High Level Synthesis Using Genetic Algorithm

Andrew Zakhary
Electronics Engineering
Hochschule Hamm-Lippstadt
Hamm, Germany
2210009

***Abstract*—With electronic components getting smaller and smaller the complexity of the circuitry is growing at an exponential rate. With this came many approaches in simplifying and streamlining the hardware design process. One of these approaches is hardware-software co-design where hardware and software design are interlinked. During this process the software is designed in a way where the hardware can be generated directly from the software design where this is done in formal models or in programming language as SystemC. This approach opened the door to a wide range of algorithms that try to break down the software design and turn it into hardware circuitry in a process called synthesis. In this paper one of these algorithms, namely Genetic Algorithm, is discussed in details and given an example that explains its workings.**

***Index Terms*—Hardware, Synthesis, hardware software co-design, Genetic algorithm**

I. Introduction

A. Genetic Algorithm

Over the years, biology has inspired technology in many different ways. After all, life has found multitude of ways to exist and persist over billions of years in an always changing environmental conditions. Also, as resources in nature are limited there's a constant competition on them, out of this competition arises a constant need for improvement and efficiency. One of the aspects of nature that has been used extensively in the tech sector is the Genetic Algorithm (GA). GA in nature is performed by two main processes, one is evolution and the second is natural selection. The evolution step is responsible for creating the different variations through different process as :

- **Cross-Over** This is process where the two sets of chromosomes (i.e traits) of the parents and mixed and matched to generate a single set of chromosomes that would be inherited by their offspring.
- **Mutation** This is a process where the chromosomes (traits) of the offspring are changed randomly.

these two processes work together to make sure that there's always a replenishing new set of traits being generated in the genetic pool.

These individuals are then governed by the law of natural selection. As the resources available are limited, this law describes how the survival of the offsprings is always contingent on them being more fit for their environment. With the creation processes mentioned before this inevitably leads to the permeation of specific traits. These traits are mostly only changed with the change the environment.

B. High Level Synthesis

High level synthesis is a method of generating digital circuits automatically from a behavioral model. This behavioural model could describe a mathematical equation, the movement of a robot or a home appliance. the output of such process is split into four different categories [1]

- **Functional units (FUs)** these are responsible for implementing the logical functions.
- **Registers and memory units** these are the spaces where the data would be stored.
- **Switchable connections** these the busses and multiplexers that connect the memory units with functional units.
- **Controller** this is is the unit that is responsible for switching the switchable connections.

During the process of High level synthesis a lot of constraints have to managed. Some are fixed like number of pins, space available or time required for each operation while some others need optimization like power consumption and costs. Due to the complexity of the synthesis process process which arises because of the complexity of the model, multiple scheduling algorithms have been developed that try to optimize different resources [2] like reducing the number of FUs or registers or reducing power consumption among many others. Since optimizing all factors at the same time is nearly impossible, algorithms must prioritize some over others. For example an algorithm can prioritize execution time and therefore may decide to sacrifice power consumption.

II. Implementation

A. Methodology

As discussed the algorithm in HLS is a search algorithm that is searching for the optimized

Genetic algorithm in this regard a more flexible solution as the priority of the fitness function can be changed to serve different functions. The way genetic algorithms are implemented usually is with population containing individuals that are created randomly. This population is then graded on a specific fitness function and the top performing individuals are selected for the next phase. The algorithm then creates a new population based on mixing "traits" of the successful ones to try to improve the overall fitness of the population from one generation to another. Implementing GA in HLS, similar to other HLS algorithms start by a system

B. Example

Assume a system of differential equations described by the following HDL code :

```
module diffEqn (x, dx, u, a, clock, y)
input x, dx, u, a, clock;
output y;
always @(posedge clock)
    while (x<a)
        begin
        x1=x+dx;
        u1=u-(3*x*u*dx)-(3*y*dx);
        y1=y+(u*dx);
        x=x1;
        u=u1;
        y=y1;
        end
endmodule
```

A control data flow graph (CDFG) can be generated for the process as shown in Fig.1. This shows the different operations (multuplications,comparisons, addition and subtraction). The inputs could be removed from the top layer, and processes given names in order to make it more readable as shown if Fig.2

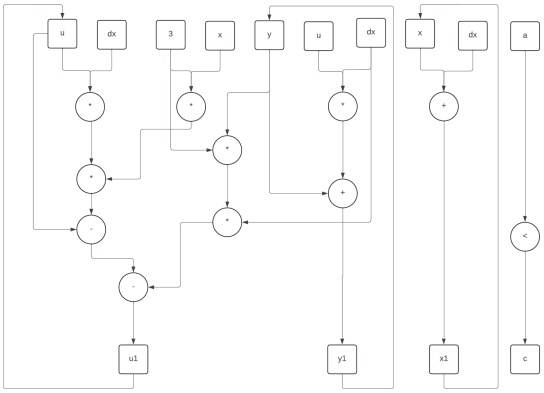

Fig. 1. CDFG for second order differential equation

As can be visualized from Fig.2, some processes need other processes to be completed before they can be run themselves. This condition is called *Dependency* and it is very important in hardware synthesis. As multiple solutions will be proposed in the upcoming section will be presented, it is very important to be able to check if the dependencies are met or not. For this TableI is created to easily check if dependencies are met.

To better visualize the power of genetic algorithm as searching algorithm in such a simple example it's wise to constrain the resources of the hardware to be designed in order to get a wider variety of the population generated. For this example a constraint of only 3 ALUs are assumed. This means since our focus is on the mathematical operations in the code, the solutions are limited to three operations in parallel in each clock cycle. In this example other operations (e.g reading from memory, writing to memory, sending control signals,.. etc) are not calculated as they are assumed similar in all proposed solutions. In practice this might not be the case depending on the program to be designed in hardware.

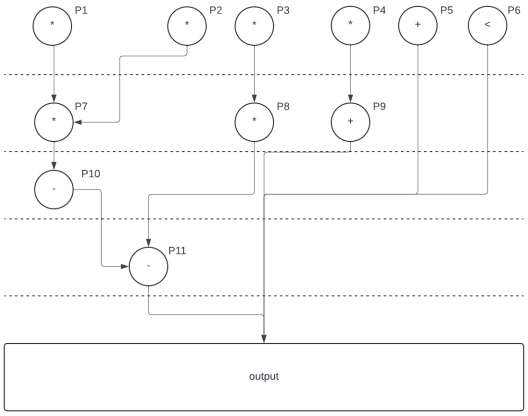

Fig. 2. Cleaned CDFG diagram

Dependent process	Dependent on
$P7$	$P1, P2$
$P8$	$P3$
$P9$	$P4$
$P10$	$P7$
$P11$	$P10, P8$

TABLE I
DEPENDENCIES TABLE

The fitness formula used in this example is proposed to be the utilization of the 3 ALUs available during the program run time as shown in the equation (1). As every process is done on a single ALU, it is assumed each process only takes one clock cycle. This makes the total runtime of the ALUs fixed to 11 units of time the program has 11 processes.

$$\rho = \Sigma \frac{T_{ALU}}{T_{Total}} = \frac{11}{T_{Total}} \tag{1}$$

With this explained the creation of the first generation can start. Each Individual will be created in a separate table and their fitness will be compared afterwards.

ALU #	T_1	T_2	T_3	T_4	T_5	T_6
ALU #1	P4	P2	P1	P7	P10	P11
ALU #2	P5	P3	P8			
ALU #3	P6	P9				

TABLE II
INDIVIDUAL #1

ALU #	T_1	T_2	T_3	T_4	T_5	T_6
ALU #1	P3	P8	P1	P7	P10	P11
ALU #2	P4	P9	P2			
ALU #3	P5	P6				

TABLE III
INDIVIDUAL #2

ALU #	T_1	T_2	T_3	T_4	T_5
ALU #1	P1	P7	P3	P8	P11
ALU #2	P2	P9	P6		
ALU #3	P4	P5	P10		

TABLE IV
INDIVIDUAL #3

ALU #	T_1	T_2	T_3	T_4	T_5
ALU #1	P1	P7	P9	P8	P11
ALU #2	P2	P4	P6		
ALU #3	P3	P5	P10		

TABLE V
INDIVIDUAL #4

Individual	#1	#2	#3	#4
Fitness	61.11%	61.11%	73.33%	73.33%

TABLE VI
FITNESS COMPARISON

By comparing the first generation that was randomly created, the highest rated individuals can be selected in order to create the second generation's offspring. According to tableVI Individuals #3 and #4 are selected. In order to do so, it is necessary to see where slices and matches can be done. This can be visualized by color coding the individuals' similarities and dissimilarities. In the tables below the similarities are highlighted by the green color, while the differences are highlighted by the orange and blue colors for individuals #3 and #4 respectively as shown in Tables VII and VIII.

ALU #	T_1	T_2	T_3	T_4	T_5
ALU #1	P1	P7	P3	P8	P11
ALU #2	P2	P9	P6		
ALU #3	P4	P5	P10		

TABLE VII
INDIVIDUAL #3

ALU #	T_1	T_2	T_3	T_4	T_5
ALU #1	P1	P7	P9	P8	P11
ALU #2	P2	P4	P6		
ALU #3	P3	P5	P10		

TABLE VIII
INDIVIDUAL #4

Using these two individuals as parents to the next generation, the Cross-over process can simulated by randomly swapping the differences while keeping the similarities the same. In this example two offsprings are generated by swapping the differences one at a time as shown in Tables IX and X.

ALU #	T_1	T_2	T_3	T_4	T_5
ALU #1	P1	P7	P9	P8	P11
ALU #2	P2	P3	P6		
ALU #3	P4	P5	P10		

TABLE IX
OFFSPRING #1

By analyzing the two offsprings, it is noticed that the the fitness stays the same at $\frac{11}{15}$ or 73.33%. By analyzing the

ALU #	T_1	T_2	T_3	T_4	T_5
ALU #1	P1	P7	P4	P8	P11
ALU #2	P2	P9	P6		
ALU #3	P3	P5	P10		

TABLE X
OFFSPRING #2

dependencies mentioned in Table I however, it is noticed that Offspring #2 has the process P9 at time T_2 before executing P4 at time T_3. In Genetic Algorithm such cases can happen where even though both parents offer valid solutions this doesn't necessarily mean that their offspring must be also a valid solution.

As shown the cross-over process alone retains the majority of the traits of the parents specially if they are fairly similar as in the example given. This is where the Mutation process plays a big role. It introduces random changes that has the ability to break the pattern of the traits had by both parents and introduces new traits. In practicality Mutation's contributions are almost always very small and in search algorithms as GA it can introduce a lot of invalid traits as it is random by nature. The inclusion of such invalid offsprings is a design choice as similar to how valid parents can give birth to invalid offsprings, invalid parents can give birth to valid offsprings. This process however needs a lot of time, larger sized populations and algorithms to check the validity of each offpsring.

In this example a mutation can be simulated by swapping the execution of P8 at T_4 with P5 at T_2 in individual #4 which results in The dependency of the process P11 to be done earlier. This allows the parallelism of the process P5 and P11 saving a complete clock cycle as shown in Table XI with the mutated processes shown in red.

ALU #	T_1	T_2	T_3	T_4
ALU #1	P1	P7	P9	P5
ALU #2	P2	P4	P6	P11
ALU #3	P3	P8	P10	

TABLE XI
MUTATED OFFSPRING #3

Analyzing the fitness of offspring #3 it is seen that it reached 91.67% which in this case is the maximum utilization possible for a solution with the constrain of 3 ALUs assumed at the beginning of the example.

REFERENCES

[1] E. Torbey and J. Knight. High-level synthesis of digital circuits using genetic algorithms. In *1998 IEEE International Conference on Evolutionary Computation Proceedings. IEEE World Congress on Computational Intelligence (Cat. No.98TH8360)*, pages 224–229, 1998.

[2] K. Ohmori. High-level synthesis using genetic algorithm. In *Proceedings of 1995 IEEE International Conference on Evolutionary Computation*, volume 1, pages 209–, 1995.