Final Project Report: Hardware Accelerated Genetic Optimization for PCB Layer Assignment

Heather Berlin, Zachary Zumbo
6.375: Complex Digital Systems

Project Mentor: Chanwoo Chung

December 11, 2019

1 Acknowledgements

We would like to acknowledge 6.375 TAs Chanwoo Chung and Shuotao Xu, as well as Professor Arvind for their mentorship and guidance throughout the research, design and implementation of this project. Without them, there would be no final implementation.

2 Problem Statement

Many modern problems are combinatorial in nature, have exponential time deterministic solutions or have very high dimensional search spaces. For example, the traveling salesman problem (which asks for the optimal route through a list of N cities) is NP-Hard and is known to be infeasible to solve for large N. Instead of using an exact algorithm, we can take advantage of non-determinism / stochastic processes to grow solutions that quickly find optima, especially if we aren't worried about finding it globally. One optimization strategy that fits this mold is genetic optimization (GenOpt).

Two major issues with using a GenOpt strategy are finding an effective model / fitness function for the problem, and getting a solution within a reasonable amount of time. Luckily, some problems map very well to a GenOpt approach, and we can also translate the GenOpt process to hardware which will allow for parallelism inside the micro-architecture, circumventing typical compute-bound CPU approaches. Unfortunately, GenOpt as a general concept is not easily mappable to hardware due to each problem having a specific representation. For example, the traveling salesman problem has a different genetic representation than the popular OneMax problem (crossover must be implemented completely differently).

Instead of attempting to implement a generalized approach to GenOpt, we will apply GenOpt to a problem in the domain of printed circuit board design. More specifically, we would like to find an optimal layer assignment for K straight line segments on a L layer board that would minimize crossings on each layer. Take Figure 1 as an example. Notice the lines L3 and L6 intersect on the same layer (red). Moving L6 to the yellow layer would resolve this crossing. This is what our algorithm will seek to do automatically.

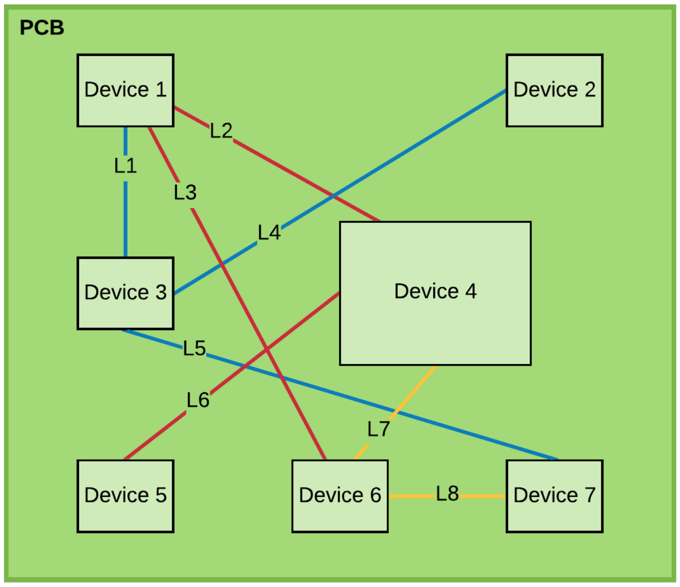


Figure 1: Example PCB. Each line color represents a different layer.

3 Background

GenOpt is based on the biological concept of Darwinian evolution. In nature, variations in genetic material (i.e. DNA) gives rise to small differences in physical expression, which affects an individual creature's ability to mate and survive in a process called natural selection. This phenomenon is named such because the changes occur naturally through genetic operators called crossover (mating) and mutation, and generally the most well adapted creatures are able to find food, a mate, and evade prey.

If we can find a way to map this concept of Darwinian evolution to an algorithm, we can often get arbitrarily close to solving exponential time problems. We can see population diversity in the Darwinian model as a mechanism for exploring a high-dimensional state space, and use a fitness function to model how well equipped an individual is to survive in the environment and pass on its genes to the next generation. In order to solve problems in terms of GenOpt, a chromosome, genetic operators, and a fitness function need to be defined in relation to the problem. In the next section, we will explain how the concepts of GenOpt apply to our multi-layer line intersection problem, and our vision for the high-level design.

4 Benefits of modeling on an FPGA

Genetic algorithms are useful for optimizations problems that have many parameters across a large search space, where traditional techniques may run into troubles. Implementing

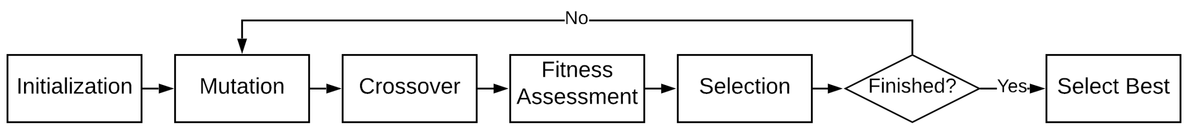


Figure 2: General steps in Genetic Optimization

<L2,L4>	<L3,L4>	<L3,L5>	<L3,L6>	<L5,L6>	<L5,L7>

Figure 3: Labels represent the lines that intersect if placed on a single layer together

genetic optimization on hardware allows for highly parallelized operations, allowing us to solve the problem faster than we'd otherwise be able to if we used software alone.

5 FPGA Inputs

We will use connectal to communicate between the host processor and FPGA. The host PC will provide a serialized list of connection index pairs that are known to intersect when on one layer. These give us an idea of the candidate intersections to check when calculating fitness. An example of how this will look for the design in Figure 1 is in Figure 3

Note that for our problem statement, the line segment positions are fixed, and our goal is to find the optimal z values for each line to minimize the number of line segment intersections.

The variable `N_LEVELS` indicating the total number of levels on the PCB is fixed in our framework. Note that for each gene index i in an individual, z_i is the value of that gene, an integer such that $0 \leq z_i \leq$`N_LEVELS-1`. The initial level assignments for each individual in the population will be generated randomly on the hardware side during initialization.

5.1 Loading intersecting pairs

The input intersecting pairs will be loaded from a binary file, `intersecting-pairs.bin`. `intersecting-pairs.bin` contains connection index information every 2 bytes. Since there are two indices in a pair, each pair takes 4 bytes. For our example in Figure 3, our input file would be 6 pairs * 2 indices/pair * 2 bytes/index or 24 bytes total, and would look like the following (assuming conversion of labels to their associated index, i.e. L1 becomes 1):

0002 0004 0003 0004 0003 0005 0003 0006 0005 0006 0005 0007

5.1.1 Creation of the intersecting pairs binary file, `intersecting-pairs.bin`

We create the intersecting pairs input file, `intersecting-pairs.bin`, by running a python script and specifying the number of line segments to be placed on the PCB along the with number of desired intersecting pairs for the `intersecting-pairs.bin` to contain. The python code for this is located in our git repository in create_intersectingpairs_input.py.

6 High-level design

Before any hardware design occurs, we need to define our problem in terms of Darwinian evolution, which is outlined in Figure 2. Then, we can break up the problem into modules which will help compartmentalize and modularize the design process.

In our case, it makes sense to model a set of line segments and their layer information as a single chromosome, with each gene being an integer representing a layer a single line segment is on. This is because a chromosome is a set of full design parameters which acts as an individual, and an individual should represent a potential solution to the problem. Thus, a population would represent a set of possible designs which have different fitness scores.

$$\text{population} = \begin{cases} \begin{bmatrix} z_1, & z_2 & \dots & z_k \end{bmatrix}_1 \\ \\ \begin{bmatrix} z_1, & z_2 & \dots & z_k \end{bmatrix}_2 \\ \vdots \\ \begin{bmatrix} z_1, & z_2 & \dots & z_k \end{bmatrix}_M \left\{ \text{chromosome for the } M^{th} \text{candidate design} \right. \end{cases}$$

For a set of k connections with fixed end points regardless of their layer placement, a chromosome in the population contains a potential configuration of the z (level) positions for each of the line segments. Within a chromosome, the i^{th} gene, z_i, represents i^{th} line segment's z coordinate, which can be any integer between 1 and the number of levels, N.

The fitness function will evaluate an individual by the sum of crossings on each layer. This will tie in with the selection algorithm which will seek individuals with a low number of crossings. Mutation will randomly modify layer information, and crossover will swap two chromosome's information via array splicing (in hardware this will just be a mux). The full pipeline will be initialized by user stimulus with a preset or randomized initial population, then the key steps of GenOpt will run until a stop condition is met. GenOpt will then yield the best individual to the user.

6.1 Common Params

6.1.1 `GenOptTypes.bsv`

These are fixed types defined for use on a single design, i.e. PCB. Thus, compilation of the bluespec module will happen once per design, and the rest of the variable parameters will be passed in prior to running GenOpt. Fixed parameters include the number of layers (`N_LAYERS`), the number of intersecting pairs (`N_INTPAIRS`) and the number of connections / genes (`N_CONNECTIONS`).

Randomization is essential for genetic optimization; to decide whether to modify datapoints using the user-defined probabilities mutpb, indpb, and cxpb, we will generate a random number of `RAND_SIZE` bits and compare it to the relevant probability threshold (the probability multiplied by $2^{\texttt{RAND\_SIZE}}$. We chose `RAND_SIZE` to be 8, which allows us to get within a half of a percentage point of precision when using the probability thresholds, since 256 is the upper bound.

```
typedef 8 N_LAYERS;
typedef 100 N_CONNECTIONS;
typedef 50 N_INTPAIRS;
// <A,B> is the same as <B,A>, i.e. pairs are not double counted.
typedef N_CONNECTIONS N_GENES; // Same meaning
typedef 8 RAND_SIZE; // number of bits denoting size of random value generated
typedef 100 MAX_POP_SIZE; // (was 8 for Selection test)
typedef 20 MAX_TOURN_SIZE; // Keep this <= MAX_POP_SIZE
typedef TAdd#(TLog#(MAX_POP_SIZE), 1) LOG_MAX_TOURN_SIZE;
typedef TAdd#(TLog#(MAX_POP_SIZE), 1) LOG_MAX_POP_SIZE;
typedef Bit#(TLog#(N_LAYERS)) Layer; // Which layer a line segment / connection is on.
typedef Vector#(N_GENES, Layer) Chromosome; // What we actually want to modify in GenOpt


interface GeneticOptimizer;
        method Action setIndividualMutationProb(Bit#(RAND_SIZE) indpb); //setIndpb
        method Action setGeneMutationProb(Bit#(RAND_SIZE) mutpb);  //setMutpb
        method Action setCrossoverProb(Bit#(RAND_SIZE) cxpb);  //setCxpb
        method Action setPopulationSize(UInt#(LOG_MAX_POP_SIZE) pop_size);//setPopSize
        method Action setTournamentSize(Bit#(LOG_MAX_POP_SIZE) tourn_size);
        method Action setNumGenerations(Bit#(32) ngen);
        method Action putSampleInput(CxnPair in); // put int pair into the intpairs vec
        method Chromosome getChromosomeOutput;
endinterface
```

7 Genetic Optimization Pipeline

For our genetic optimization pipeline, we use a test bench connectal_test.cpp to send and receive populations through connectal portals. Our full framework, which we describe next, is shown in Fig. 4.

Diagram Legend

N = population size
M = number of genes (i.e. number of connections on PCB board)
L = number of layers on PCB board
K = total number of generations
T = tournament size
P = number of intersecting connection pairs

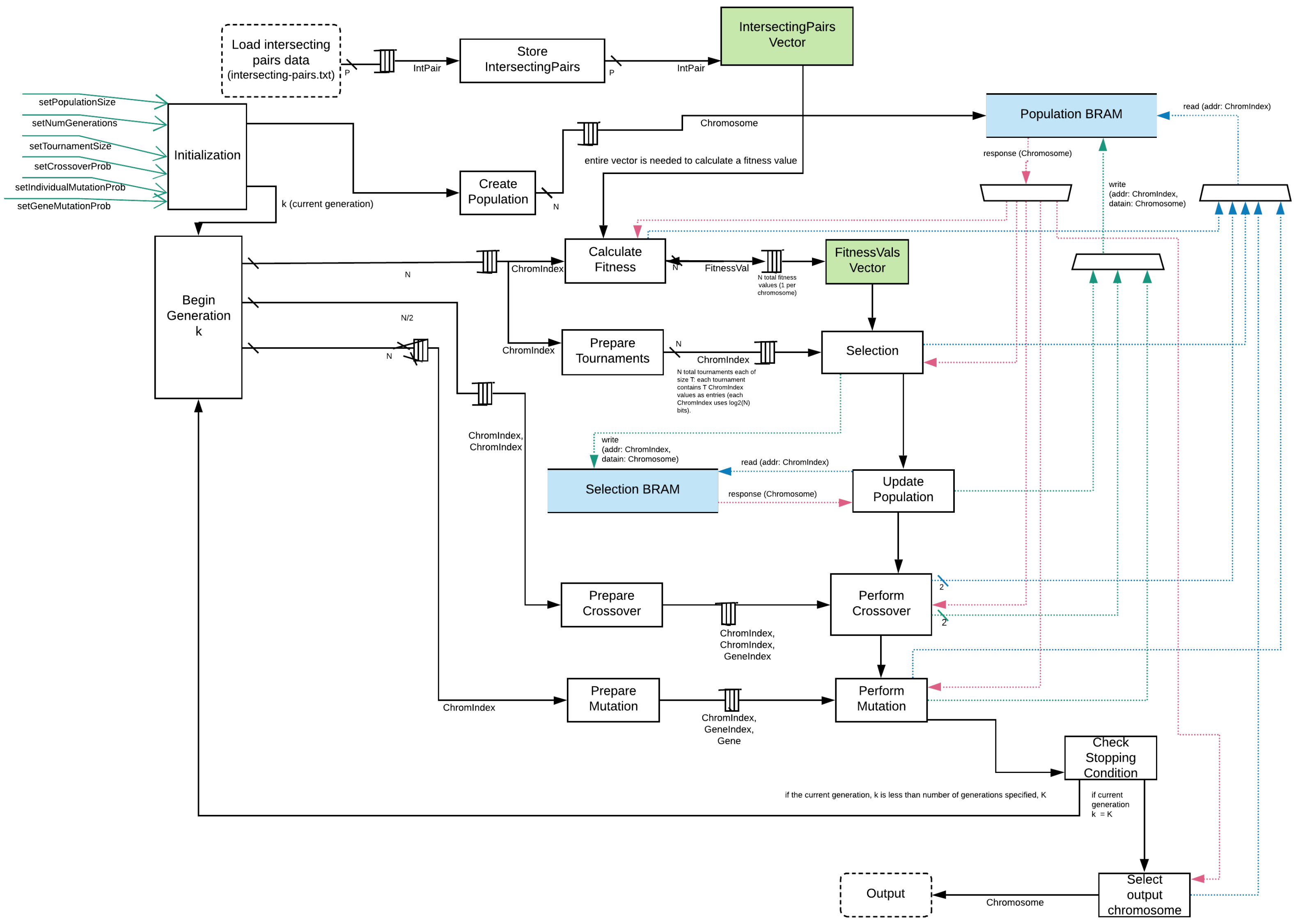
Data Storage

Population (BRAM)
- read and write operations
- addr: chromosome index (0 through N-1)
- word: chromosome (M-length vector of genes)

Selection (BRAM)
- read and write operations
- addr: chromosome index (0 through N-1)
- word: chromosome (M-length vector of genes)

IntersectingPairs register vector
- read-only operations
- vector containing 32bit IntPair entries
    - IntPair contains two 16bit gene index values

FitnessVals register vector
- read and write operations
- M-length vector containing integers denoting fitness



7.1 `GeneticOptPipeline.bsv`

```
LFSR#(Bit#(RAND_SIZE)) lfsr <- mkLFSR_8 i_rand;  //use to generate random integers

module mkGeneticOptPipeline(GeneticOptimizer);
    GeneticOptimizer genopt <- mkGeneticOptimizer();
endmodule

// make specialized Fitness in its own synthesis boundary
module mkGenOptPipelineFitness(SettableFitness#(N_GENES, POPULATION_SIZE));
    SettableFitness#(N_GENES, POPULATION_SIZE) <- mkFitness();
    return fitness;
endmodule

// make specialized Select in its own synthesis boundary
module mkGenOptPipelineSelection(SettableSelection#(N_GENES, 16));
    SettableSelection#(N_GENES, 16) selection <- mkSelection();
    return selection;
endmodule

// make specialized Crossover in its own synthesis boundary
module mkGenOptPipelineCrossover(SettableCrossover#(N_GENES, 16));
    SettableCrossover#(CXPB) crossover <- mkCrossover();
    return crossover;
endmodule

// make specialized Mutate in its own synthesis boundary
module mkGenOptPipelineMutation(SettableMutation#(N_GENES, 16));
    SettableMutation#(N_GENES, 16) mutation <- mkMutation();
    return mutation;
endmodule
```

7.2 Initialization

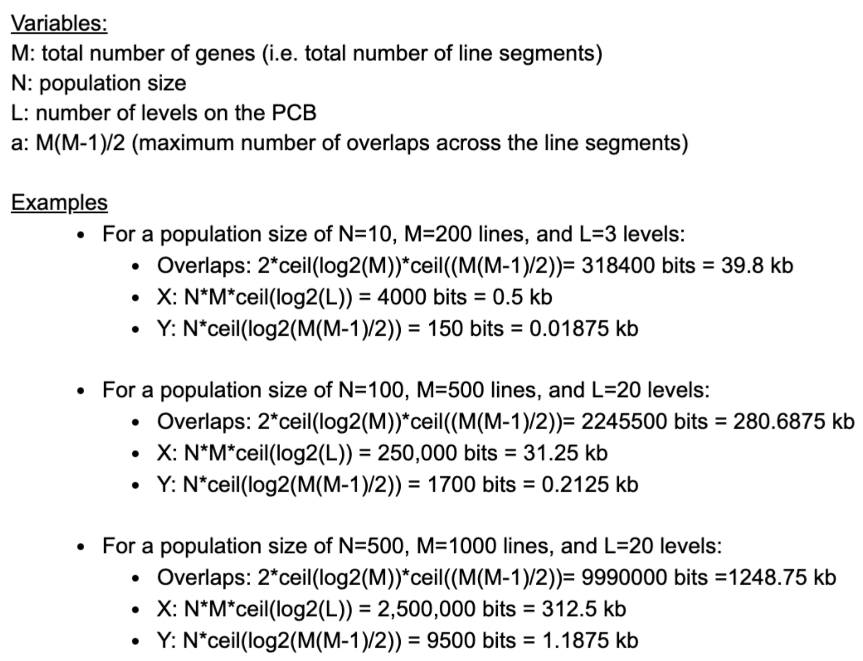
7.2.1 Initializing the population

At the first generation $k = 1$, we will randomly initialize the population, $\mathbf{X_k}$. We only need to initialize the population once. If it is the first generation, we need to randomly initialize a population to use throughout the genetic optimization process.

For each of the N chromosomes in the population, generate M (number of genes, i.e. number of connections on PCB board) random integers between 1 and L (number of layers on PCB board) to create a chromosome vector containing M genes (the i-th gene in a chromosome specifies the layer assignment for the i-th connection on the PCB). The population will contain N chromosomes, each chromosome containing M genes.

7.2.2 *Bits required (storing in BRAM)*

$N \times M \times \lceil \log_2(L) \rceil$ bits are required to store the population. One chromosome corresponds to one word in BRAM. Examples of BRAM size considerations are shown in Figure 5.



Variables:
M: total number of genes (i.e. total number of line segments)
N: population size
L: number of levels on the PCB
a: M(M-1)/2 (maximum number of overlaps across the line segments)

Examples

- For a population size of N=10, M=200 lines, and L=3 levels:
  - Overlaps: 2*ceil(log2(M))*ceil((M(M-1)/2))= 318400 bits = 39.8 kb
  - X: N*M*ceil(log2(L)) = 4000 bits = 0.5 kb
  - Y: N*ceil(log2(M(M-1)/2)) = 150 bits = 0.01875 kb

- For a population size of N=100, M=500 lines, and L=20 levels:
  - Overlaps: 2*ceil(log2(M))*ceil((M(M-1)/2))= 2245500 bits = 280.6875 kb
  - X: N*M*ceil(log2(L)) = 250,000 bits = 31.25 kb
  - Y: N*ceil(log2(M(M-1)/2)) = 1700 bits = 0.2125 kb

- For a population size of N=500, M=1000 lines, and L=20 levels:
  - Overlaps: 2*ceil(log2(M))*ceil((M(M-1)/2))= 9990000 bits =1248.75 kb
  - X: N*M*ceil(log2(L)) = 2,500,000 bits = 312.5 kb
  - Y: N*ceil(log2(M(M-1)/2)) = 9500 bits = 1.1875 kb

Figure 5: Examples with BRAM (note that Overlaps in the examples shown refers to the IntersectingPairs vector described in this report)

7.3 Mutation

Mutation will alter the genes in a chromosome from their initial states. Note that since we are dealing with indices, we can prepare the mutation phase in parallel with fitness value calculations.

Mutation will occur on an individual with probability `mutpb` and will vary each gene of a chromosome with probability `indpb`. If a gene is chosen to be mutated, it will happen uniformly over the choice of valid layers. For example, if `N_LAYERS` = 3, then the mutation can set a single gene to be 0, 1, or 2. In our framework, the number of layers is fixed to be 8, so valid gene values are integers between 0 and 7.

7.3.1 Prepare Mutation

Generate an 8-bit random number, `mut_rn`, which is used to decide whether a chromosome will be mutated Create a vector, `ind_rn`, containing `nchromosomes` randomly generated 8-bit numbers Create a vector, `layer_rn`, containing `nchromosomes` randomly generated numbers between 0 and `N_LAYERS-1`.

7.3.2 Microarchitecture: `Mutation.bsv`

```
interface SettableMutate;
        interface BRAMClient#(Bit#(LOG_MAX_POP_SIZE), Chromosome) bram;
        interface Put#(UInt#(LOG_MAX_POP_SIZE)) setPopulationSize;
        interface Put#(Bit#(RAND_SIZE)) setIndividualMutationProb;
        interface Put#(Bit#(RAND_SIZE)) setGeneMutationProb;
        method Action start;
        method Bool finished;
endinterface
```

7.3.3 Perform Mutation

For each chromosome in the population (`population_bram`), get random numbers from preparing mutation (`prep_mutate.getRNs`) where PrepareMutate `prep_mutate <- mkPrepareMutate`.

If the `mut_rn` is less than `mut_pb`, (`mut_pb` is the threshold created by the user-inputted mutation probability value), then the chromosome will be mutated. Otherwise, we set the value of the mutated chromosome, `mut_chromosome`, to be the same as the original chromosome If the chromosome will be mutated, we go through each of the `nchromosomes` indices in `ind_rn`, and for each index i: if `ind_rn[i]` is less than `ind_pb`, (`ind_pb` is the threshold created by the user-inputted individual mutation probability value) then we set the ith entry of the mutated chromosome, `mut_chromosome`, to be the randomly generated layer specified by `layer_rn[i]`. Otherwise, we set the ith entry of `mut_chromosome` to be the preexisting entry at the ith location of the chromosome, `chromosome[i]`.

Store the mutated chromosome in the population at the same location as the original chromosome.

7.4 Crossover Phase

We found that while crossover is generally useful in creating additional diversity in the population, crossover is not very useful in this problem structure because it nearly always creates chromosomes with much lower fitnesses. In the software (Java) implementation, we have omitted crossover as well. That being said, we still provide an explanation and partial implementation of the crossover module below.

In the crossover phase, we randomly select chromosome pairs from the new population (post-selection) to crossover. Since we have N chromosomes in the population both before and after crossover, we will prepare crossover by selecting pairs of chromosome index values, since the population will change after selection but we can still access chromosome values using the index of a chromosome in the population.

7.4.1 Prepare Crossover

Choose pairs of elements specifying chromosomes in a population of size N (pairs will be chromosome index values from 1 to N).

This can be done in $\frac{N}{2}$ parallel processes where, for a pair of chromosome index values (specified by $2 \times \lceil \log_2(N) \rceil$ bits since each pair consists of 2 chromosome index values from the population.

We generate a random integer, and if it is above the crossover threshold, then we add the pair of chromIndex values to a FIFO vector containing the pairs for crossover.

We perform *one-point crossover*, so a random point is specified to indicate the starting point in the chromosome at which genes in the chromosomes are swapped. Along with the chromosome index values, we also include the gene index within the chromosome where crossover begins in the FIFO.

Note that the FIFO will have a length $\leq \frac{N}{2}$, and the FIFO stores vectors of length 3 that use log2(N) bits for the first two vector entries (the index values for the chromosome index values that will crossover, where each chromosome index (chromIndex) uses log2(N) bits), and log2(M) bits for the third entry, which denotes the gene index within the chromosomes to begin the crossover.

Note that because we perform crossover after selection, if chromIndex is i, this corresponds to the chromosome located at the i-th index of the selection output vector.

7.4.2 Perform Crossover

To perform crossover, we use entries from the FIFO from prepare crossover (an entry contains two ChromIndex vals and a GeneIndex), take the two chromosomes specified by the ChromIndex values in the population after selection and swap their genes starting from the GeneIndex location, all the way to the end of the chromosome.

7.5 Fitness

Fitness will be determined as the sum of crossings on each layer. Given a set connections (line segments), this requires running a line intersection algorithm between all pairs of lines to determine if each cross. Since the line segments do not change location, only layers, we can precompute the crossings for every line intersection and store them as a list of intersecting pairs. This process is done in software prior to FPGA initialization, then passed into HW via indication.

Then, we can just check if the two intersecting lines are on the same layer. If yes, then a crossing has occurred. If no, then no crossing has occurred because the lines are on separate layers. For the example in Figure 1, the associated intersecting pairs vector and chromosome are provided below in Figure 6.

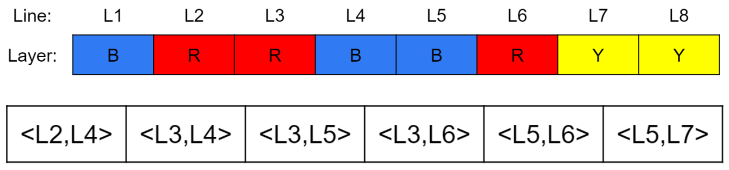


Figure 6: Intersecting pairs & chromosome representation of the problem

For this example, we go through each of the intersecting pairs, checking if the layer information for these two lines (which is stored in the chromosome) is equal. The only intersection happens to occur for the pair $< L3, L6 >$, which shows both lines on the red layer. If L6 was instead on the yellow layer, there would have been zero intersections for this design, and therefore the fitness would have been a perfect zero.

7.5.1 Microarchitecture: `Fitness.bsv`

```
interface SettableFitness;
    interface BRAMClient#(Bit#(LOG_MAX_POP_SIZE), Chromosome) bram;
    interface Put#(UInt#(LOG_MAX_POP_SIZE)) setPopulationSize;
    interface Put#(IntersectingPairsVec) setIntersectingPairsVec;
    method Action process;  // begin processing the fitness values
    method Bool is_finished;
    method ActionValue#(FitnessVec) getFitnesses;
endinterface

interface FitnessOneIndividual;
```

```
    method Action start(Chromosome chromosome, IntersectingPairsVec intersectingpairs);
    method UInt#(32) getFitness;
endinterface
```

7.5.2 *Constraints*

Before proceeding to the selection phase, a fitness value for every chromosome j in the population of size N in generation k must be calculated, as the selection phase must have access to the fitness values of all chromosomes in the population.

7.5.3 *Hardware Design*

(a) For each chromosome j in the population of size N, the fitness value can be calculated in parallel. In our implementation, we instead iterate through each chromosome to reduce hardware requirements.

(b) Once all N fitness values are calculated (one for each chromosome), the calculations in the fitness phase are complete

7.5.4 *Output*

The fitness phase will produce a vector of length N containing fitness values for each chromosome. The j^{th} entry represents the fitness value of chromosome j and takes up a bits, where a is the number of bits required to specify a level of the PCB design. In our examples, $a = 8$ bits (allowing for 256 levels)

For generation k, the fitness phase output required to proceed to the selection phase is below.

$$\mathbf{Y_k} = \begin{bmatrix} y_{1k} \\ y_{2k} \\ \vdots \\ y_{Nk} \end{bmatrix}$$

7.5.5 *Bits required*

For a generation k, this requires $N \times b$ bits, where N is the population size and b is the number of bits required to express the fitness value for a chromosome j in the population.

We specify the fitness value by the number of intersections (line crosses) in a particular design. Since we have a total of M lines and the maximum number of intersections is $\frac{M(M-1)}{2}$, b must be large enough to express this maximum number of intersections.

$b = \log_2(\frac{M(M-1)}{2})$ bits required to express the fitness value for a chromosome j in the population.

7.6 Selection

In order to select new individuals for our updated population, we implement two modules, Selection and SelectBest. Selection provides a tournament-based approach which produces N individuals, while SelectBest is a simple module which returns the best individual in the population. The second module is used at the very end of the GenOpt process to find the best design, which is then output to software via indication.

7.6.1 Tournament Selection

To inject randomness into the selection process, GenOpt libraries often default to using tournament selection N times (where N is the number of individuals in a population) with a fixed tournament size k. This entails selecting k individuals at random, then selecting the individual with the highest fitness. This process occurs N times. We perform N tournaments of size T ($T=$ tournament size) for the selection phase.

7.6.2 Prepare Tournaments

We can run `Prepare_Tournments` in parallel for each of the N tournaments to create a vector $\mathbf{v}$ of length T, where each vector entry is a value between 1 and M (M=number of genes in chromosome).

$$\mathbf{U} = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_N \end{bmatrix}$$

Each tournament is prepared by randomly generating T values to between 1 and M (number of genes in chromosome) to create a T-length vector containing chromIndex values. These values specify the chromosomes included in each tournament. We can prepare the tournaments independently, so we could have N parallel processes that create vectors taking up $T \times \log_2(M)$ bits.

The output of `Prepare_Tournaments` is $\mathbf{U}$, which contains N of these tournament vectors (each tournament vector denoted as u_i), requiring a total of $N \times T \times \log_2(M)$ bits. Note that each tournament vector u_i requires $T \times \log_2(M)$ bits.

7.6.3 Selection (Perform Tournaments)

Once we have $\mathbf{U}$, we can perform the ith tournament, u_i if

- for the ith tournament, the fitness values for the T chromosomes included in tournament u_i have been calculated

We can perform the tournaments independently. There will be N separate tournaments, in which the chromosome with the best fitness value will be chosen for each tournament.

7.6.4 *Bits required (storing in BRAM)*

$N \times M \times \lceil \log_2(L) \rceil$ bits are required to store the population. One chromosome corresponds to one word in BRAM. Examples of BRAM size considerations are shown in Figure 5.

7.6.5 *Updating the population*

The selection phase will request chromosomes from the population BRAM and then store them to the selection BRAM (same size as popuation BRAM) as the tournaments are performed. The winning chromosome id (chromosome id with the best fitness) of tournament i will be requested from the population bram, and once it is received, the chromosome value will be inserted into the selection bram at location i.

Once all tournaments are completed and the selection bram has been updated with all chromosomes in the new population (winning chromosomes from each of the N tournaments), BRAM requests to write to the population BRAM will be sent from the Selection module. The chromosome at location j in selection bram will be written to location j in the population bram, overwriting the population contents once selection is complete.

7.6.6 *Bits required*

For a generation k, $\mathbf{V}$ requires $N \times \lceil \log_2(N) \rceil$ bits, where N is the population size and $\log_2(N)$ is the number of bits required to express the index of a chromosome in $\mathbf{X_k}$ ($\mathbf{X_k}$ contains N chromosomes).

7.6.7 Microarchitecture: `Selection.bsv`

```
interface SettableSelection;
        interface BRAMClient#(Bit#(LOG_MAX_POP_SIZE), Chromosome) bram;
        interface Put#(UInt#(LOG_MAX_POP_SIZE)) setPopulationSize;
    interface Put#(Bit#(LOG_MAX_POP_SIZE)) setTournamentSize;
        method Action start(FitnessVec x);
        method Bool finished;
endinterface

interface SelectBest;
        interface BRAMClient#(Bit#(LOG_MAX_POP_SIZE), Chromosome) bram;
        interface Put#(UInt#(LOG_MAX_POP_SIZE)) setPopulationSize;
        method Action start(FitnessVec x);
        method ActionValue#(Chromosome) getbestInd();
        method Bool finished;
endinterface
```

7.7 Stopping Condition

If the current generation k is equal to the number of generations specified, K, we proceed to the SelectBest module. Otherwise, proceed with another generation, incrementing the current generation by 1.

When selecting the best chromosome in the population, the SelectBest module loads the fitnesses vector and outputs the chromosome at the `population_bram` address location being the index value of the minimum-valued entry in `fitnesses` (i.e. the chromosome with the best fitness score).

This chromosome is our output. The vector output of length `nchromosomes` denotes the layers of the pcb board for each line segment. I.e. the ith entry of the vector output specifies the layer of the PCB to place the ith input line segment on.

7.7.1 Connectal Outputs

Once the population has converged (determined by a stopping condition), the genetic algorithm terminates and the best chromosome from selection will be the output sent to connectal.

We will call this best chromosome `opt_levels`, which is a vector of length k contains the optimal z coordinates for each of the k line segments given that there are N levels on the PCB.

$$\texttt{opt\_levels} = \begin{bmatrix} z_1, & z_2, & \dots & z_k \end{bmatrix}$$

The i^{th} entry of `opt_levels`, z_i, denotes the optimal level chosen for the i^{th} input line segment. Note that z_i is an integer such that $1 \leq z_i \leq N$.

Once the genetic optimization pipeline is finished, entries from the otuput vector are sent to connectal, which then writes the contents to the binary file `out.positions` and closes the file, terminating, when the number of entries received is equal to the number of line segments to place on the PCB.

7.8 Design Verification and Testing

For the line intersection algorithm, we will have a set of unit tests to ensure proper operation. As for the entire module, since the algorithm is non-deterministic, we will compare its operation to a pre-existing known-working software implementation written in Python using *Distributed Evolutionary Algorithms in Python (DEAP).* We also have unit test cases for the modules within the Genetic Optimizer pipeline, which we use to test modules on an individual level.

8 Implementation Evaluation, Performance

Our genetic optimization source code is located at github repo.

8.1 Running the simulation

To run the software with user-specified arguments once the project is initially built, we can do so with the following command (in this example, running simulation):

```
connectal$ NGENS=10 POPSIZE=80 TOURNSIZE=10 CXPB=0 INDPB=0.4 MUTPB=0.02 make run_simulation
```

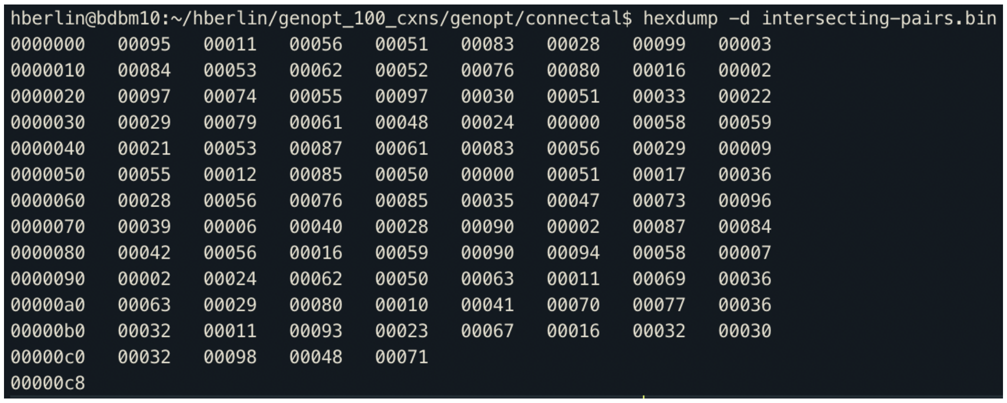
In this case, the number of generations is specified to be 10, the population size is 80, the tournament size is 10, the crossover probability is 0 (this module does not get called in our implementation since mutation has been observed to help with convergence the most in software examples and we decided to focus on the other modules of the genetic optimization process), the individual mutation probability is 0.4, and the gene mutation probability is 0.02. For this example, 100 lines were specified to be placed on an 8-layer board, with a total of 50 intersecting pairs for the 100 line segments specified in `intersecting-pairs.bin` (see Figure 7 for file contents).

Contents of the output file produced by running the simulation for this example are shown in Figure 8.

8.2 Running on the FPGA

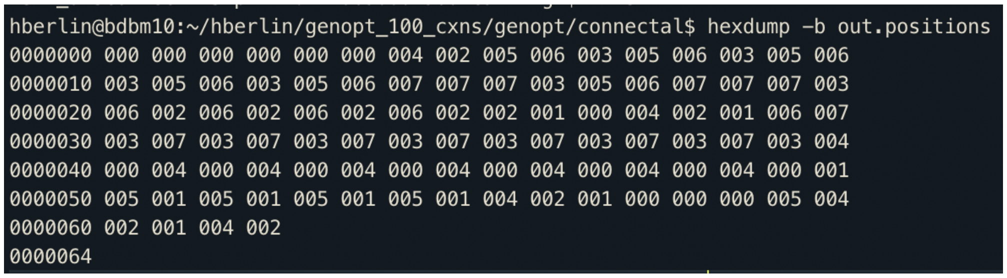
To run our program with user-specified hyperparameters on the FPGA, we can run following command:

```
connectal$ NGENS=10 POPSIZE=80 TOURNSIZE=10 CXPB=0 INDPB=0.4 MUTPB=0.2 make run_fpga
```



```
hberlin@bdbm10:~/hberlin/genopt_100_cxns/genopt/connectal$ hexdump -d intersecting-pairs.bin
0000000   00095   00011   00056   00051   00083   00028   00099   00003
0000010   00084   00053   00062   00052   00076   00080   00016   00002
0000020   00097   00074   00055   00097   00030   00051   00033   00022
0000030   00029   00079   00061   00048   00024   00000   00058   00059
0000040   00021   00053   00087   00061   00083   00056   00029   00009
0000050   00055   00012   00085   00050   00000   00051   00017   00036
0000060   00028   00056   00076   00085   00035   00047   00073   00096
0000070   00039   00006   00040   00028   00090   00002   00087   00084
0000080   00042   00056   00016   00059   00090   00094   00058   00007
0000090   00002   00024   00062   00050   00063   00011   00069   00036
00000a0   00063   00029   00080   00010   00041   00070   00077   00036
00000b0   00032   00011   00093   00023   00067   00016   00032   00030
00000c0   00032   00098   00048   00071
00000c8
```

Figure 7: `intersecting-pairs.bin` file contents for the FPGA experiment outlined. 50 pairs of line segments which overlap if placed on the same PCB layer are stored in this file. A total of 100 line segments are used in this experiment.

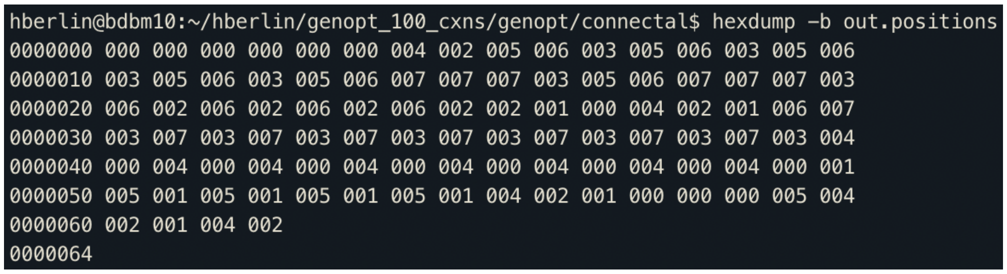


```
hberlin@bdbm10:~/hberlin/genopt_100_cxns/genopt/connectal$ hexdump -b out.positions
0000000 000 000 000 000 000 000 004 002 005 006 003 005 006 003 005 006
0000010 003 005 006 003 005 006 007 007 007 003 005 006 007 007 007 003
0000020 006 002 006 002 006 002 006 002 002 001 000 004 002 001 006 007
0000030 003 007 003 007 003 007 003 007 003 007 003 007 003 007 003 004
0000040 000 004 000 004 000 004 000 004 000 004 000 004 000 004 000 001
0000050 005 001 005 001 005 001 005 001 004 002 001 000 000 000 005 004
0000060 002 001 004 002
0000064
```

Figure 8: Simulation output when running `connectal$ NGENS=10 POPSIZE=80 TOURNSIZE=10 CXPB=0 INDPB=0.4 MUTPB=0.2 make run_simulation`

The hyperparameters can be modified by the user, but results for this run are shown below. For this example, a total of 50 intersecting pairs for the 100 line segments are specified in `intersecting-pairs.bin` (see Figure 7 for file contents).

When running the program on the FPGA, the output is the binary file `out.positions`, which denotes the layer assignments for the 100 layers on the PCB. The hexdumped contents of the output file for the experiment described indicating the layer assignments for each of the 100 line segments on the PCB are shown in Figure 9.



```
hberlin@bdbm10:~/hberlin/genopt_100_cxns/genopt/connectal$ hexdump -b out.positions
0000000 000 000 000 000 000 000 004 002 005 006 003 005 006 003 005 006
0000010 003 005 006 003 005 006 007 007 007 003 005 006 007 007 007 003
0000020 006 002 006 002 006 002 006 002 002 001 000 004 002 001 006 007
0000030 003 007 003 007 003 007 003 007 003 007 003 007 003 007 003 004
0000040 000 004 000 004 000 004 000 004 000 004 000 004 000 004 000 001
0000050 005 001 005 001 005 001 005 001 004 002 001 000 000 000 005 004
0000060 002 001 004 002
0000064
```

Figure 9: FPGA output when running `connectal$ NGENS=10 POPSIZE=80 TOURNSIZE=10 CXPB=0 INDPB=0.4 MUTPB=0.2 make run_fpga`

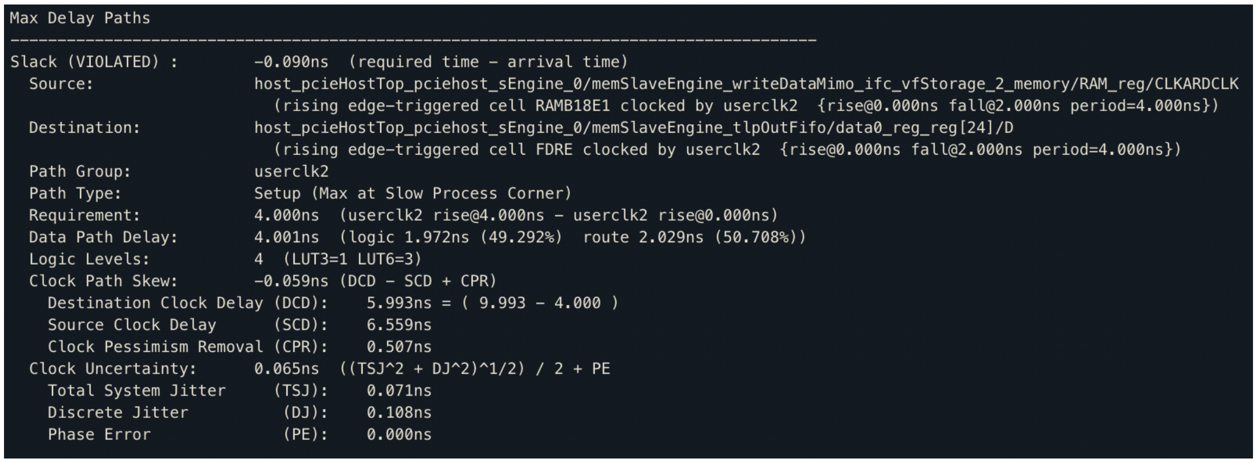
8.3 FPGA Performance

Our design (containing 100 line segments, 8 layers on the PCB) builds on the FPGA and meets the timing conditions when the main clock period is set to 17ns, which meets our design requirements. The limiting factor in performance is the amount of information stored in BRAM, which is determined by the number of line segments to place on the PCB and the size of the population.

Looking at `top-post-route-utils-summary.txt`, in our entire design (mkPcieTop), the total number of LUTs used is 36897 (12.15%), the total number of FFs is 45716 (7.53%), and the design uses 0 DSP48 blocks.

8.3.1 Critical Path

To find the critical path, we built our program on the FPGA with clock period of 8, which was chosen to use a main clock period that's much smaller than what the current clock minus the slack was, since prior to doing this, Vivado could not find a global optimal so that every clock domain meets the timing requirements and the violated path was in the clock domain, so we didn't gain insight on the critical path for our specific program. Building the program on the FPGA with a main clock period of 8ns, we found our critical path, which is shown in the partial output from `top-post-route-timing-summary.txt` in Figure 10.



```
Max Delay Paths
--------------------------------------------------------------------------------------
Slack (VIOLATED) :        -0.090ns  (required time - arrival time)
  Source:                 host_pcieHostTop_pciehost_sEngine_0/memSlaveEngine_writeDataMimo_ifc_vfStorage_2_memory/RAM_reg/CLKARDCLK
                            (rising edge-triggered cell RAMB18E1 clocked by userclk2  {rise@0.000ns fall@2.000ns period=4.000ns})
  Destination:            host_pcieHostTop_pciehost_sEngine_0/memSlaveEngine_tlpOutFifo/data0_reg_reg[24]/D
                            (rising edge-triggered cell FDRE clocked by userclk2  {rise@0.000ns fall@2.000ns period=4.000ns})
  Path Group:             userclk2
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            4.000ns  (userclk2 rise@4.000ns - userclk2 rise@0.000ns)
  Data Path Delay:        4.001ns  (logic 1.972ns (49.292%)  route 2.029ns (50.708%))
  Logic Levels:           4  (LUT3=1 LUT6=3)
  Clock Path Skew:        -0.059ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    5.993ns = ( 9.993 - 4.000 )
    Source Clock Delay      (SCD):    6.559ns
    Clock Pessimism Removal (CPR):    0.507ns
  Clock Uncertainty:      0.065ns  ((TSJ^2 + DJ^2)^1/2) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Discrete Jitter          (DJ):    0.108ns
    Phase Error              (PE):    0.000ns
```

Figure 10: partial output from `top-post-route-timing-summary.txt` when building on an FPGA with a small main clock period of 8ns

This result showed that our critical path was when we write data from BRAM to the outFIFO, showing that this is an area for us to explore implementing design variations to cut the path and get the program to build on an FPGA with a faster clock.

8.3.2 Comparing performance with software results

For 100 connections, 50 intersecting pairs, a population size of 100 and 100 generations, the FPGA implementation ran in 28009 cycles * (17 ns per cycle) = 4.8×10^{-4} sec.

The software implementation, ran on a CPU (Single Core), took 1.99 sec for the same experiment.

Our FPGA implementation shows a 4000x speed improvement with the fitnesses essentially equal, showing that our FPGA implementation ran running quickly and produced a valid result on par with the result produced by the software implementation, validating our design.