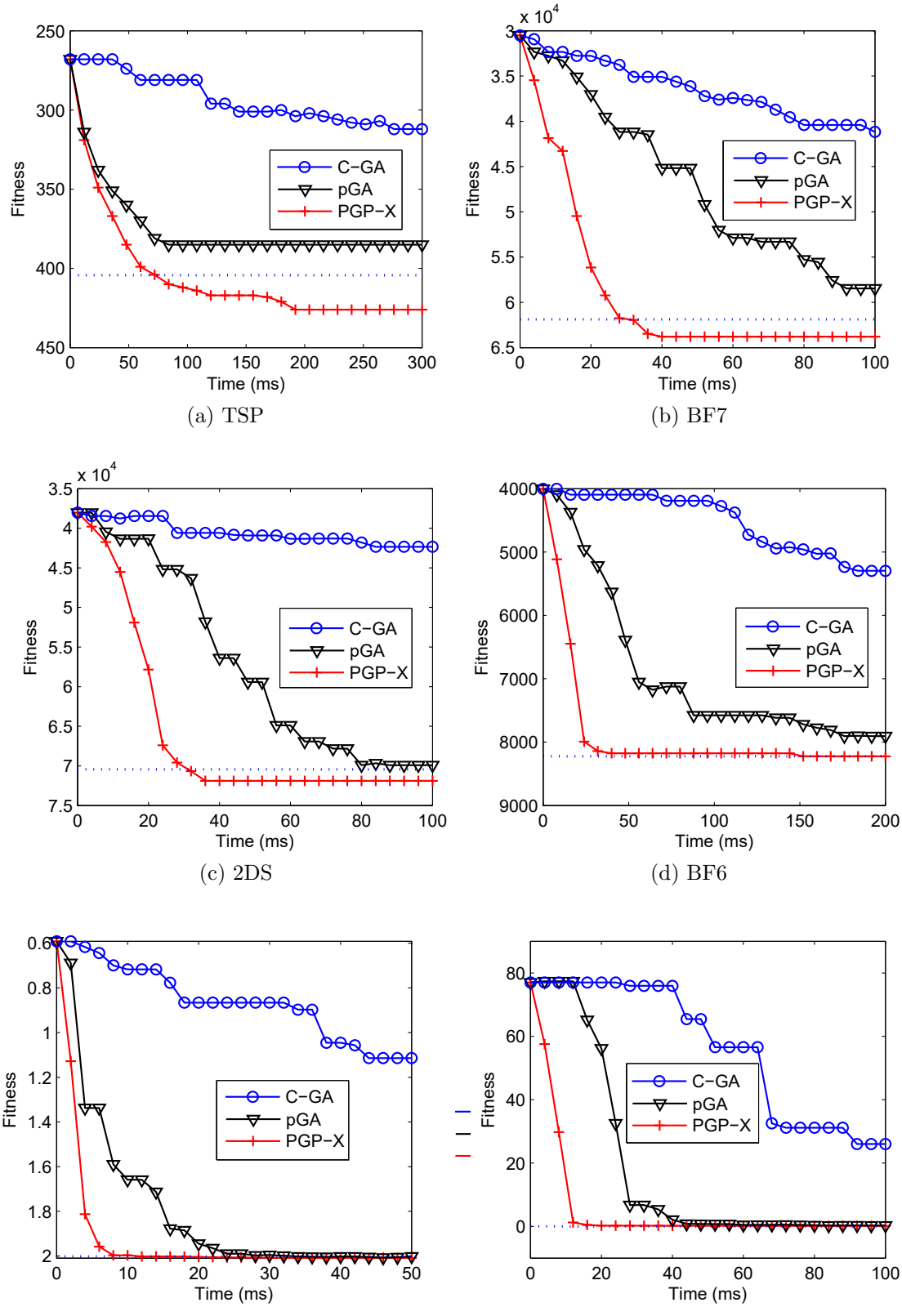
We can make the following three observations from Figure 5.16:

Table 5.1: Resource Usages of PGP for MAX-SAT Using Full Crossbar and Sparse Crossbar

# of Nodes	Crossbar	LUTs(%)	FFs(%)	BRAMs(%)
5	Full	6.17	5.11	3.85
5	Sparse	5.98	5.01	3.85
5	Reduction by	3.08%	1.96%	0.00%
10	Full	12.90	9.54	6.95
10	Sparse	10.31	8.49	6.20
10	Reduction by	20.08%	11.01%	10.79%
20	Full	26.38	17.86	13.44
20	Sparse	21.50	16.96	13.35
20	Reduction by	18.50%	5.04%	0.67%

Table 5.2: Experimental Results of PGP.
SP.P: Speedup over pGA, SP.C: Speedup over CPU-based GA, Res.: Resources, Freq. Frequency, M: mutation, C: crossover, S: selection. Fun.: Function

Fun.	# of M	# of C	# of S	Freq. (MHz)	Res. (%)	SP.P	SP.C
TSP	2	1	2	110	77.17	-	91
BF6	3	3	6	160	20.18	6	98
BF7	3	3	6	160	21.22	5.8	91
F11	3	3	6	160	46.24	3.8	83
2DS	3	3	6	160	38.73	3.1	103
RF	3	3	6	160	23.16	3.7	76
Avg	-	-	-	-	-	4.5	90



(a) TSP (b) BF7

(c) 2DS (d) BF6

(e) F11 (f) RF

Figure 5.16: Fitness Results of PGP.
Dotted line: plateau fitness of C-GA. pGA: FPGA-based pGA system in chapter 4. C-GA: CPU-based GA system

- The PGP approach provides better results versus time spent than the other GA systems for different types of objective functions, across a wide variety of problems.
- The PGP architecture is less likely to get stuck in a local plateau.
- The fitness of the pipelined genetic propagation architecture is smoother than those of the generational GA architecture.

To sum up, the pipelined genetic propagation outperforms the other two systems in terms of optimisation power and computational efficiency. A reason behind the observation is that inter-topology crossing helps to escape from local optimum by switching the topologies. Another reason is that the non-trivial topological structure brings high genetic diversity in the population. A third reason is that the distributed memory and lower fan-out enable our system to compile at a higher frequency than the generational FPGA-based GA. For instance, the maximal frequency of generational FPGA-based GA system for TSP is 75MHz, while that of our proposed work is 110MHz.

5.7 Summary

Most existing FPGA-based GAs are adaptations of classic software-based genetic algorithms, which model evolution using consecutive distinct generations. This approach introduces three significant problems in hardware. First, to compute a new generation of candidate solutions, the algorithm needs to wait for the evolution results of the previous generation, forming a hard data dependency. Second, the generational algorithm requires a large centralised memory space to store a pool of candidate solutions, presenting a memory-access bottleneck in FPGAs. Third, when a genetic algorithm in FPGA reaches a local optimum, it is difficult for users to affect the evolutionary process by re-tuning hardware architecture because a modification usually results in hours of recompilation.

To address these problems, this chapter proposes a novel GA approach called Pipelined Genetic Propagation (PGP), designed specifically for reconfigurable hardware. PGP is a hardware-

oriented way of structuring GAs as a graph of parallel pipelined components with local candidate storage, breaking the data dependency found in generational genetic algorithms, and eliminating the shared memory bottleneck. The chapter also designs a resource-efficient concrete architecture for PGP, demonstrating high performance compared to existing FPGA solvers, and dynamic changing of solver topology at run-time. In addition, we also propose a topology generator for producing PGP solver instances, allowing multiple candidate graphs to be created for a given FPGA.

According to the experiments on two classical problems (TSP and MAX-SAT) and five benchmark problems, pipelined genetic propagation is 5 times faster than the parallel GA system from chapter 4, 30 times faster than an existing GPU-based GA, and 90 times faster than a multi-core GA system, showing higher performance than other systems. The PGP strategy is flexible and can be extended to support more features, the next chapter extends PGP using multiple level parallelism, machine learning and partial reconfiguration.

Chapter 6

Pipelined Genetic Propagation Extensions

Classic GA systems require a considerable amount of FPGA resources, but the increase in performance is limited, due to large centralised memory, an inability to escape local optimum, and many pipeline stalls. To address these problems, the last chapter proposed a novel GA approach called Pipelined Genetic Propagation (PGP), which uses distributed memory, removes pipeline stalls, and escapes local optima by using changing topology at run-time. PGP is a flexible framework, so we will now consider a number of extensions.

This chapter extends the PGP platform to increase the flexibility and performance in the following ways:

1. Solving multiple level optimisation problems. The current PGP strategy has significant parallelism in the genetic operators and is much faster than classical GAs, but for multiple level optimisation problems it still needs a good combination of multiple level parameters. Adding multiple levels in the PGP architecture can address this problem, and further increases the parallelism.

2. Guiding topology change. Topology change is a core concept of the PGP strategy, but chapter 5 only uses random topology switching when a PGP gets stuck in a local optimum.

Here we observe that every topology might not have the same quality, and present a smarter strategy based on a machine learning method which can bring greater benefits.

3. Further increasing flexibility. While PGP already has good flexibility in the topology connecting genetic nodes, this chapter proposes some additional ways to increase the flexibility in the genetic notes and the connections between them, including using partial reconfiguration.

The chapter describes these extensions in three sections and briefly evaluate how these new features improve PGP's performance and flexibility:

- section 6.1 designs a novel Recursive Pipelined Genetic Propagation called R-PGP, which can effectively solve bi-level optimisation problems.

- section 6.2 applies machine learning to PGP, using Q-learning to guide the inter-topology crossing. The result shows the performance of Q-PGP is improved by 5% freely compared to random switching without any manual tuning.

- section 6.3 proposes some possible ways to maximise the flexibility of PGP system, such as using Partial Reconfiguration (P-PGP) to change the genetic nodes and connection at run time.

6.1 R-PGP: Recursive Pipelined Genetic Propagation

Multiple-level optimisation problems often arise in the real world, for example bi-level optimisation is applied to tax policy making between government and companies. To solve these problems efficiently, this section proposes to extend the original PGP by adding multiple level parallelism.

6.1.1 Motivation

The bilevel optimisation problem (BLP) is a subclass of optimisation problems in which one of the constraints of an optimisation problem is another optimisation problem. BLP is widely used to model hierarchical decision making involving two levels, namely an upper level and lower level optimisation problem. In BLP, the optimal solution to the lower level optimisation problem is feasible solution to the upper level problem, which makes it particularly difficult to solve. That is to say, for every solution in the upper level problem, the lower-level optimisation has to find the lower level optimum, which results in a large computational load across both levels.

PGP can solve a lot of optimisation problems efficiently, however, it is still not easy to solve bilevel optimisation problems, because it is hard to design a good chromosome combining both levels and an effective evaluator. However, because the structure of PGP is flexible, it is possible to add a new level in the architecture. This section proposes Recursive Pipelined Genetic Propagation (R-PGP) to solve BLP efficiently on FPGA by adding multiple-level parallelism to the original PGP architecture.

To the best of our knowledge, we are the first to solve BLP on FPGA. The goal of this section is to build a hardware-based BLP solver which operates mainly on the FPGA. The highly pipelined architecture and concurrent computation in two levels greatly reduces computation time, compared to conventional CPU-based BLP solvers.

6.1.2 Bilevel Optimisation Problem

The bilevel optimisation problem (BLP) is a two-level nested optimisation problem shown in (6.1). In BLP, the lower level optimisation problem (f) appears as a constraint of the upper level optimisation problem (F).

$$
\begin{aligned}
&\max_{x \in X} && F(x, y) \\
&\text{subject to} && G(x, y) \leq 0 \\
&&& \min_{y \in Y} && f(x, y) \\
&&& \text{subject to} && g(x, y) \leq 0
\end{aligned}
\tag{6.1}
$$

BLP is used to model two-level hierarchical decision making where two agents are involved: the *leader* and the *follower*. The leader, corresponding to the upper level optimisation problem in BLP, optimises x with respect to the objective $F(x, y)$, constraints $G(x, y)$ and the follower. Meanwhile, in the lower level problem, the follower optimises y with respect to the objective $f(x, y)$ and constraints $g(x, y)$. When making a decision with a specific value x, the leader has to takes account of the follower's potential response(s) - the optimal solution(s) to the lower level problem given x. The leader can foresee the follower's potential reactions. The leader has the initiative to choose an optimal x^* such that when the follower reacts as predicted, the overall result is the optimal for the leader. In the setting of BLP, the leader effectively guides the follower.

There are many practical problems that fit BLP's structure very well [Bar98]. A common case is the interaction between the government's policy and the industry or person affected [SMFD13], [BLMS01]. In [SMFD13], the leader is the government who sets a taxation policy with the aim of maximising tax revenue; the follower is a mining company who wants to maximise its profit under the limitation of the tax. In [BLMS01], toll gate setting and drivers' reaction are modelled as a BLP. In other areas such as corporation management [Bar83] and engineering design [KNPDK98], BLP is also found to be a powerful modelling approach.

While being useful, BLP is intrinsically very challenging to solve. This is because for each upper level candidate x_0, the lower level optimisation task must be solved to find the lower level optimum $y_0 = \underset{y \in Y}{\operatorname{argmin}} f(x_0, y)$, which will be used to evaluate upper level objective function $F(x_0, y_0)$. Additionally, the two sets of constraints can often be inconsistent, which introduces extra difficulty for convergence. Even in the linear case, BLP is known to be NP-hard [Bar98]. As a result, heuristics, such as genetic algorithms, are the major ways to solve

BLP. When using heuristics, solving a BLP often requires solving the lower level optimisation problem many times, which means the lower level objective function $f(x, y)$ will be evaluated frequently, such as 1 million times [SMD12]. This large amount of evaluation is extremely time consuming for a CPU, especially when these functions are non-trivial. As a result, objective function evaluations are the major performance overhead.

6.1.3 Recursive PGP Architecture

In this section, we present a new PGP architecture to solve BLP efficiently on FPGA. The proposed hardware architecture, Recursive Pipelined Genetic Propagation (R-PGP), extends PGP to the bilevel case.

R-PGP is a nested optimisation approach using genetic algorithm, so it can be applied to a wide range of BLP problems regardless of smoothness or convexity. In R-PGP we use a two-level PGP, with the lower-level PGP used to solve the low-level optimisation problem for each upper level candidate evolved in the upper level PGP. Because the selection node is the only place to do the evaluation, we put the low-level PGP system into the selection node of the upper level optimisation. Meanwhile, the upper level selection, mutation and crossover nodes form the upper-level PGP system. That is to say, our R-PGP system is essentially PGP instances nested within a large PGP instance, while all PGP instances evolve concurrently, we call the architecture a 'recursive' version of PGP.

Figure 6.1 shows the structure for whole R-PGP system. As seen from the figure, lower level PGPs (LPGP1, LPGP2, LPGP3 and LPGP4) are instantiated within the upper level selection nodes of UPGP. The upper level PGP and lower level PGPs all have the data path in the center, which can be switched at run-time in order to escape from local optima in the upper level.

When compiling the R-PGP system, the user needs to specify the lower level objective function $f(x, y)$, the upper level objective function $F(x, y)$, and constraints. In the same way as section 5.5, when running the FPGA system, parameters for genetic operations are uploaded to R-

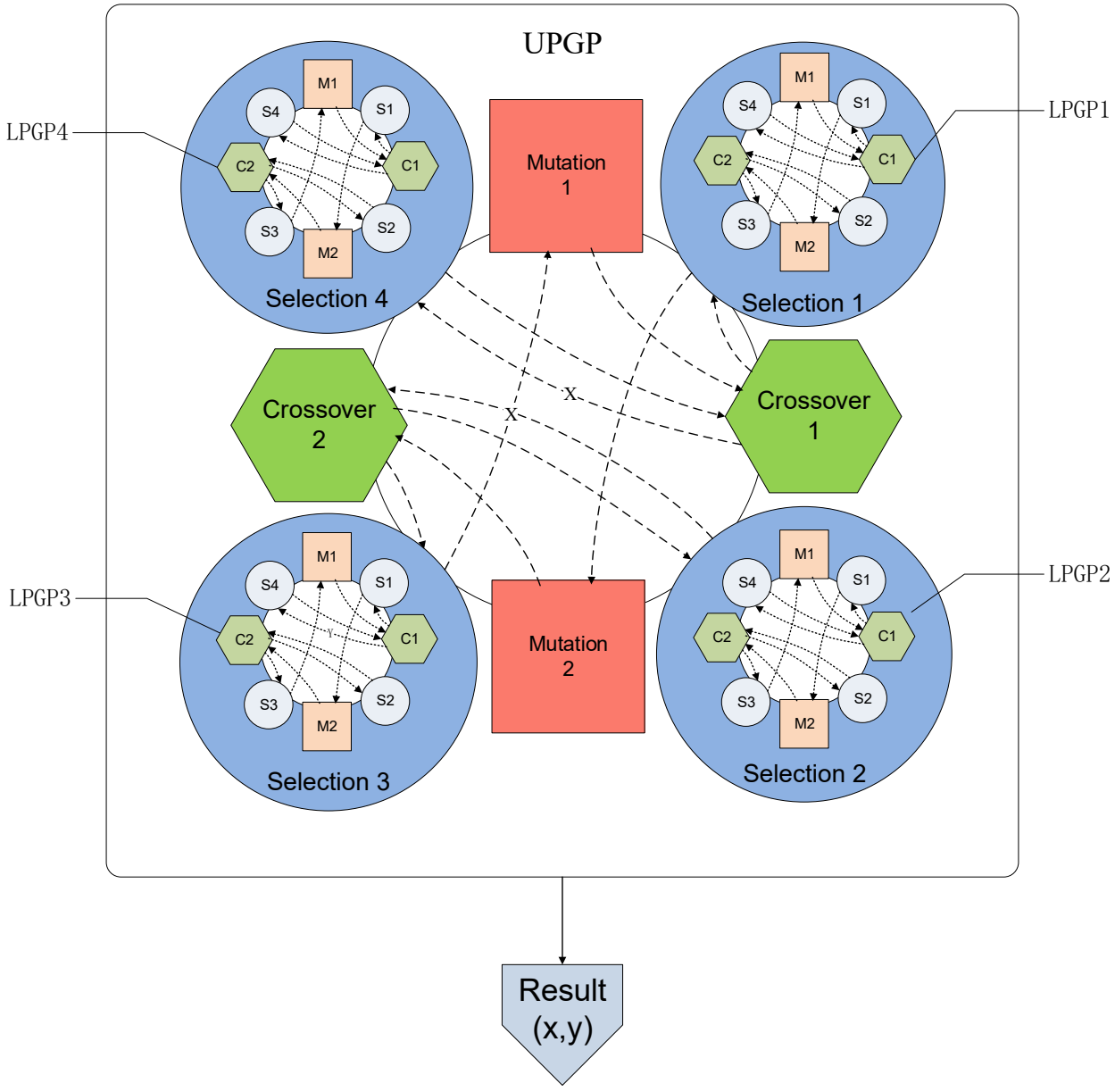


Figure 6.1: Whole R-PGP Architecture.
UPGP: Upper-level PGP, LPGP: lower-level PGP.

PGP at the beginning, such as the crossover and mutation rates, lower and upper level check periods. To minimise CPU-to-FPGA I/O overhead, random numbers are generated on FPGA, so R-PGP does not require any inputs from the CPU when running. R-PGP sends results back to CPU every check period. The details of the two levels are described in following subsections.

(1) Low Level PGP

The lower level PGP searches the optimal lower level candidate for each upper level candidate, i.e. to find $y_0 = \underset{y \in Y}{\operatorname{argmin}} f(x_0, y)$ for each x_0, as can be seen in Figure 6.2. This can also be interpreted as finding the follower's optimal response given the leader's action. As shown in Figure 6.3, a lower level selection receives the x from the upper level and y from other genetic nodes, and evaluate them together to calculate the fitness. At every cycle, it sends out y selected from the local memory to other genetic nodes for evolution.

The number of selection, mutation and crossover nodes can be adjusted to fully exploit FPGA resources. For example in Figure 6.2, the lower level PGP is composed of 4 lower level selection nodes, 2 lower level mutation nodes and 2 lower level crossover nodes.

(2) Upper Level PGP

Upper level optimisation is also carried out by a PGP composed of several selection, mutation and crossover nodes. The upper level mutation and crossover nodes have the same structure as their lower level counterparts. Although similar, the upper/lower level mutation and crossover nodes are not interchangeable in general, because lower and upper level candidates may have different chromosomes.

As required by nested optimisation, for each upper level candidate x_0, we need to find its corresponding optimal lower level candidate $y_0 = \underset{y \in Y}{\operatorname{argmin}} f(x_0, y)$. Then the fitness of x_0 is evaluated by the upper level objective function $F(x_0, y_0)$.

With this in mind, we put the lower level PGP inside the upper level selection node, so that optimal lower level candidate y_0 will go to the upper level evaluator $F(x, y)$ together with upper

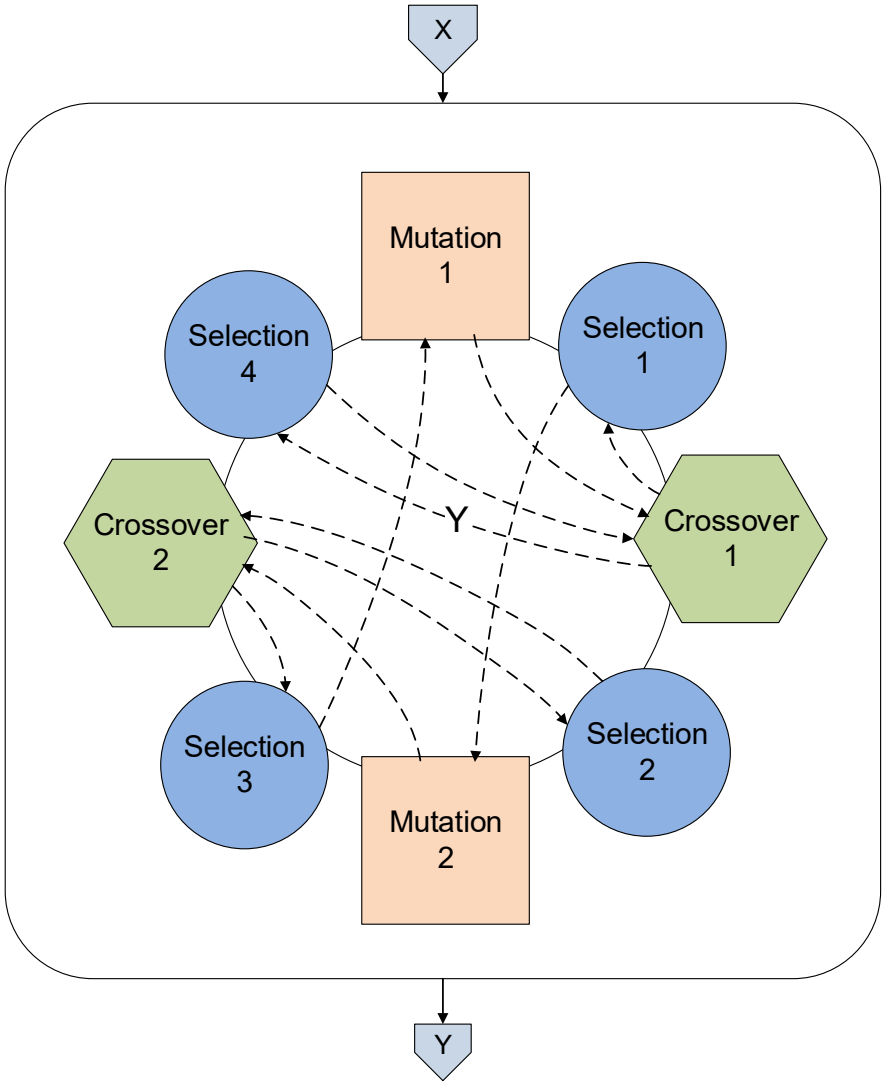


Figure 6.2: Lower Level PGP Architecture.
Lower Level PGP performs lower level optimisation for each upper level candidate x and outputs corresponding optimal lower level candidate y. The connections between selection, mutation and crossover nodes can be switched by inter-topology switch at run-time. The dynamic switch helps PGP to escape from local optima.

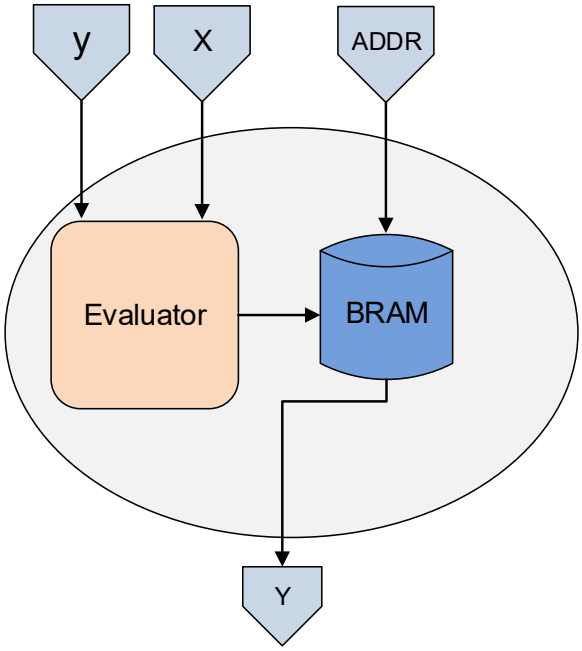


Figure 6.3: Lower Level Selection in R-PGP.
Selection node receives x and y, and sends out y from local memory every cycle

level input x_0. Figure 6.4 shows the structure of upper level selection node. The upper level candidate is fed into the lower level PGP in order to find its corresponding optimal lower level solution. Then the upper level candidate and corresponding optimal lower level candidates are evaluated and compared with existing candidates from local BRAM. Then an existing candidate in BRAM will be propagated to the output port, while the candidate with higher fitness will be written back into BRAM. BRAM address is generated on FPGA using a uniform random number generator [TL07].

6.1.4 Implementation and Comparison

The number of selection, mutation and crossover nodes in both lower level and upper level are customisable to exploit the available resources on FPGA. The R-PGP configuration used in this section corresponds to Figure 6.1 is:

- Lower Level: Selection×4, Mutation×2, Crossover×2

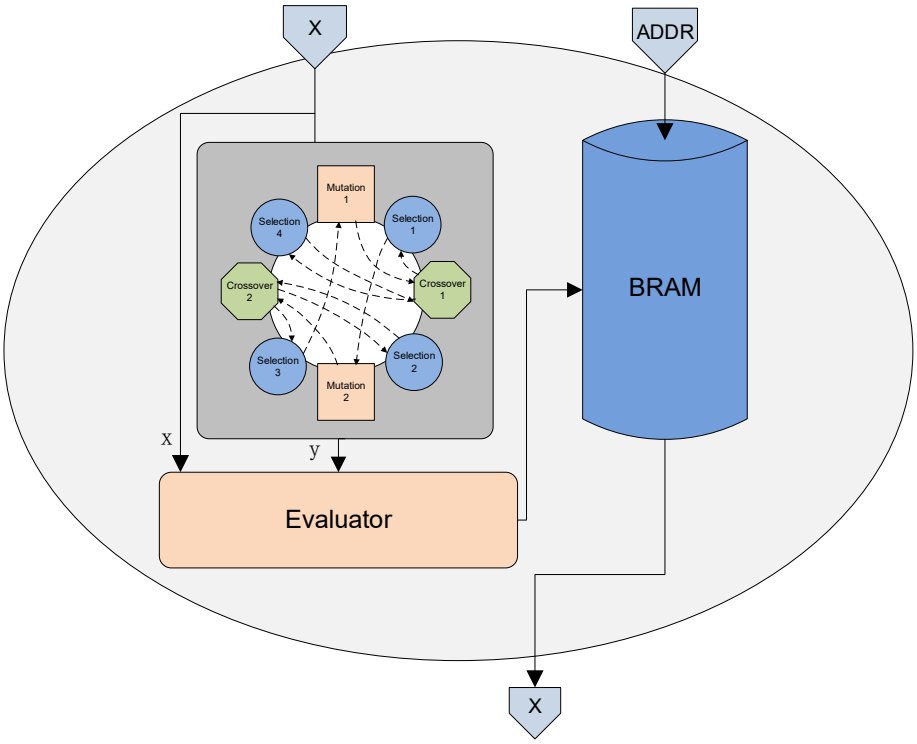


Figure 6.4: Upper Level Selection Node in R-PGP.
The lower level PGPs are nested inside the upper level selection nodes to find the lower level optimum y for the upper level candidates x. The selection receives x from other upper level genetic nodes and outputs x from local BRAM every cycle

- Upper Level: Selection×4, Mutation×2, Crossover×2

We choose these number after trying several different configurations, but users can easily configure these parameters using a high level description.

The proposed R-PGP system is described using a dataflow programming language by Maxeler Technologies. User will need to specify the BLP problem in a high level input, then function evaluators $F(x, y)$ and $f(x, y)$ will be instantiated within R-PGP. Users also can choose legal PGP topologies for the lower level PGP and the upper level PGP using the algorithm presented in section 5.4. For the lower level PGP, the check period is 16384 clock cycles, and for upper level PGP the check period is 16384*16384 cycles. We switch the topologies at the check points until the optimum are found. The check periods are empirical numbers which can be configured by the user. We choose the number to balance performance and the overhead of the CPU-to-FPGA communication.

The proposed R-PGP system is compared with BLEAQ, a recent CPU-based genetic BLP solver [SMD13]. It makes quadratic approximations of the lower level problem whenever possible to boost performance. The BLEAQ solver is publicly available, so we use BLEAQ's open sourced MATLAB version in our tests.

The proposed R-PGP system is built on an FPGA-based platform with Altera Stratix-V 5SGSD8. FPGA frequency is set to 100MHz. The BLEAQ program runs in a 64-bit MATLAB R2012b environment on a server with dual Intel Xeon E5-2640 6 Core CPU with 2.5GHz speed and 64GB DDR3-1333 memory. The operating system used is CentOS 6.4.

To evaluate the proposed R-PGP system, we use four benchmark BLP problems proposed in [SMD12]. These BLP problems have different properties, and the problem can be cooperative or conflicting [SMD14]. In the cooperative relationship, an improvement in the lower level solution leads to an improvement in the upper level solution too, while in the conflicting relationship the reverse is true.

- SMD1: Both the lower and upper level problems are convex, while the two levels cooperate with each other.
- SMD2: Both the lower and upper level problems are convex, while the two levels are in conflict.
- SMD5: The lower level has difficulties in convergence introduced by the Rosenbrock's function. Lower level optima lies in a narrow, parabolic valley. The two levels are in conflict.
- SMD6: The lower level contains infinitely many optimal solutions for each upper level candidate, but within the entire global solution set only one lower level point corresponds to upper level optimum. The two levels are in conflict.

These problems are scalable in terms of dimension. In our tests, we set the total dimension to 10: 5-dimensional lower level problem and 5-dimensional upper level problem, which is difficult to solve.

Table 6.1: Resource Usage of R-PGP on Stratix-V 5SGSD8 FPGA

Problem #	Logic Usage	DSP Usage	BRAM Usage
SMD1	94.43%	100.00%	100.00%
SMD2	46.00%	82.53%	66.46%
SMD5	38.56%	78.45%	45.93%
SMD6	33.28%	47.48%	45.46%

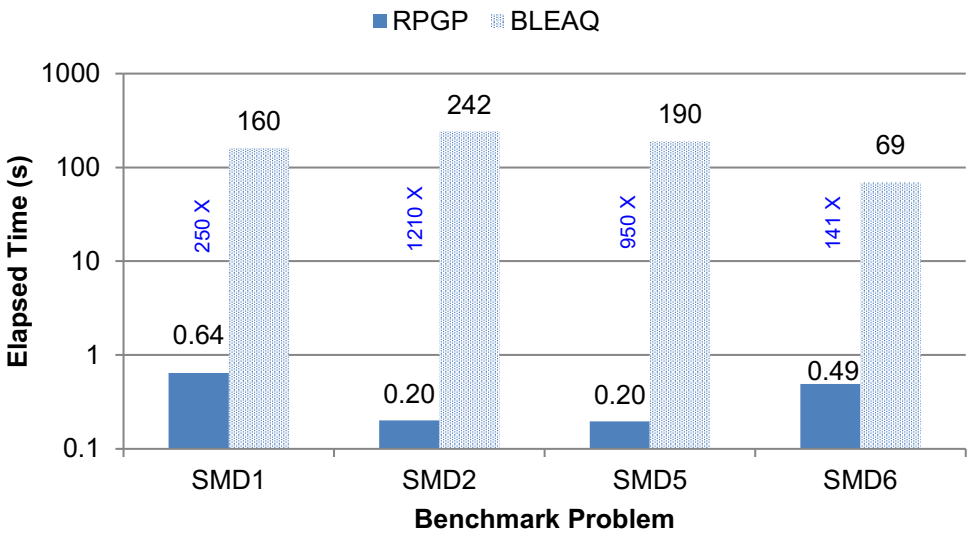


Figure 6.5: Elapsed Time of R-PGP and BLEAQ to Converge to $err < 0.01$. Speed-up factors achieved: 250(SMD1), 1210(SMD2), 950(SMD5), 141(SMD6).

6.1.5 Resource Usage and Performance

Table I shows the resource usage of Stratix-V 5SGSD8 FPGA. Although the R-PGP configuration and parallelism are the same for all problems, resource usage varies significantly, because most of the FPGA resources are taken by the upper level and lower level evaluators. For different problems these evaluators result in various resource usages. For the SMD1 problem in the 10-dimensional case, both the upper level and lower level objective function contain two $tan()$ functions, which are expensive in hardware. In contrast, SMD6 objective functions do not involve such expensive functions, so its resource usage is much less.

We compare the elapsed time for R-PGP and BLEAQ to achieve the same level of accuracy. In our tests we set the error threshold to 0.01 according to the features of the problems, and the elapsed time is shown in Figure 6.5. As can be seen in the figure, R-PGP demonstrates

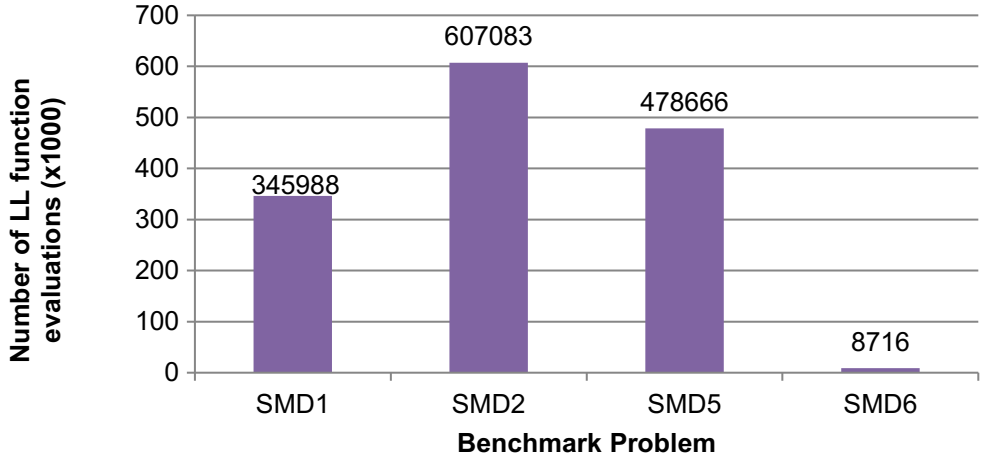


Figure 6.6: Number of Lower Level Objective Function Evaluations for BLEAQ

significant speed-up against BLEAQ. The best acceleration number is achieved for SMD2 test problem, in which R-PGP running on FPGA is 1210 times faster than a Xeon-based BLEAQ solver in software. The reasons probably are: a) SMD2 has the largest number of function evaluations; b) objective functions in SMD2 contain $log()$, which is expensive for CPU to calculate.

6.1.6 Result Discussion

The major computational bottleneck for CPU-based BLP solvers is the huge number of lower level objective function evaluations. As can be seen from Figure 6.6, for test problems SMD1, SMD2 and SMD5, BLEAQ needs hundreds of thousands of lower-level evaluations, which is very time consuming for CPU, especially when the functions are non-trivial. For SMD6, the BLEAQ solver is quickly converged to the threshold after using about 10,000 lower level function evaluations. Consequently, in Figure 6.5 the CPU time for SMD6 is much shorter than those for other test problems. In general, the hardware speed-up is likely to be the most significant for the problems which are difficult to converge and whose objective functions are costly to evaluate, such as those involving $tan()$ or $log()$. Apart from evaluations, the genetic operations such as mutation and crossover are also slower on CPU than on FPGAs because PGP uses a fully pipelined structure. For SMD6, although the number of function evaluations are low (such as 8716), it still takes 69 seconds to reach the 0.01 threshold.

On the other hand, in the FPGA-based R-PGP system, all lower level and upper level function evaluators are built in hardware and highly-pipelined. Therefore, R-PGP is actually indifferent to the time cost of evaluating objective functions, because on every cycle a fitness result will come out from the pipeline. As PGP is a non-generational approach, the lower level PGP architecture does not need global synchronisation in each generation, it is free from pipeline stall and has better efficiency.

In terms of area cost, R-PGP is sensitive to the objective function's complexity. In each lower level selection node, there is a lower level objective function evaluator. As a result, as parallelism increases, evaluation units quickly occupy the available FPGA resources. The R-PGP systems for SMD1 and SMD6 have the same level of parallelism so they have the same number of objective function evaluation units. However, SMD1's $tan()$ functions use a lot more resources than SMD6's polynomials, occupying almost full FPGA. For random number generation, FPGA is very efficient. In total they only use about 5% logic and 4% BRAM.

6.1.7 Section Summary

Bilevel optimisation problem (BLP) is a nested optimisation problem in which one optimisation problem is nested within another as a constraint. BLP is a useful modelling approach for hierarchical decision making, where the leader and follower correspond to the upper level and lower level problems, respectively. However, BLP is an NP-hard problem, so CPU-based heuristic solvers suffer from a very large number of lower level objective function evaluation. In this section, we propose the first FPGA-based BLP solver. Our system operates on FPGA and all function evaluators are built in hardware, which greatly reduces function evaluation overhead. The proposed Recursive Pipelined Genetic Propagation (R-PGP) architecture extend the PGP's features with multiple-level parallelism. Dynamic topology switching is also used to escape from local optima in lower and upper level. Experimental results using four BLP benchmark problems show that the proposed R-PGP system can achieve up to 1200 times speed-up compared to BLEAQ, a recent CPU-based BLP solver.

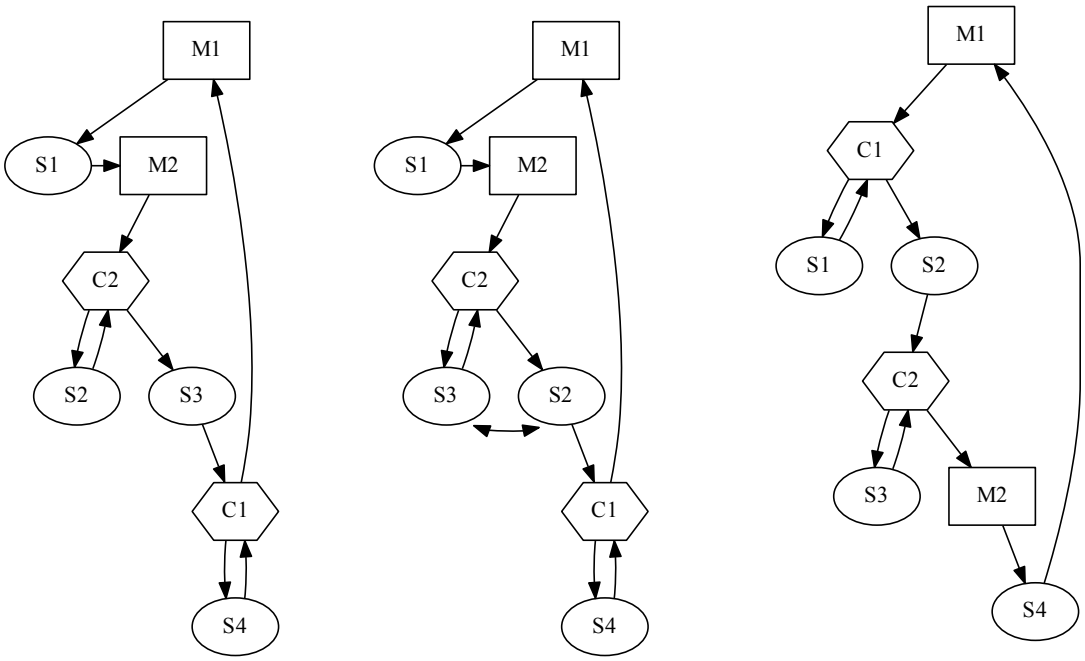
6.2 Q-PGP: Q-learning Pipelined Genetic Propagation

6.2.1 Motivation

In a PGP system, inter-topology crossing is a very important feature, because it helps the PGP escape from local optima and brings additional diversity to the evolutionary process compared to classical GA. We could switch the connection topology after fixed periods, or after the current topology get stuck in local optima. The previous switching method randomly selects a legal topology generated by the algorithm mentioned in chapter 5. This method is very simple and fast but still has space to improve. We observe that every topology has its own evolutionary characteristics and topology behaviour, some are very similar but others have much differences between them. For example in Figure 6.7, switching from T_2 to T_3 does not change the evolutionary process much compared with switching from T_2 to T_1. However, it is hard to predict if a big difference only in topology will bring benefits to the evolutionary process, because PGP also contains individuals inside, which make a great contribution to the evolutionary process. A topology change produces new individuals, which adds extra variation to the pure topology difference. Therefore, a smart method such as machine learning may help to guide the topology switch. This section applies Q-learning to PGP in order to learn about topology quality. We choose Q-learning because it works well in dynamic unknown environments as well as combines long-term and short-term benefits.

6.2.2 Reinforcement Learning and Q-learning

Reinforcement learning is a machine learning techniques which learns how to act or behave [KLM96]. The method is concerned with how agents in a dynamic environment take actions so that they end up maximising their cumulative reward. This particular area is well suited for dynamic environment problems which include a long-term versus short-term reward trade-off. The basic reinforcement model is built from the following: a set of environment states S and a set of possible actions A, together with some rules for transitioning from one state to another.



(a) Topology T_1 (b) Topology T_2 (c) Topology T_3

Figure 6.7: Three Topologies: T_1, T_2, and T_3.
T_2 and T_3 are more similar than T_1 and T_2, but they still carry different evolutionary characteristic

When developers use reinforcement learning, they can add more rules to this model which deal with how the reward of a transition is allocated, as well as rules that describe what some of the agents observe.

The model-free reinforcement learning technique we are interested in, is *Q-learning* [Wat89]. Q-learning can also be described as an off-policy learner, because learning the value of the optimal policy is independent of the agent's actions. This technique learns an optimal policy no matter what the agent does, as long as it explores enough. It usually combines the value of the policy the agent is actually carrying out and its historical rewards, so the learner can take into account the rewards associated with the whole exploration. The model contains an agent, a set of actions a and states s. One action is to perform a switch from one state to another, and is associated with a reward. The goal of Q-learning is to maximise the total reward [Wat89]. the Quality of state-action for updating is:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \times [r_{t+1} + \lambda \times \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{6.2}$$

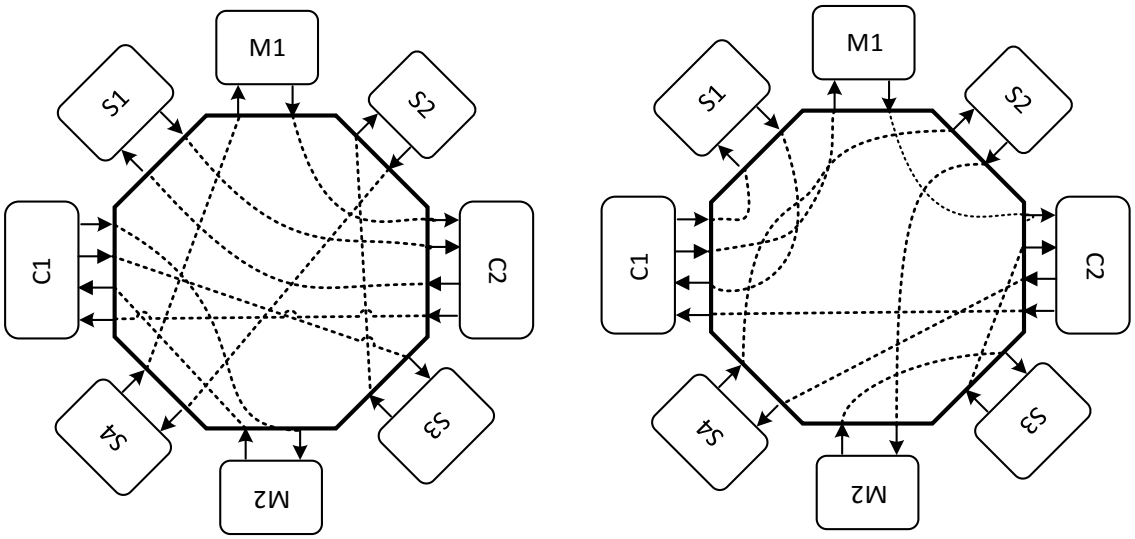


Figure 6.8: Two Topologies S_x and S_y for Switching with Q-Learning

While s is the state, r_{t+1} is the reward when s_t carries out action a_t, α is the learning rate, λ is discount factor, and $Q(s_{t+1}, a)$ is the estimated future reward, which we hope to maximise. The formula shows the reward relies on both the current reward and future reward. Here we set $\lambda = 0.01$ and $\alpha = 0.8$.

6.2.3 Inter-Topology Crossing with Q-learning

When the fitness refuses to improve during the evolution process, the system loads a different topology to escape from local optima. Compared with the previous approach of switching to a randomly selected topology, we consider that it is possible to improve the optimisation power by conducting inter-topology crossing in a controlled manner with heuristics.

We design a learning agent to control the inter-topology crossing activities. Let $|T|$ be the number of topologies in the system, each of which corresponds to a unique state S. When the GA system reaches a fitness plateau, the agent must take an action to switch the topology to a different one. The agent has $(|T| - 1)$ possible actions to switch to the other $(|T| - 1)$ states. The agents receives a reward signal calculated as the fitness gained before this topology reaches plateau again. The agent learns a deterministic policy from the reward signals using Q-learning with the ϵ-greedy policy.

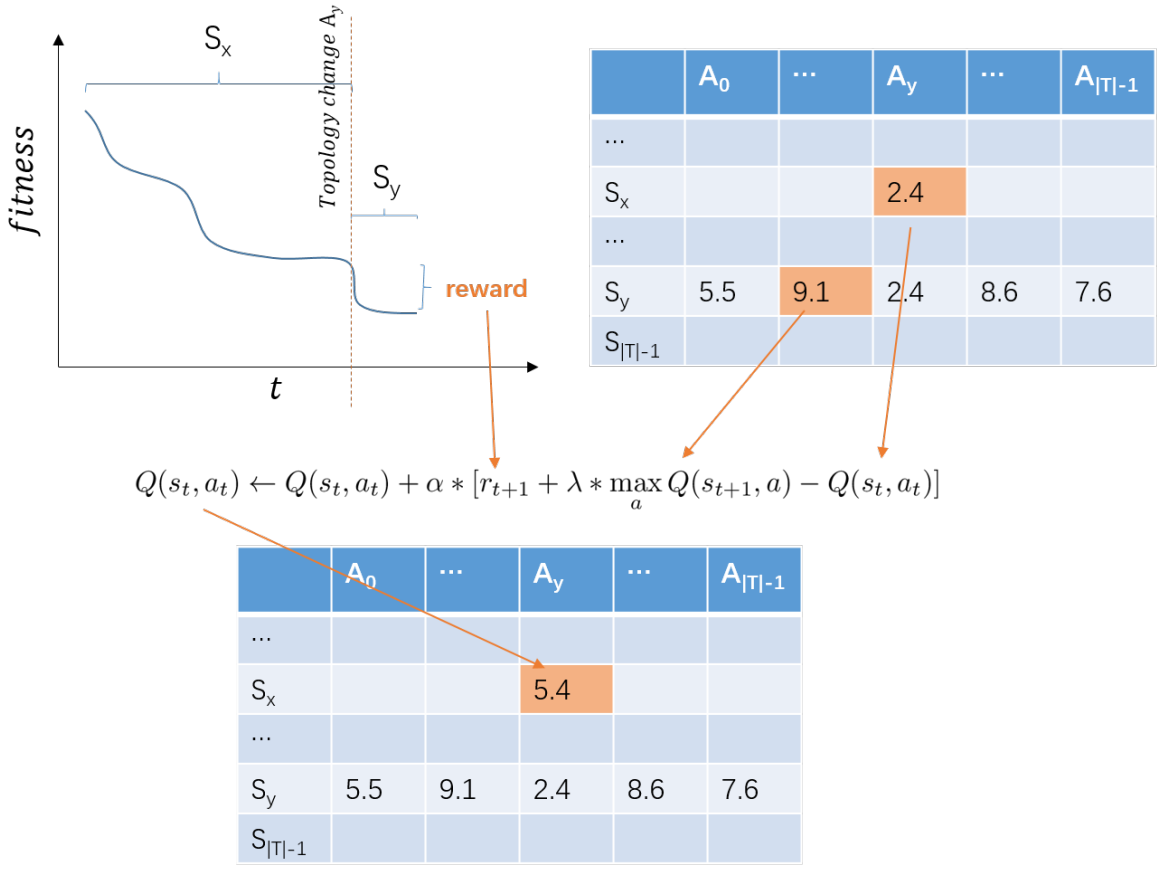
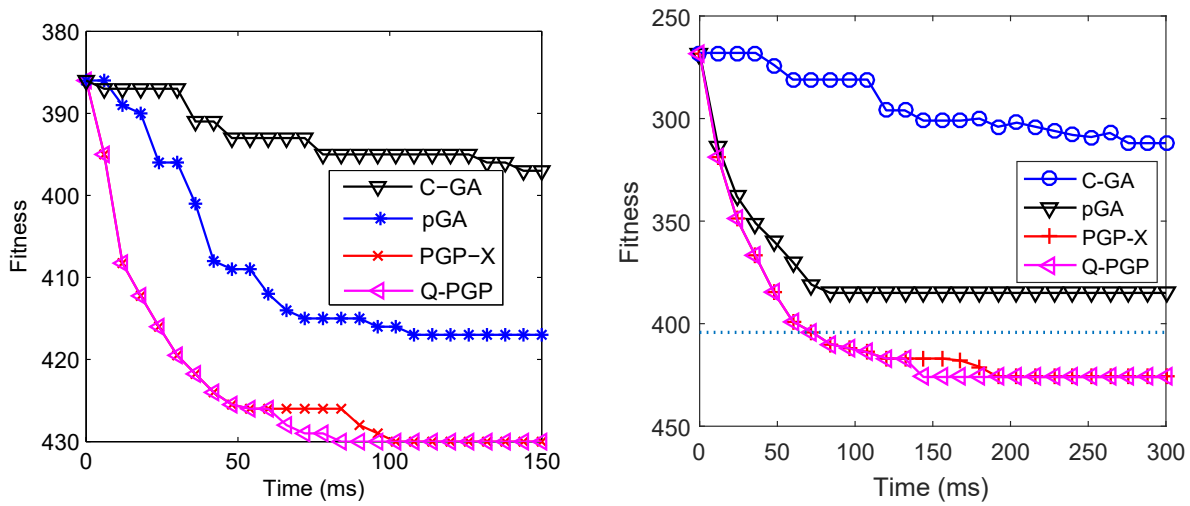


Figure 6.9: The Action Switching from Topologies S_x to S_y, and the Reward Updating



(a) MAXSAT (b) TSP

Figure 6.10: Q-PGP's Experiment Results of MAXSAT and TSP.
PGP-X: PGP with randomly inter-topology crossing. Q-PGP: Q-learning guided PGP. pGA: pGA in FPGAs from chapter 4. C-GA: CPU-based GA

In the reread table for PGP, there exist $|T|$ rows for all topologies, and $|T|-1$ columns for all possible topologies changed to. Figure 6.8 shows two topologies S_x and S_y, while Figure 6.9 shows the reward by action A_y switching from S_x to S_y. The improved fitness value will be added to the quality table according to equation 6.2. After a long-time training, the table becomes a reference for the further switching.

6.2.4 Experiments

To evaluate the Q-learning guided PGP (Q-PGP), we apply it to a NP-hard MAX-SAT and TSP. We compare the proposed work to the FPGA-based generational pGA accelerator done in chapter 4 (pGA), and a multi-core CPU-based software GALib. The pGA accelerator targets a Virtex 6 (SX475T) platform. This proposed system also targets the same platform for fair comparison. We also implement PGP with random inter-topology crossing (PGP-X) for comparison. The speedup is the ratio of the execution times of two systems when they reach the same fitness. We calculate the speedup when the slower system reaches its fitness plateau, namely its fitness does not change in 10,000 cycles.

We generate 5 different topologies for 10 genetic nodes in FPGAs, and use them to solve the

same MAX-SAT instance uf100-430 in section 4.1 and TSP problem. When the system with one topology reaches a fitness plateau, PGP-X changes the topology to a randomly selected one, while the Q-PGP changes the topology according to the learning table. The learning results come from the multiple executions of PGP-X, which learns 10,000 actions and takes 16 minutes to finish.

From the experiment results shown in Figure 6.10, we have three observations:

- Q-PGP can reach optimum solutions more quickly than PGP-X, showing the effect of Q-learning method: 5% faster than PGP-X.
- All the FPGA-based systems are faster than CPU-based work, showing the great potential of FPGA-based acceleration. The speedup of Q-PGP over CPU-based GA is 100 times.
- PGP systems achieve better results than the generational FPGA-based GA system pGA, indicating the advantages of inter-topology crossing and fully pipelined architecture. The speedup ratio of Q-PGP over pGA is 5 times.

6.2.5 Section Summary

PGP uses inter-topology crossing to escape from local optimum, and we observe every topology has its evolutionary characteristic. Using random switch is simple but ignores the difference in the quality of the topologies. This section studies a smarter topology change by using Q-learning. The experiment shows the increased effectiveness of Q-learning compared with randomised method.

6.3 P-PGP: Partially Reconfigurable Pipelined Genetic Propagation

Compared with the previous GA systems in FPGA, PGP allows a flexible strategy for the topology connecting genetic nodes. However, there is still some space left, we could further

improve the flexibility in the PGP approach, such as the communication unit as well as the genetic nodes. Partial reconfiguration is a technology which reconfigures part of the FPGA rather than the complete FPGA. This section proposes a partial reconfigurable PGP (P-PGP) which mainly uses partial reconfiguration to improve the flexibility.

6.3.1 Partially Reconfigurable Topology Unit

Inter-topology crossing is an important part of a PGP system, which can often be used to escape from local optima. To reduce the overhead of switching topology and make the new topology available immediately, we previously used a crossbar made up of multiplexers in chapter 5. To support all legal topologies for switching, we could use a full crossbar, but it consumes too much resources because even a small number of genetic nodes have a large number of legal topologies. For example, if there are 10 input ports and 10 output ports from genetic nodes, the total number of topologies is 10×10.

Chapter 5 also proposes to use a sparse crossbar to reduce resource usage, but it only supports a subset of all legal topologies. This subsection tries to solve the problem of the sparse crossbar vs full crossbar by using partial reconfiguration.

To minimise the area required by partial reconfiguration, we extract the communication part in PGP used to provide topology, and isolate it in the topology mapping kernel. We design a topology manager in the CPU to produce a new topology by using the topology algorithm in section 5.4. As shown in Figure 6.11, the topology unit is placed in a partially reconfigurable region, and the topology engine generates the new *topology 1* to perform a run-time reconfiguration.

We test the partially reconfigurable PGP on four problems, including TSP, MAXSAT, F11 and 2DS from section 3.5. Table 6.2 shows the resource usages of PGP with full crossbar and PGP with partial reconfiguration. From the table 6.2, we could see partial reconfiguration reduces the total resource usages, for example, the whole resources for MAXSAT reduce by 8.4% compared with full crossbar. However, PR region may be bigger than resource the genetic nodes need,

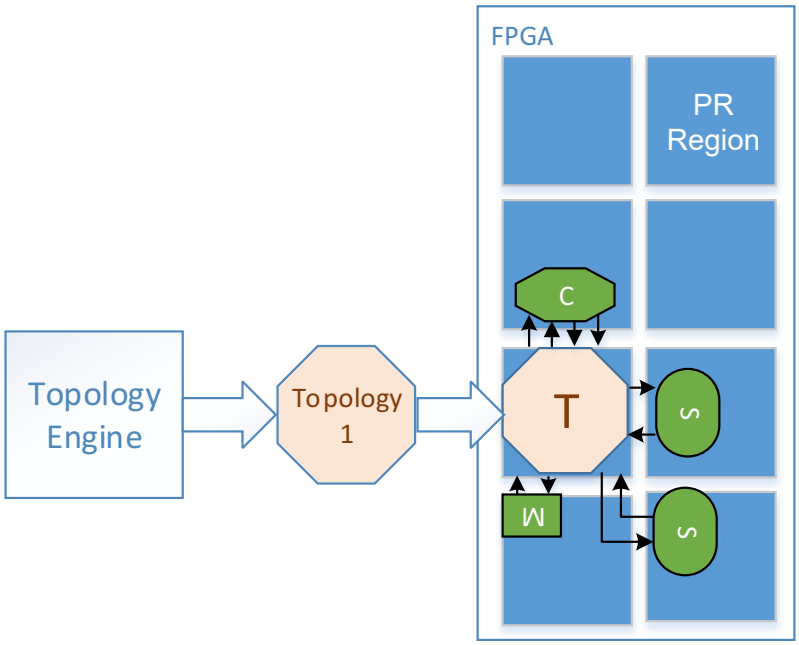


Figure 6.11: Partial Reconfigurable PGP.
Topology engine produces the new Topology 1 to replace the current topology T in a partially reconfigurable region

Table 6.2: Resource Usages of PGP with Full Crossbar and Partial Reconfiguration

Problems #	PGP with Full Crossbar (%)	PGP with PR (%)	Reduced By (%)
TSP	77.17	70.67	8.4
MAXSAT	26.38	22.54	14.56
F11	46.24	38.19	13.44
2DS	38.73	33.45	13.6

so there is an overhead in the resource usage. In addition, the P-PGP systems also loses some performance because of reconfiguration time. The performance reduction depends on the times of reconfiguration, for example, MAXSAT uses partial reconfiguration three times, which increases 40 ms execution time. Users can find a trade-off between resource usage and performance.

6.3.2 Reconfigurable Genetic Nodes

Traditional genetic algorithms introduce diversity by only using genetic operations, such as the crossover and mutation. Once they do not generate new individuals which can survive

in the population, the GAs get stuck in local optima. Our pGA system tunes operators' parameters to change the behaviours of them, and PGP further uses the inter-topology crossing to reconnect the genetic operators, in order to change the evolutionary process. However, when PGP itself cannot produce any improved population after trying many topologies, or the quality of individuals is improving too slowly, the question will be if there exist other ways to solve the problem.

The answer may come from the change of genetic operators available to a PGP instance at run-time. This subsection proposes to introduce high flexibility to genetic nodes in PGP using two methods:

The first is to reselect the sub-set of genetic nodes in the PGP architecture and leave the remaining ones out of the connectivity. For example, assume we have two ordinary mutation nodes, and one volatile mutation node significantly changes individuals so it might be more suitable when PGP gets stuck. Initially, we might not include the volatile one in the PGP architecture, but when the PGP gets stuck and inter-topology change can not help much, we remove the link from an existing mutation in the connection and reconnect the volatile mutation. The two nodes share the same link to other genetic nodes, and we use a multiplexer to choose one path at run-time. As shown in Figure 6.12, it shows the process of 'switch' of two genetic nodes $M1$ and $M1'$. It is also possible to change a crossover node to a mutation node when improvement in fitness stops. This method switches the node at run-time quickly, however, it needs extra hardware resources to store the extra genetic nodes, even though PGP does not use them at the beginning.

The second method is to use partial reconfiguration to replace a genetic node with another. This means we do not need reserved space to place specific genetic nodes used in the late stage, instead we store the configuration on disk. When the current nodes in the FPGA do not help to escape from local optimum, we carry out partial reconfiguration to replace a node in the partial reconfigurable region with the pre-defined one from disk, as shown in Figure 6.13. There is an overhead due to reconfiguration every time, but it is a possible solution for resource limited platforms.

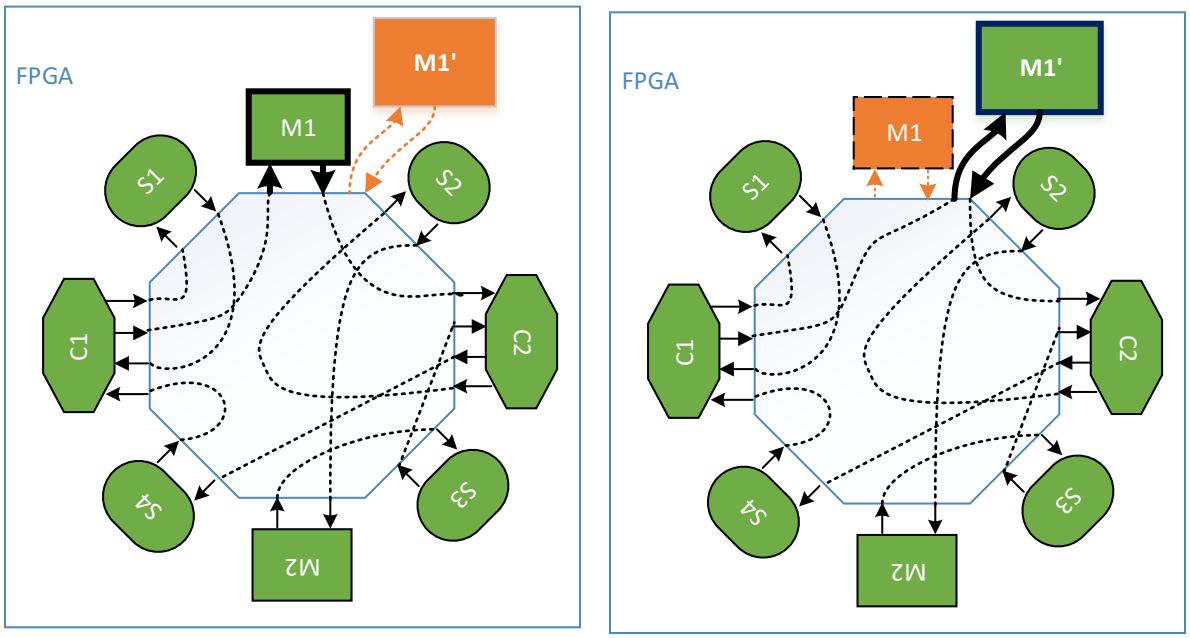


Figure 6.12: Changing Mutation Nodes at Run-time.
PGP architecture initially includes $M1$, but in a later stage removes the connectivity of $M1$ and reconnects $M1'$ to change evolutionary process

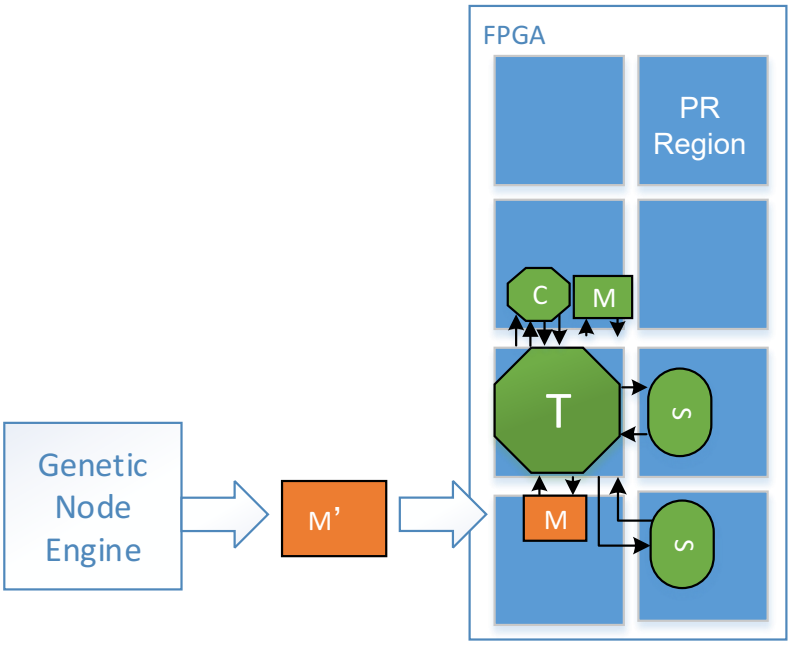


Figure 6.13: Changing Genetic Nodes with Partial Reconfiguration.
Genetic node engine reconfigures the mutation node M to M' using partial reconfiguration

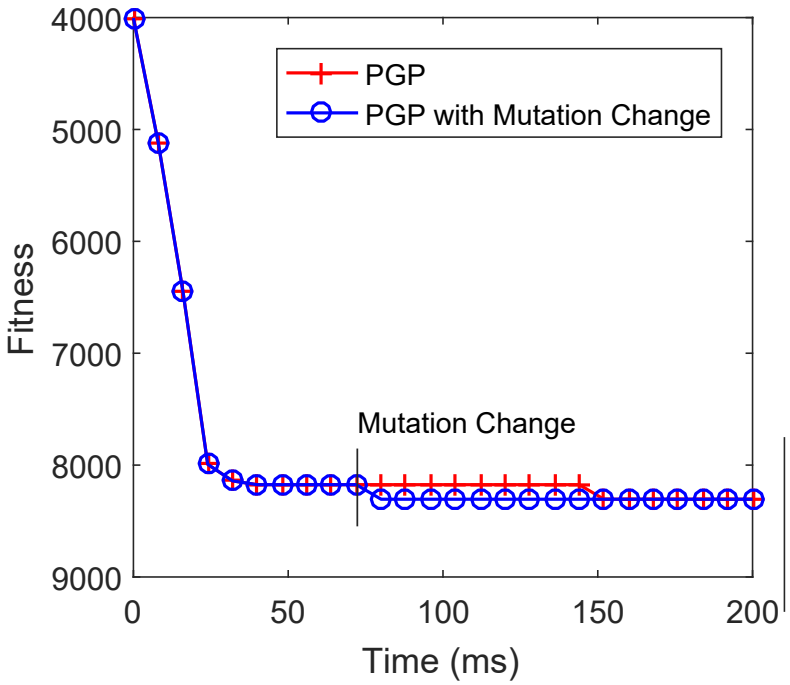


Figure 6.14: BF6 with Partial Reconfiguration.
Changing mutation makes PGP converge to better fitness quickly

6.3.3 Section Summary

The two methods enable the PGP platform to reconfigure genetic nodes, which increases the flexibility of PGP. More experiments could be done, as decribed in chapter 7 to investigate how to choose a good genetic node, or which subset is the best for a problem. Figure 6.14 shows the bf6 example, with the change of the mutation to accelerate the convergence speed. As seen from figure, at a late stage when multiple-point mutation is included, the PGP escapes from local optimum earlier than normal PGP.

6.4 Summary

This chapter extends PGP in three ways, namely solving multiple level optimisation problems by using recursive PGP, guiding PGP to escape from local optima by using machine learning, as well as partially reconfigurable PGP to increase flexibility in communication and genetic nodes. These extensions show good examples for the future users, they could find more ways to extend this flexible platform.

PGP is good at solving optimisation problems shown in Chapter 5 but still not efficient for multiple level optimisation problems. To increase the ability of PGP in solving these problems, this chapter proposes a Recursive PGP. R-PGP has multiple level parallelism, with all levels running concurrently. Experiments on benchmarks show R-PGP achieves high performance for bilevel problems and achieves up to 1200 times speed-up compared to the software solver BLEAQ.

PGP uses inter-topology crossing to allow the evolutionary process to escape from local optima. The random switching scheme presented in last chapter is simple, but machine learning provides a better way when a topology has its own evolutionary quality. This chapter applies a machine learning method called Q-learning to guide the topology change. Results show the performance of Q-PGP is increased by 5% after training.

Genetic nodes introduces the diversity of individuals, and their connections are flexible in PGP. A full crossbar needs large area to supports all connections, so the last chapter uses a sparse crossbar, but only supports a subset of all connections. This chapter proposes to apply partial reconfiguration to both support all possible connections compared with sparse crossbar and reduce resources usages compared with full crossbar. To support further flexibility in genetic nodes, this chapter also proposes to reselect genetic nodes at run-time, to include or remove a genetic node in the PGP architecture. In addition, the chapter also introduces run-time partial reconfiguration to replace genetic nodes in the partially reconfigurable regions, which saves resources.

From the extensions for PGP, we could see that PGP is much more flexible and easier to use. We could apply machine learning, multiple-level parallelism and partial reconfiguration to it. In the next chapter, we will propose more possible directions for the future extensions.

Chapter 7

Conclusion and Future Work

This thesis has presented new directions for genetic algorithms using reconfigurable computing. This thesis first presented an automated tool for creating and executing genetic algorithms on FPGAs with increased programmability. The Generic GA generated not only provides high performance, but also enables users to tune GA and application parameters at run-time, which increases architectural and run-time flexibility. The thesis also presented an extension to the framework with multiple level parallelism, to further increase performance by using multiple GA instances on one FPGA or multiple FPGAs. The thesis then discussed a novel hardware friendly strategy called pipelined genetic propagation, that makes full use of reconfigurable hardware and increase the likelihood of finding the optimum. Finally the thesis showd how to extend PGP with multiple-level parallelism, machine learning, and partial reconfiguration, to maximise flexibility and performance. PGP can be extended further in the future. This chapter summarises the thesis and proposes future work in the following sections:

- Section 7.1 summarises the achievements in the thesis, from the perspectives of both challenges and solutions.
- Section 7.2 proposes possible future work in genetic algorithms on FPGAs, such as adding co-operative evolution in PGP, introducing local optimisation nodes in PGP, and automatically generating PGP architectures using machine learning.

7.1 Summary of Achievements

Genetic algorithms evolve a population of individuals generation by generation, until it reaches desirable solutions to a problem. The evolution process indicates the algorithm need the execution platform to have high performance and support high flexibility. This thesis reviews current hardware solutions for genetic algorithms and analyses the challenges of them. The previous FPGA-based solutions all suffer from one or more challenges: 1) low-level programmability because adapting GA to FPGAs needs a deep understanding of hardware and hardware description language; 2) Lack of architectural flexibility because the previous systems are not general purpose and limit users to tune the architecture to meet the resources constraints; 3) Lack of run-time flexibility, because any modification will result in long-time recompilation and verification, making it impossible to frequently tune GA parameters in FPGAs; 4) Low performance, they have a low speedup over CPU-based systems because pipeline and parallelism are not fully exploited. These observations guided the research in the thesis, and addressing these issues are the main topics of this thesis.

The thesis first develops an general purpose tools to support high-level programmability, architectural flexibility and run-time flexibility. In chapter 3, this thesis presents a framework producing a combined hardware and software system automatically on one FPGA, based on users' high-level inputs without requiring them to learn or use hardware languages. This framework increases the programmability for the users, especially for non-expert users, compared with the previous systems requiring users to write low-level hardware code. Our hardware-based GA is created from a combination of hardware GA template and genetic operators chosen from an operator library, which all have architectural flexibility and pipelined structure. The framework also executes the hardware with run-time parameters, allowing users to tune GA parameters and application parameters without time-consuming recompilation or verification. Even though it uses high-level inputs, the framework still shows an average of 28 times speedup over multiple CPUs and 5 times over a GPU, compared with the work in [FZS10] using low-level inputs but achieving only 6.8 times speedup than a CPU. The thesis also extends the existing framework with multiple level parallelism, making it easy to parallelise a GA within one FPGA and even

across multiple FPGAs.

Parallel GAs are likely to decrease the execution time needed to find high quality solutions, because multiple GA instances evolve from different starting points in the search space and exchange good quality individuals with other GA instances. We add the parallelism features to the framework in chapter 3, and allow the users to configure the parallel GAs in one FPGA or multiple FPGAs easily by using high-level input. The extension also allows users to configure the migration parameters at run-time, saving time when trying different configurations. Finally the chapter gives a suggestion on how to specify migration parameters. The pGAs used in testing are 1.4 times faster than a single kernel GA when finding the desirable solutions on tested optimisation problems. We also observe that the difference in connection topology in pGA brings diversity in evolution, and frequent communication is likely to produce good results. These observations help us to rethink the design of GA system.

Then the thesis focuses on the improvement of performance. After analysing the problems of current solutions, we find the problems of classic GA. The classic GA implementation on FPGAs introduces significant problems: 1) many pipeline stalls because the evolutionary process has to wait for population updates before entering the next generation; 2) memory-access bottleneck because a large centralised memory is used to store all possible candidate solutions; 3) lack of ability to escape from local optima because classical GA brings diversity to population only though the genetic operators.

PGP is proposed to solve these problems: it redesigns a fully pipelined architecture by breaking the generation updating into stages, it reduces the memory-access bottleneck by distributing memory into local storage, it also provides a novel way to escape from local optima by changing the topologies connecting genetic operators at run-time. PGP has both high flexibility and high performance, it shows 5 times faster than parallel GA systems in chapter 4. We also design an automated algorithm to produce good quality topologies.

To further increase the performance, chapter 6 extends PGP with machine learning, multiple level parallelism and partial reconfiguration. Chapter 6 adds multiple level parallelism to support bilevel optimisation problems, showing 1200 times faster than a CPU-based solver. We

observe every topology in PGP has its evolutionary characteristics, so we propose Q-PGP using a machine learning method called Q-learning to guide the topology change. Q-PGP shows a 5% faster than the randomly topology change. We also add partial reconfiguration in PGP to reduce resource usage, as well as increase flexibility in topology unit and genetic nodes.

The thesis addresses the issues of previous systems and reaches objectives proposed in the section 1. The results in the thesis proves FPGA is a promising platform for genetic algorithms: the FPGA-based GA can be fast, easy-to-use and flexible. There are still some space for further research to explore. For example, GA has a group of parameters, setting the values for them is difficult and relays on experience. PGP also can work with different groups of genetic nodes, but the choice of genetic nodes is still set manually. The next section will cover some possible directions of future work, and future researchers can propose new methods based on the work of the thesis.

7.2 Future Work

PGP is a flexible platform, and chapter 6 has already discussed some ways of extending it. Here we propose some possible ideas to further extend PGP: to support large scale problems by co-operative PGP, to increase the convergence by introducing local optimisation nodes, and to generate PGP architectures by an automatic method.

7.2.1 Co-operative PGP

In genetic algorithms, a candidate solution to a problem is encoded as a chromosome, which may contain a set of parameters (genes). PGP supports a large number of parameters in a chromosome, for example, chapter 5 presented a chromosome with 100 parameters for MAXSAT problems. However, when the size of a chromosome is too large for an FPGA, we need a new way to support this kind of problem.

Researchers have used a co-operative genetic algorithm to solve this problem. Co-operative

GA uses multiple GA instances to solve one problem, similar to pGA. But the big difference between co-operative GAs and pGAs is that sub-populations in the co-operative GA have different chromosomes, so a co-operative GA cannot support direct migration as used in pGA. [dlCR98] proposed co-operative genetic algorithms to solve a robotic path planning problem because the complete search space was large. That paper split the whole population into sub-populations and every sub population had its own chromosome specie. So a GA instance has to co-operate with all other instances to evolve the complete solutions to the problem, because each of the GA instances only solves a sub-set of the problem. The co-operative GA has also been used to solve other problems, for example a 3D container loading problem. [PC01] partitioned a 205 element loading sequence into a number of shorter sequences, reducing execution time by 30%. To solve large scale problems in PGP, we propose co-operative PGP.

To support the co-operative method in PGP, we have several considerations:

- No conflict in different type individuals. If two different individuals are mixed in a pipelined structure, we must distinguish them. We could learn from the previous experience of R-PGP because it already supports different chromosomes evolving in chapter 6. In R-PGP, the lower-level individuals are for lower-level optimisation, while the upper-level individuals are for the high-level optimisations. Because the lower-level PGP is embedded in the upper-level PGP, the two types of chromosomes do not have any conflict. Here we could put PGP in different FPGAs, or use a flag label to class a chromosome in one FPGA.

- Cooperative fitness evaluation. Because every individual is part of a complete candidate solution to the problem, they need to cooperate to make sure the total fitness is the largest.

- Minimising resource usage. The resources in an FPGA are limited, but we must use the resources for both genetic nodes and links. If the chromosome is too long, it means the topology links between genetic nodes become large, which will occupy a large number of resources. We have to partition the problem into a platform-specific size to balance resource usage.

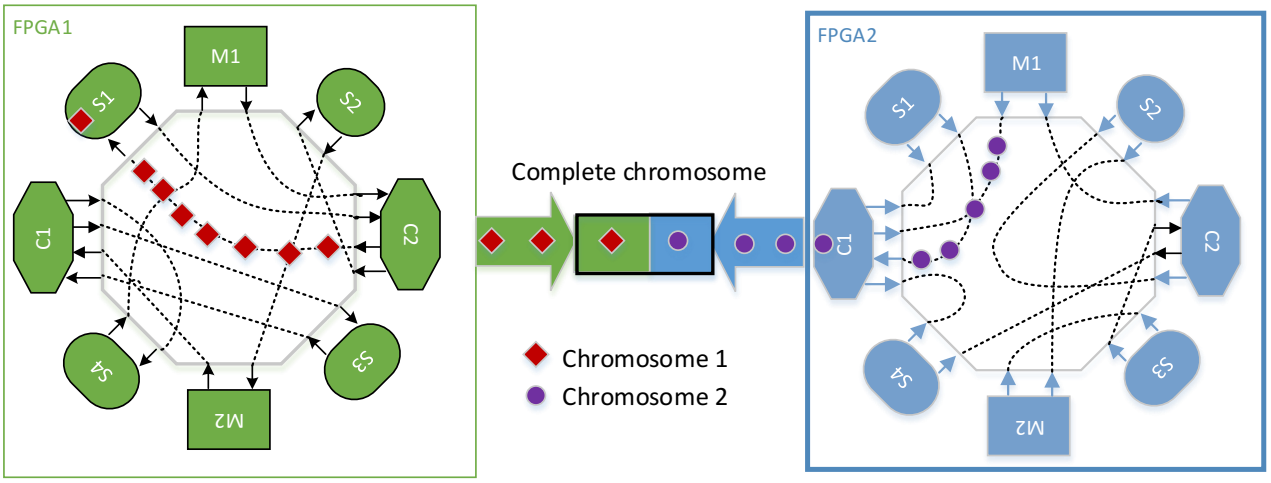


Figure 7.1: Co-operative PGP on Multiple FPGAs.
PGP1 on FPGA1 evolves chromosome1, while PGP2 on FPGA2 evolves chromosome2. The complete chromosome is made up of chromosome1 and chromosome2

We propose two approaches for co-operative PGP. The first approach is a simple and straightforward method, which partitions PGP into multiple FPGAs. Every FPGA carries a PGP, which evolves its own chromosome. As shown in Figure 7.1, PGP1 on FPGA1 evolves chromosome1, while PGP2 on FPGA2 evolves chromosome2. The complete chromosome is made up of chromosome1 and chromosome2 which are calculated independently, but the whole fitness can be calculated in another machine if needed.

The second way is to use only one PGP to support multiple chromosomes. As shown in Figure 7.2, two types of chromosomes flow though the connection and genetic nodes. It reduces the size of the link between genetic nodes because the size is the maximal size of all chromosomes, rather than a sum of all chromosomes. To allow a genetic node to support several types chromosomes, we will need to add a flag in the chromosome to distinguish the type. We also need to modify all genetic nodes to perform different operations according to the flag of the chromosome.

7.2.2 Local Optimisation in PGP

Genetic algorithms are a heuristic for finding global optima, using a stochastic search method. However, for problems with many local optima, GAs sometimes get stuck. Researchers have proposed different ways to escape from them, one way is to combine GA with a local opti-

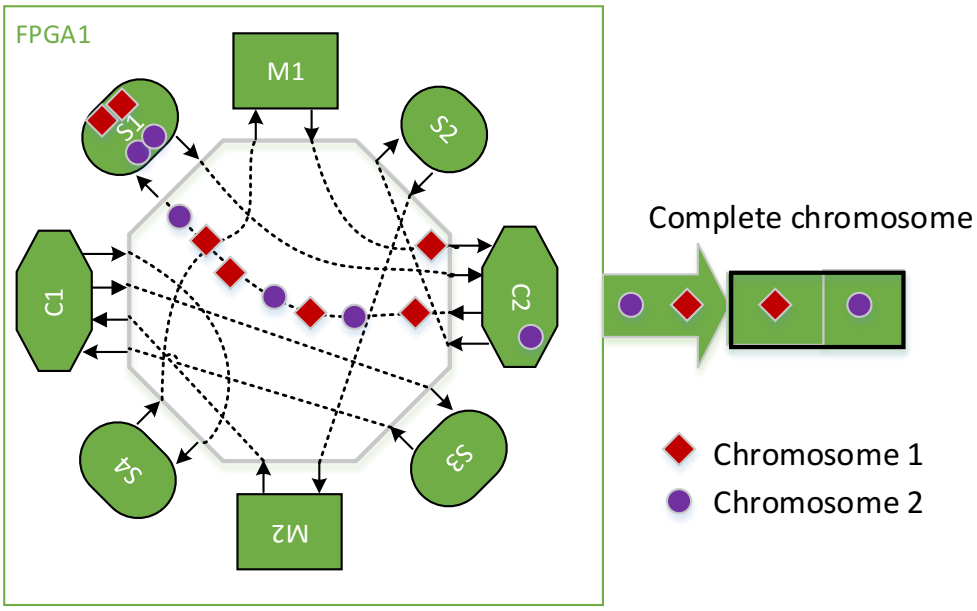


Figure 7.2: Co-operative PGP on One FPGA.
One FPGA evolves both chromosome1 and chromosome2, which contain a flag to indicate the class

misation method. For example, [Pál95] proposed use of a hybrid system combining GA and local optimisation to test a massively multi-modal spin-lattice problem with a huge configuration space. [BH14] also applied two local optimisation strategies to GA for travelling salesman problems, showing 30% improvement on the solution compared with a classical GA. We think it is a good addition to our PGP system, and the local optimisation can be built as a node connected to other genetic nodes.

PGP already has an operator library which contains different genetic operators for chromosomes. We could put some pre-defined local optimisation operators in the library. The local optimisation node can have an internal memory and contain an evaluator, which means it becomes a non-blind node. As shown in Figure 7.3, the genetic operators work with other genetic nodes to solve a problem.

Another consideration for a local optimisation node is its pipelined structure, because in the PGP strategy, all nodes receive and send individuals in every cycle. If the local operator cannot have a fully pipelined structure, it cannot output a result every cycle. The solution is that we add a label indicating the status of the individuals as 'invalid' or 'valid'. As shown in Figure 7.3, the blind node could receive or send an 'invalid' node because it has to send out anything

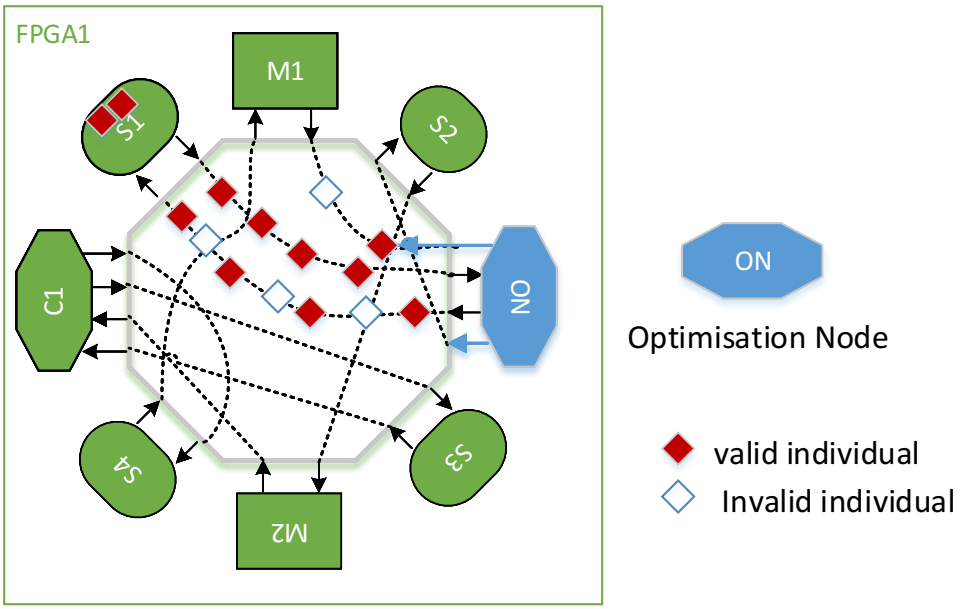


Figure 7.3: PGP with Local Optimisation Node.
If an optimisation node is not fully pipelined, we label the individual as 'valid' or 'invalid' using a flag. Non-blind nodes such as the selection node (S1) will drop the 'invalid' chromosomes and output the 'valid' ones

it receives. But when a non-blind node such as selection node receives an 'invalid' node, it will drop it and send out a valid one stored in the internal memory. In this way, we can support pipelined or non-pipelined optimisation node.

7.2.3 Automatic PGP Architecture Generation

PGP is a flexible platform, which contains flexible genetic nodes and flexible links between these nodes. However, it needs the users to choose which genetic nodes to use, or which type of genetic node should be included in the PGP architecture. Even though we present a topology generator in chapter 5 to automatically create topologies based on the selected genetic nodes, it still needs the users to choose which genetic nodes are used. Here we propose a system to automatically generate PGP graph without any work from users.

We propose to use an automatic generation system to create a PGP architecture. As shown in Figure 7.4, the system randomly generates a configuration for a number of genetic nodes from a problem independent library. To make full use of resources in reconfigurable hardware, we use

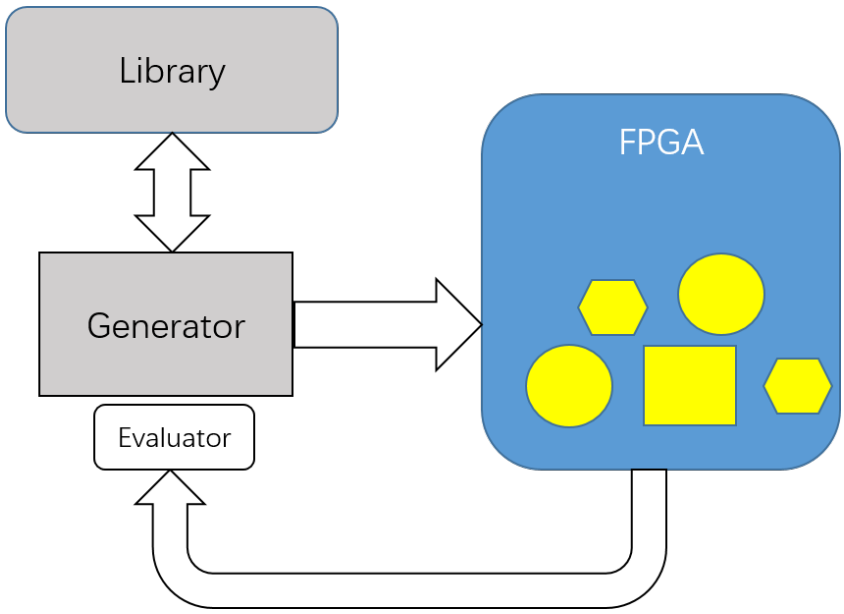


Figure 7.4: Automatic PGP Generation System.
Generator uses the genetic nodes in the library to fill an FPGA, then evaluates the produced PGP architecture and adjusts it based on the PGP performance

algorithms to choose genetic nodes repeatedly to fill an FPGA, which is similar to a packing problem. After that, we evaluate the PGP generated, and give a reward to the system based on its convergence rate. We could modify the configuration based on the fitness, and could use machine learning or PGP itself to evolve the PGP architecture.

7.3 Summary

This chapter first examines the work that has been done in the thesis. First, the thesis analyses the previous work and identifies the challenges of the genetic algorithm in reconfigurable computing, including the lack of flexibility and low performance. Then, this thesis addresses these problems by designing an automated framework reading high level inputs, a flexible GA kernel customisable to users, a novel fully pipelined genetic approach and its extensions. It presents more flexible and faster systems to the researchers in the field.

The chapter also suggests potential research directions for the future. It proposes a co-operative PGP for large scale problem, introduces local optimisation solvers into the PGP to increase performance, and gives advices to use machine learning to automatically generate PGP archi-

tectures. This chapter presents new thoughts about the genetic algorithm in reconfigurable computing, and shows examples of how to extend the work of the thesis in the future. In sum, PGP is not only a genetic-based problem solver, it is also a research tool for evolutionary computing.

Bibliography

[AC01] Chatchawit Aporntewan and Prabhas Chongstitvatana. A hardware implementation of the compact genetic algorithm. In *IEEE Congress on Evolutionary Computation*, pages 624–629. Citeseer, 2001.

[AST09] Carlos Ansótegui, Meinolf Sellmann, and Kevin Tierney. A gender-based genetic algorithm for the automatic configuration of algorithms. In *International Conference on Principles and Practice of Constraint Programming*, pages 142–157. Springer, 2009.

[AT99] Enrique Alba and José M Troya. A survey of parallel distributed genetic algorithms. *Complexity*, 4(4):31–52, 1999.

[ATLF08] François CJ Allaire, Mohamed Tarbouchi, Gilles Labonté, and Giovanni Fusina. FPGA implementation of genetic algorithm for uav real-time path planning. In *Unmanned Aircraft Systems*, pages 495–510. Springer, 2008.

[AV11] Enrique Alba and Pablo Vidal. Systolic optimization on GPU platforms. In *Computer Aided Systems Theory–EUROCAST 2011*, pages 375–383. Springer, 2011.

[Bai15] Donald G. Bailey. The advantages and limitations of high level synthesis for FPGA based image processing. In *Proceedings of the 9th International Conference on Distributed Smart Cameras*, ICDSC '15, pages 134–139, New York, NY, USA, 2015. ACM.

[Bal82] E. Balas. *A class of location, distribution and scheduling problems: Modeling and solution methods*. 1982.

[Bar83] Jonathan F Bard. Coordination of a multidivisional organization through two levels of management. *Omega*, 11(5):457–468, 1983.

[Bar98] Jonathan F Bard. *Practical bilevel optimization: algorithms and applications*, volume 30. Springer Science & Business Media, 1998.

[BB04] Jean Berger and Mohamed Barkaoui. A parallel hybrid genetic algorithm for the vehicle routing problem with time windows. *Computers & operations research*, 31(12):2037–2053, 2004.

[Bel95] Theodore C Belding. The distributed genetic algorithm revisited. *arXiv preprint adap-org/9504007*, 1995.

[Bet76] Albert Donally Bethke. Comparison of genetic algorithms and gradient-based optimizers on parallel processors: Efficiency of use of processing capacity. 1976.

[BFGS03] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröoder. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924, July 2003.

[BFM97] Thomas Bäck, DB Fogel, and Z Michalewicz. Handbook of evolutionary computation. *Release*, 97(1):B1, 1997.

[BH14] Keivan Borna and Vahid Haji Hashemi. An improved genetic algorithm with a local optimization strategy and an extra mutation level for solving traveling salesman problem. *arXiv preprint arXiv:1409.3078*, 2014.

[BLMS01] Luce Brotcorne, Martine Labbé, Patrice Marcotte, and Gilles Savard. A bilevel model for toll optimization on a multicommodity transportation network. *Transportation Science*, 35(4):345–358, 2001.

[CC00] Y. Choi and D. J. Chung. VLSI processor of parallel genetic algorithm. *IEEE Asia Pacific Conference on ASIC*, pages 143–146, 2000.

[CDF+86] William Carter, Khue Duong, Ross H Freeman, H Hsieh, Jason Y Ja, John E Mahoney, Luan T Ngo, and Shelly L Sze. A user programmable reconfigurable gate array. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, 1986.

[CDW10] João MP Cardoso, Pedro C Diniz, and Markus Weinhardt. Compiling for reconfigurable computing: A survey. *ACM Computing Surveys (CSUR)*, 42(4):13, 2010.

[CH02] K. Compton and S. Hauck. Reconfigurable computing: A survey of systems and software. *ACM Computing Surveys*, 34(2):171–210, 2002.

[Col03] D. A. Coley. *An introduction to genetic algorithms for scientists and engineers*. World Scientific Publishing, Singapore, 2003.

[CP97] Erick Cantu-Paz. Designing efficient master-slave parallel genetic algorithms, 1997.

[CP98] Erick Cantú-Paz. A survey of parallel genetic algorithms. *Calculateurs paralleles, reseaux et systems repartis*, 10(2):141–171, 1998.

[CP00] Erick Cantu-Paz. *Efficient and accurate parallel genetic algorithms*, volume 1. Springer Science & Business Media, 2000.

[DeJ75] Kenneth DeJong. An analysis of the behavior of a class of genetic adaptive systems. *Ph. D. Thesis, University of Michigan*, 1975.

[Des12] Xilinx Design, Model-Based DSP. Vivado design suite user guide. 2012.

[DKT+95] Thomas H Drayer, W Ely King, Joeseph G Tront, Richard W Conners, and Philip A Araman. Using multiple FPGA architectures for real-time processing of low-level machine vision functions. In *Proceedings of IECON 21st International Conference on Industrial Electronics, Control, and Instrumentation*, volume 2, pages 1284–1289. IEEE, 1995.

[dlCR98] Victor de la Cueva and Fernando Ramos. Cooperative genetic algorithms: a new approach to solve the path planning problem for cooperative robotic manipulators sharing the same work space. In *Intelligent Robots and Systems, 1998. Proceedings., 1998 IEEE/RSJ International Conference on*, volume 1, pages 267–272. IEEE, 1998.

[EHM99] Ágoston E Eiben, Robert Hinterding, and Zbigniew Michalewicz. Parameter control in evolutionary algorithms. *IEEE Transactions on evolutionary computation*, 3(2):124–141, 1999.

[Est60] Gerald Estrin. Organization of computer systems: the fixed plus variable structure computer. In *Papers presented at the May 3-5, 1960, western joint IRE-AIEE-ACM computer conference*, pages 33–40. ACM, 1960.

[FSK+08] Pradeep Fernando, Hariharan Sankaran, Srinivas Katkoori, Didier Keymeulen, Adrian Stoica, Ricardo Zebulum, and Ramesham Rajeshuni. A customizable FPGA IP core implementation of a general purpose genetic algorithm engine. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–8. IEEE, 2008.

[FZS10] P. R. Fernando, R. Zebulum, and A. Stoica. Customizable FPGA IP core implementation of a general-purpose genetic algorithm engine. *IEEE Transactions on Evolutionary Computation*, 14(1):139–149, 2010.

[GL13] Fred Glover and Manuel Laguna. *Tabu Search*. Springer, 2013.

[GN95] Paul Graham and Brent Nelson. A hardware genetic algorithm for the traveling salesman problem on Splash 2. In *International Workshop on Field Programmable Logic and Applications*, pages 352–361. Springer, 1995.

[Gol89] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[GPG10] Janice Gaffney, Charles Pearce, and David Green. Binary versus real coding for genetic algorithms: A false dichotomy? *ANZIAM Journal*, 51:347–359, 2010.

[HH95] JD Hadley and Brad L Hutchings. Designing a partially reconfigured system. In *Photonics East'95*, pages 210–220. International Society for Optics and Photonics, 1995.

[HH04] R. L. Haupt and S. E. Haupt. *Practical genetic algorithms*. John Wiley & Sons, 2004.

[Hol75] John H Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press, 1975.

[JC08] Y. Jewajinda and P. Chongstitvatana. FPGA implementation of a cellular compact genetic algorithm. *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 385–390, 2008.

[JKFA06] M. S. Jelodar, M. Kamal, S.M. Fakhraie, and M.N. Ahmadabadi. SOPC-based parallel genetic algorithm. *IEEE Congress on Evolutionary Computation*, pages 2800–2806, 2006.

[Kao05] Cindy Kao. Benefits of partial reconfiguration. *Xcell journal*, 55:65–67, 2005.

[KK12] T. Kamimura and A. Kanasugi. A parallel processor for distributed genetic algorithm with redundant binary number. *6th International Conference on New Trends in Information Science and Service Science and Data Mining (ISSDM)*, pages 125–128, 2012.

[KLM96] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[KNPDK98] C Kirjner-Neto, E Polak, and A Der Kiureghian. An outer approximations approach to reliability-based optimal design of structures. *Journal of optimization theory and applications*, 98(1):1–16, 1998.

[KS09] Schalk Kok and Carl Sandrock. Locating and characterizing the stationary points of the extended rosenbrock function. *Evolutionary Computation*, 17(3):437–453, 2009.

[KTR08] Ian Kuon, Russell Tessier, and Jonathan Rose. FPGA architecture: Survey and challenges. *Foundations and Trends in Electronic Design Automation*, 2(2):135–253, 2008.

[LA11] G. Luque and E. Alba. Parallel genetic algorithms: Theory and real world applications. *Springer*, 367, 2011.

[LPG94] Shyh-Chang Lin, W. F. Punch, and E. D. Goodman. Coarse-grain parallel genetic algorithms: categorization and new approach. In *Parallel and Distributed Processing, 1994. Proceedings. Sixth IEEE Symposium on*, pages 28–37, Oct 1994.

[MEA05] Syamsiah Mashohor, Jonathan R Evans, and Tughrul Arslan. Elitist selection schemes for genetic algorithm based printed circuit board inspection system. In *2005 IEEE Congress on Evolutionary Computation*, volume 2, pages 974–978. IEEE, 2005.

[MG95] Brad L Miller and David E Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Complex systems*, 9(3):193–212, 1995.

[MWMA09] Asim Munawar, Mohamed Wahib, Masaharu Munetomo, and Kiyoshi Akama. Hybrid of genetic algorithm and local search to solve MAX-SAT problem using nVidia CUDA framework. *Genetic Programming and Evolvable Machines*, 10(4):391–415, 2009.

[Nag06] Yuichi Nagata. New eax crossover for large TSP instances. In *Parallel Problem Solving from Nature-PPSN IX*, pages 372–381. Springer, 2006.

[NDL+09] Antonio J Nebro, Juan J Durillo, Francisco Luna, Bernabé Dorronsoro, and Enrique Alba. Mocell: A cellular genetic algorithm for multiobjective optimization. *International Journal of Intelligent Systems*, 24(7):726–746, 2009.

[NS05] John Newborough and Susan Stepney. A generic framework for population-based algorithms, implemented on multiple FPGAs. In *Artificial Immune Systems*, pages 43–55. Springer, 2005.

[OHL+08] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[PA12] Oliver Pell and Vitali Averbukh. Maximum performance computing with dataflow engines. *Computing in Science & Engineering*, 14(4):98–103, 2012.

[Pál95] Károly F Pál. Genetic algorithm with local optimization. *Biological Cybernetics*, 73(4):335–341, 1995.

[PAL12] Martin Pedemonte, Enrique Alba, and Francisco Luna. Towards the design of systolic genetic search. In *26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, pages 1778–1786. IEEE, 2012.

[PC01] C. Pimpawat and N. Chaiyaratana. Using a co-operative co-evolutionary genetic algorithm to solve a three-dimensional container loading problem. In *Evolutionary Computation, 2001. Proceedings of the 2001 Congress on*, volume 2, pages 1197–1204 vol. 2, 2001.

[Pit95] Laurens Jan Pit. Parallel genetic algorithms. *Department of Computer Science, Master Thesis, Leiden University, Holanda*, 1995.

[PP02] C. Plessl and M. Platzner. Custom computing machines for the set covering problem. *Proceedings of 10th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 163–172, 2002.

[PS82] Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1982.

[Rot11] Franz Rothlauf. *Design of modern heuristics: principles and application*. Springer Science & Business Media, 2011.

[SAF13] P. V. D. Santos, J. C. Alves, and J. C. Ferreira. A framework for hardware cellular genetic algorithms: an application to spectrum allocation in cognitive radio. *23rd International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, 2013.

[SD07] SN Sivanandam and SN Deepa. *Introduction to genetic algorithms*. Springer Science & Business Media, 2007.

[Sha92] John A Sharp. *Data flow computing: theory and practice*. Intellect Books, 1992.

[SMD12] A. Sinha, P. Malo, and K. Deb. Unconstrained scalable test problems for single-objective bilevel optimization. In *2012 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8, 2012.

[SMD13] Ankur Sinha, Pekka Malo, and Kalyanmoy Deb. Efficient evolutionary algorithm for single-objective bilevel optimization. *arXiv preprint arXiv:1303.3901*, 2013.

[SMD14] Ankur Sinha, Pekka Malo, and Kalyanmoy Deb. Test problem construction for single-objective bilevel optimization. *Evolutionary computation*, 22(3):439–477, 2014.

[SMEH14] Gamal Abd El-Nasser A Said, Abeer M Mahmoud, and El-Sayed M El-Horbaty. A comparative study of meta-heuristic algorithms for solving quadratic assignment problem. *arXiv preprint arXiv:1407.4863*, 2014.

[SMFD13] A. Sinha, P. Malo, A. Frantsev, and K. Deb. Multi-objective stackelberg game between a regulating authority and a mining company: A case study in environmental economics. In *2013 IEEE Congress on Evolutionary Computation (CEC)*, pages 478–485, 2013.

[SSC+01] Barry Shackleford, Greg Snider, Richard J Carter, Etsuko Okushi, Mitsuhiro Yasuda, Katsuhiko Seo, and Hiroto Yasuura. A high-performance, pipelined, FPGA-based genetic algorithm machine. *Genetic Programming and Evolvable Machines*, 2(1):33–60, 2001.

[SSS95] Stephen D Scott, Ashok Samal, and Shared Seth. HGA: A hardware-based genetic algorithm. In *Proceedings of the 1995 ACM third International Symposium on Field-Programmable Gate Arrays*, pages 53–59. ACM, 1995.

[SWSS95] N. Sitkoff, M. Wazlowski, A. Smith, and H. Silverman. Implementing a genetic algorithm on a parallel custom computing machine. In *IEEE Symposium on FPGAs for Custom Computing Machines, 1995. Proceedings*, pages 180–187, Apr 1995.

[Sys91] Gilbert Syswerda. A study of reproduction in generational and steady state genetic algorithms. *Foundations of genetic algorithms*, 2:94–101, 1991.

[TCW+05] Timothy J Todman, George A Constantinides, Steven JE Wilton, Oscar Mencer, Wayne Luk, and Peter YK Cheung. Reconfigurable computing: architectures and design methods. In *IEEE Proceedings on Computers and Digital Techniques*, volume 152, pages 193–207. IET, 2005.

[Tec13] Maxeler Tech. Programming mpc systems white paper, 2013.

[TG94] Dirk Thierens and David Goldberg. Elitist recombination: An integrated selection recombination GA. In *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, pages 508–512. IEEE, 1994.

[THL09] David Barrie Thomas, Lee Howes, and Wayne Luk. A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 63–72. ACM, 2009.

[TL07] D. B. Thomas and Wayne Luk. High quality uniform random number generation using lut optimised state-transition matrices. *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 47(1):77–92, 2007.

[TMSY06] T. Tachibana, Y. Murata, N. Shibata, and K. Yasumoto. General architecture

for hardware implementation of genetic algorithm. *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 291–292, 2006.

[TY04] W. Tang and L. Yip. Hardware implementation of genetic algorithms using FPGA. *47th IEEE Midwest Symposium on Circuits and Systems*, pages 549–552, 2004.

[VPP09] Michalis Vavouras, Kyprianos Papadimitriou, and Ioannis Papaefstathiou. High-speed FPGA-based implementations of a genetic algorithm. In *International Symposium on Systems, Architectures, Modeling, and Simulation. SAMOS'09.*, pages 9–16, 2009.

[Wal96] Matthew Wall. Galib: A C++ library of genetic algorithm components. *Mechanical Engineering Department, Massachusetts Institute of Technology*, 87:54, 1996.

[Wat89] Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards*. PhD thesis, University of Cambridge, England, 1989.

[WBT06] R Wisniewski, AA Barkalov, and L Titarenko. Synthesis of compositional microprogram control units with sharing codes and address decoder. In *Proceedings of the International Conference Mixed Design of Integrated Circuits and System, 2006. MIXDES 2006.*, 2006.

[WCT94] Shin-Shu Wang, Tzu-Chiang Chung, and Y. T. Tsai. Implementing a digital signal processor for an electronic still camera using multiple-FPGAs. In *ASIC Conference and Exhibit, 1994. Proceedings., Seventh Annual IEEE International*, pages 258–260, Sep 1994.

[YY99] N. Yoshida and T. Yasuoka. Multi-GAP: parallel and distributed genetic algorithms in VLSI. *Systems, Man, and Cybernetics*, 5:571–576, 1999.

[ZSHD02] Heidi Ziegler, Byoungro So, Mary Hall, and Pedro C Diniz. Coarse-grain pipelining on multiple FPGA architectures. In *10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2002. Proceedings*, pages 77–86. IEEE, 2002.