Revisiting Genetic Algorithms for the FPGA Placement Problem

Peter Jamieson
Miami University, Oxford, OH, 45056
Email: jamiespa@muohio.edu

Abstract— *In this work, we present a genetic algorithm framework for the FPGA placement problem. This framework is constructed based on previous proposals in this domain. We implement this framework in an academic FPGA tool, and run a set of experiments that show that the fine grain genetic mutation approach, previously proposed, is not as good as an existing simulated annealing algorithm. This does not discount the use of genetic algorithms in this domain, and instead, provides motivation to explore other aspects of the problem to apply genetic algorithms to this problem.*

Keywords: GA, Placement, FPGA, Simulated Annealing

1. Introduction

Very Large Scale Integration (VLSI) design automation consists of a number of complex problems that map a users digital design onto a target Integrated Chip (IC) technology. One of these technologies, which is seeing more an more popularity, is the Field-Programmable Gate Array (FPGA), since it can quickly be programmed with any digital design. FPGAs are analogous to the general purpose processor except that FPGAs programmably can implement digital logic instead of sequential programs.

One of the problems, when a design is mapped to an FPGA, is how to place the circuit components onto the programmable silicon such that various metrics such as the critical path (corresponds to speed), power consumption, and wire consumption are minimized. The placement problem is an NP-complete problem [1] and various heuristic algorithms have been used to solve this problem. In this paper, we focus on genetic algorithms (GA) used for the FPGA placement problem. In particular, we set out to investigate how previous approaches using a GA for placement improve the results compared to the exiting simulated annealing approach (SA). It is our hypothesis that based on the no-free-lunch (NFL) theorem [2], which suggests that there does not exist an algorithm for solving all optimization problems that is generally on average better than competitors, that the GA proposed in [3] for FPGA placement may be as good as SA, but is not better since there solution does not exploit any aspect that the SA does not.

To test this hypothesis, we implemented a GA framework for FPGA placement within the VPR 5.0 academic framework [4] and then ran a set of experiments with the SA implemented in VPR 5.0 compared to various GAs looking at how the algorithms perform with respect to the critical path and power consumption of each of the benchmarks. Our results show that when both the SA and GA algorithms are given the same amount of processing time (based on the time taken for the SA placement) that the results generated from the GA algorithms (including an algorithm similar to the one in [3]) produce significantly slower critical paths and consume more power than the SA generated results. In another experiment, without a strict run-time dependence between the algorithms, the GAs are run for a specific number of generations and the results do improve, approaching those of the SA.

Our results demonstrate that the existing fine grain mutation approaches for FPGA GA placers is not the best approach. This does not mean that GA for FPGA placement is not an appropriate approach, and in our discussion we provide some suggestions on how to create a better algorithm.

The motivation to further study and improve GAs for FPGA placement is due to the increasing runtime of the Computer Aided Design (CAD) flow that maps user designs to the FPGA (placement is one stage of this flow) [5]. GAs are inherently parallelizable, and for this reason alone, it is important to understand how this type of algorithm works within the FPGA domain. This work is a first step in moving away from previous approaches.

The remainder of this paper is organized as follows. Section 2 briefly describes FPGAs, CAD for FPGAs, and the FPGA placement problem for FPGAs including genetic algorithms. Section 3 describes our implementation of a genetic algorithm. Section 4 describes our experimental setup and shows results of our experiment for the MCNC benchmarks. Finally, Section 5 concludes this work.

2. Background

FPGAs are programmable ICs that can implement any digital design. These devices consist of programmable logic blocks and a programmable routing [6] where the programmable routing consists of wire segments that are connected to either logic blocks or other wire segments via programmable switches. The logic blocks are also called clusters (which is the term we will use throughout this work) where these clusters commonly consist of a combination of Look-up Tables (LUTs), flip-flops, and internal programmable routing. The most important aspect of this architecture for the placement problem is the cluster, and the

placement algorithm maps design clusters onto the FPGA, which, itself, consists of an array of these clusters.

An example open source CAD flow used by VPR 5.0, which is an academic FPGA tool that allows us to experimentally test algorithms and FPGA architectures. First, a digital design is created in Verilog HDL [7] and used as the input to this CAD flow. A series of CAD flow stages convert the design to a programmable bit-stream that can be uploaded to the FPGA to implement the digital design. The focus of this work is on the placement stage, and that will be the focus of this background section.

2.1 Details of FPGA Placement

FPGA placement algorithms try to place the clusters, representing the digital design, onto the array of FPGA clusters such that the critical path (the longest path from either a primary input to a primary output, a primary input to flip-flop, flip-flop to flip-flop, or flip-flop to primary output) is minimized, the power consumption of the programmable routing is minimized, and the overall wire-length of the mapped circuit is minimized. This problem has been shown to be NP-complete to solve optimally, and a number of popular algorithms have been used to solve this problem including simulated annealing ([8], [6]), which is the algorithm used in VPR 5.0 [4], min-cut ([9], [10], [11]), and analytic ([12], [13]), which includes force-directed placers.

We focus on SA and GA algorithms in this work, and the simulated annealing algorithm, when used in the FPGA domain for placement, attempts to minimize the various metrics for optimization based on the process of cooling metals. Basically, a cooling schedule controls the weighting of a probability function. This function determines if randomly selected swaps between clusters on an FPGA are accepted or not. Each random swap of points will either improve or degrade the critical path, and initially, all swaps are accepted regardless if they improve the optimizations metrics or not. As the temperature cools, only swaps that improve the critical path are accepted. In this way, the early phases of the cooling schedule is used to allow hill climbing that will, hopefully, avoid local minimums in this optimization problem [6].

The two most relevant aspects of the annealer as a placement algorithm for FPGAs are the scheduling of the cooling and the cost function. The scheduling of the annealer determines if a random swap is accepted and determines the maximum Manhattan distance of the cluster swaps. As the algorithm continues, swapping of clusters that don't improve the cost function are not accepted, and the distance between the swaps is reduced.

The distance of a random swap of clusters on a X by Y array is based on the term R_{limit}. Given a 5 by 5 FPGA, R_{limit} can have a maximum value of 5 meaning that a cluster located at the x coordinate 0 and y coordinate 0 could be swapped with another cluster located at x coordinate 4 and y coordinate 4. As R_{limit} is reduced by the annealer's scheduler, the distance for a swap is reduced, and this represents the stabilizing of the placement algorithm (the cooling and lower excitation of the molecules in a metal). For example, Figure 1 shows a 5 by 5 FPGA where random swaps could happen for the digital logic at x = 0, y = 1 with an $R_{limit} = 2$ (The candidate cluster to swap is surrounded by a thick dotted line, and the clusters it can swap with are shaded in darker grey).

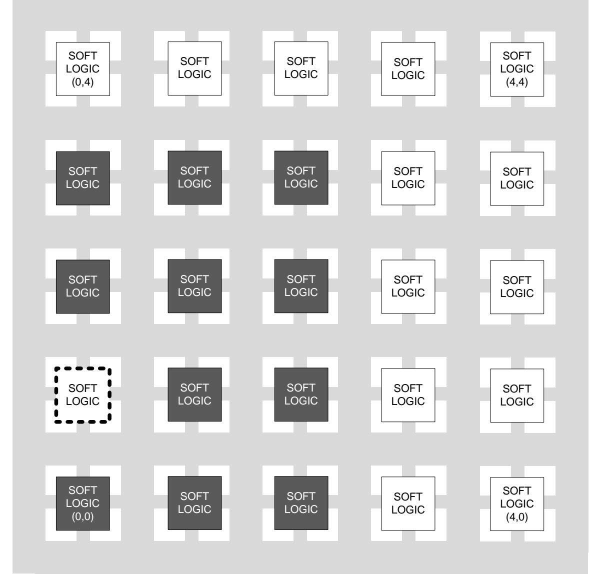


Figure 1: Shows how the R_{limit} factor affects the distance of swaps

The second aspect of the annealer is the cost function used estimate the quality of the placement. The cost function for SA in VPR 5.0 consists of two components defined in [6]. First is the sum of the bounding box dimensions of all nets which estimates the total amount of wire needed to implement the circuit (also know as wire-length). Given N nets, $bb_x(i)$ and $bb_y(i)$ are the x and y dimensions of a bounding box for each $net(i)$, and $q(i)$ as a scaling factor for better wire-length estimates, then the first component of the cost function is defined as:

$$WiringCost = \sum_{i=1}^{N} q(i) \cdot [bb_x(i) + bb_y(i)] \quad (1)$$

The second component of the cost function evaluates the timing cost of a placement where,

$$TimingCost = \sum_{\forall i,j \in circuit} Delay(i,j) \cdot Criticality(i,j)^{(CE)} \quad (2)$$

where CE is a constant, Delay(i,j) is the delay of the connection from source i to sink j, and Criticality(i,j) is a measure of how close the given i, j path is to the global critical path.

The perceived change in the cost function for each placement change is:

$$Cost = \lambda \cdot \frac{TimingCost}{PreviousTimingCost} + (1 - \lambda) \cdot \frac{WiringCost}{PreviousWiringCost} \quad (3)$$

where the previous costs are used to normalize the two components of the cost function, and the λ parameter is used to weight the optimization importance of each of the two components.

Lamoureux *et. al.* [14] extended this cost function to be power aware. They added a new term:

$$PowerCost = \sum_{i=1}^{N} q(i) \cdot [bb_x(i) + bb_y(i)] \cdot Activity(i) \quad (4)$$

where $Activity(i)$ is the switching activity on a particular net, and by reducing this component, the power consumed over long and power hungry programmable routing lines is reduced. The new cost function with this component is the following:

$$Cost = \lambda \cdot \frac{TimingCost}{PreviousTimingCost} + (1 - \lambda) \cdot \left[(1 - \gamma) \cdot \frac{WiringCost}{PreviousWiringCost} + \gamma \cdot \frac{PowerCost}{PreviousPowerCost} \right] \quad (5)$$

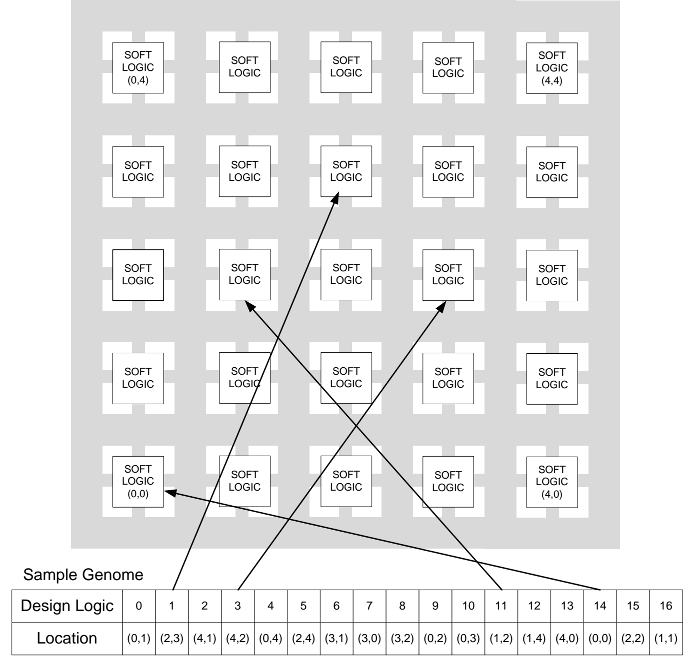
where the γ factor is used to control how strong or weak the power optimization component is.

As described, the parameters γ and λ are used to control the weighting of the cost function, or how much the cost function cares about optimizing for a particular metric. Previous research has shown that for a cost function that attempts to optimize for power has a γ equal to 0.8 and a λ equal to 0.5 [14]. In our experiments, we use this cost function as the fitness function to fairly compare the SA versus GA implementations. It is possible to experimentally search for the best of these parameters for a GA (as they were experimentally tuned for an SA algorithm), but for the scope of this paper we simply use the above values instead of tuning each of the algorithms.

2.2 Genetic algorithms and placement

Genetic algorithms and evolutionary algorithms use the "survival of the fittest" idea to heuristically find good solutions for a problem. Within the VLSI domain GAs have been applied to a number of CAD problems, and for a more thorough survey of these problems the reader should read [15]. In this work, we are specifically concerned with GAs for FPGA placement.

Genetic algorithms (GA) and evolutionary programming algorithms have been previously implemented and explored for FPGA placement in two detailed cases. The first instance, which this paper re-investigates, is the implementation by Venkatraman *et. al.* [3] in which they implemented a GA based placer in VPR 4.3 (the predecessor to VPR 5.0). In their work, each cluster's location on the FPGA array is a gene, and the 2-D location of each of the clusters forms an individuals genome. Figure 2 shows how a genome for a design consisting of 20 elements is represented.



Design Logic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Location	(0,1)	(2,3)	(4,1)	(4,2)	(0,4)	(2,4)	(3,1)	(3,0)	(3,2)	(0,2)	(0,3)	(1,2)	(1,4)	(4,0)	(0,0)	(2,2)	(1,1)

Figure 2: Sample genome for 20 elements on a 5x5 FPGA

A population of these individuals is created and each individual is evaluated based on a fitness function similar to the one described earlier for timing and wiring cost (there is no component for power reduction). Within the population, the fittest individuals are kept and mutated to create the next generation based on proportional representation, which is left undefined by the authors. Mutations are based on local cluster swaps (R_limit = 1) and global swaps (R_limit set to the maximum). Their results show that this algorithm improves the critical path compared to VPR's simulated annealing algorithm for ten benchmarks. Unfortunately, there is no analysis of run-times for the two algorithms (SA and GA), which leaves a few questions unanswered.

More recently, Meng *et. al.* [16] have created an algorithm that combines both GA and SA algorithms for placement. Their approach claims that the GA aspect of the algorithm are used to escape local minimums (as another form of hill climbing) and the simulated annealing is used to quickly improve solutions. The genome for their approach is the same as the one previously described. They also use a fine-grain mutation based on swapping clusters, and they propose a new method for cross-breeding fit individuals. Their results

show similar costs compared to VPR 4.3's simulated annealing approach with similar run-times. Within this work, though their concepts for cross-breeding of placements is very interesting, we have not considered or compared their implementation in our GA framework.

3. GA Framework for Placement

We have built a genetic algorithm placer framework in VPR 5.0 that focuses on optimizes placement for wire-length, speed, and power consumption as defined by equation (5). A key difference between this work and previous evaluations of GAs for FPGA placement is the power estimation framework that allows the activation estimation (estimation of switching activity) for each of the nets in a design to be read in and used to estimate power consumption of a particular design on an FPGA [17]. This allows us to use the cost function as shown in equation (5) to be used to rank the the fitness of a population of placement solutions.

Similar to other GA implementations of FPGA placement, our GA placement algorithm framework creates a genome based on the x and y coordinates of each cluster in the design (see Figure 2). In addition to how the genome is represented, we define a number of parameters within the framework that control the GA. The size of a population is defined by σ. Using this number we define the parameters ω, α, and β as percentages where $\omega + \alpha + \beta = 100\%$, and $\omega\%$ of the population is the number of individuals from the current generation to maintain as parents in the next generation, $\alpha\%$ of the population are the children of the parents, and $\beta\%$ of the population is randomly created new individuals. The β parameter is present for another CAD experiment we have performed with this algorithm, but in this work it is always set to 0%. Table 1 summarizes these parameters.

Our GA framework uses a mutation operator to create new individuals in a population. Random swaps of clusters on the FPGA are the mutation operations for our GA framework, and therefore, this mutation is related to the term R_{limit}, which controls the distance between cluster for a random swap. The number of mutations per new individual is defined by the parameters $local_swaps\%$ and $global_swaps\%$ where $local_swaps\%$ multiplied by the number of clusters in a circuit defines the number of mutations (or swaps) to try where R_{limit} is equal to one, and $global_swaps\%$ multiplied by the number of clusters in a circuit is the number of mutations/swaps to try where R_{limit} is scheduled to be between 1 and the maximum size of the FPGA array in one dimension. These parameters were previously described in [3] and are used here so that we can compare SA to the original GA in [3] and our own implementations. A mutation (or swap) is only accepted if the mutation results in an improved placement based on the cost function. Table 1 also summarizes these parameters.

The pseudocode for our GA framework for FPGA placement is shown in Algorithm 1. Based on all of the parameters we define three different GAs for FPGA placement. As stated earlier the following values are constant: $\sigma = 0\%$, $\lambda = 0.5$, and $\gamma = 0.8$. Our first algorithm, which we call GA_OLD, is a version of [3], and the parameters for this algorithm are shown in the second column of Table 2. There are a few differences between our implementation of GA_OLD and the one in [3]. For example, in [3] the mutations are only accepted with some probability that is experimentally set (ours only selects improvements), their cost function is defined slightly differently than ours, and the ω value is not described in enough detail to replicate in this work. Finally, the R_limit parameter is set to 1 for local mutations and R_limit is set to the maximum size of the FPGA array in one dimension for global mutations.

The two other GA algorithms that we have defined within our framework for our experiments are called GA_SS, which stands for steady state GA, and GA_SIMPLE, which stands for steady state GA. Table 2 columns three and four define the parameters for these two algorithms, respectively. The difference between the two algorithms is that GA_SIMPLE creates an entirely new population based on the fittest individuals (ω = 0) while GA_SS maintains the parents in each new generation. Finally, the R_limit parameter for these two algorithms is scheduled in the same way it is done for the SA in VPR 5.0. Initially, R_limit is set to the maximum size of the FPGA array in one dimensions. As the algorithm proceeds this value is decreased when there are no new individuals in the population that are better than the last population for the fitness function.

4. Comparison of Placement Results

In this section, our goal is to compare SA and GA for FPGA placement in the VPR 5.0 framework to optimize for power consumption and speed. We are not trying to discredit previous attempts at implementing a GA for FPGA placement, and as described in the previous section our implementation of the GA_OLD in [3] is not exact by any means. The two experiments that we run are, first, two compare the 4 placement algorithms (3 from the GA framework we have created) where in one case all the algorithms run for a set amount of time, and second, the GA algorithms are executed for a set number of generations. Before we present these results, we describe the experimental setup.

4.1 Experimental Setup

For our experimental setup, there are a two details that we must describe:

1) The architectural parameters describing the FPGA we are placing our benchmarks on
2) The computation system and conditions on which the algorithm is executed

The FPGA architectural parameters that describe the FPGA we are mapping to are shown in Table 3. For a more

Table 1: Configurable parameters for the GA

Parameter	Description of parameter
ω	The percentage of the fittest individuals in the population to use as parents
α	The percentage of the population created from the fittest individuals
β	The percentage of the population that is randomly created
σ	The number of individuals in the population
R_limit	The distance between swaps on the FPGA array
$global_swaps$	The percentage of the number of clusters that defines the number of global mutations for a new individual
$local_swaps$	The percentage of the number of clusters that defines the number of local mutations for a new individual
λ	A cost function parameter to weight timing optimization importance
γ	A cost function parameter to weight power optimization importance

Algorithm 1 The outline of the GA algorithm for placement

for $i = 1$ to σ **do** {Initialize the population}
  $new_population[i] = create_random_placement()$
end for

loop {For time –OR– For set number of generations}
  $R_{limit} = update_rlimit()$

  {Evaluates the populations based on the cost function and orders a list}
  $ranked_population = evaluate_population(new_population, \gamma, \lambda)$

  {Create the next generation}
  $current_spot = floor(\omega \cdot \sigma)$
  for $i = 1$ to floor($\omega \cdot \sigma$) **do** {Create the children from the best parents}
    $new_population[i] = ranked_population[i]$ {Copy the parents to the next generation}
    for $j = 1$ to $\alpha \cdot \omega \cdot \sigma$ **do**
      $new_population[current_spot] = mutate_genome_x_times(ranked_population[i], local_swaps, global_swaps, R_{limit})$
      $current_spot++$
    end for
  end for
  for $i = 1$ to floor($\beta \cdot \sigma$) **do** {Create random individuals}
    $new_population[current_spot] = create_random_placement()$
  end for
end loop

Table 2: Parameters for the three GAs for FPGA placement

Parameter	Value for GA_OLD	Values for GA_SS	Values for GA_SIMPLE
σ	3 * (number of clusters in a design)	3 * (number of clusters in a design)	3 * (number of clusters in a design)
ω	10% of σ	10% of σ	0% of σ
α	90% of σ	90% of σ	100% of σ
R_limit	1 and MAX	Variable	Variable
$global_swaps$	10%	20%	20%
$local_swaps$	10%	0%	0%

Table 3: The FPGA architectural parameters

Parameter	W	N	K	F_{cin}	F_{cout}	F_s	routing	transistor sizing
Value	20% larger than minimum	10	5	0.18	0.1	3	uni-directional	27mwt

detailed explanation of these parameters please consult [6], but for the sake of space and unnecessary details, we do not describe these parameters here. Note, the transistor size for these experiments is based on some our results in [18], which also describes how VPR 5.0 has been updated to support power estimation.

These experiments are run using thirteen of the MCNC benchmarks [19] where these benchmarks have been converted to a netlist of clusters using an academic CAD flow. Each benchmark is passed into VPR 5.0 for the same FPGA as described in Table 3. VPR 5.0 uses one of the 4 placement algorithms and then routes the design. The output of VPR

5.0 is the size of the FPGA (which is the same for each benchmark regardless of placement algorithm), the speed of the circuit, and the power consumption of the device.

This framework is run on a Intel Core Duo E8400 CPU with 2GB of RAM running at 3.00 GHz in Cygwin for Windows XP. For the experiments where run-time is kept constant VPR 5.0 gets uncontested access to one of the cores, and the only processing interference will be due to the operating system. This contention should be equal for all runs.

4.2 Fixed runtime results

In the first experiment, all four FPGA placement algorithms are run for a fixed amount of time, and this shows which algorithm finds the best solution for a given time limit. The time limit is based on how long the SA takes to execute for it's complete schedule. Our hypothesis, originally, is that the present design for the GAs, which is similar to the one proposed in [3] will not be as good as the SA in VPR 5.0 for minimizing critical path and power consumption.

Table 4 shows the results for power consumption and critical path for each of the MCNC benchmarks run over each of the four FPGA placement algorithms. Column one shows the MCNC benchmark name, and SA, GA_OLD, GA_SS, and GA_SIMPLE have the critical path and power consumption results over the following respective columns in groupings of two. In the final row of the table, we show the geometric average for all 13 benchmarks.

In terms of SA versus the GAs, it is clear that the SA produces better speed and power results compared to the three GAs. This confirms our original hypothesis. Comparing the GAs to each other it appears that the GA_SS is the best for optimizing the critical path and GA_SIMPLE is the best for power optimization. However, if we look at energy consumption compared to power consumption where energy is the power consumed for an amount of work done (a clock cycle in this case), then GA_SS is the best optimization algorithm.

If we look at the benchmark by benchmark results, there are a few cases when a GA has found a better solution than SA. For example, the GA_OLD placement has a better critical path for apex4 compared to SA. This result is not surprising as the GAs are searching a broader space and might find good solutions, but on average the SA algorithm outperforms the GAs. In [16] they claim that the benefit of SA is it converges faster, but tends to get stuck in local minimums. Our results are showing the same results for the GA framework described.

4.3 GAs run for a fixed number of generations

In [3], the GA was run for a fixed number of generations. We have also ran this experiment for our GA framework with the hypothesis that the GAs will produce better results than SA. We execute the GAs for 30 generations as this is the maximum number of generations shown in the results in [3].

Table 5 shows the geometric average of both the fixed runtime and the fixed generations experiments for each of the four algorithms. We have not included the individual benchmark results since this information does not add any value to our discussion.

These results show that given more time to execute, the GAs solutions are better for the most part, but the majority of the improved results improve the critical path of the circuits(likely due to the larger potential for improving this metric). Additionally, the improvements are not as significant as we hypothesized. The results are better than the fixed runtime experiments, but are still worse than the SA placement results.

Given sufficient runtime, we expect that the GA results will converge to those of the SA, and likely, be superior. Our results, however, suggest that how the genome and mutations are implemented for this framework is lacking. The mutation implementation, as a cluster swap, means that the population change is similar, if not the same, as how the simulated annealer works, and therefore, the GA approach is exploring multiple points in the search space for short amounts of time as opposed to SA which may do a bit of hill climbing, but is generally converging to the local minimum from which the random initialization starts. This is also in line with the points made in [16].

5. Conclusion

Our experiments show that the GA framework based on a fine grain mutation operator is no better than SA algorithms for FPGA placement. As discussed in the introduction, we expected that the SA would outperform the GA approaches currently implemented for FPGA placement. The main reason for this is that these GA implementations are searching over a range of optimization points in parallel while the SA is exploring one optimization point with significant depth (finding a local minimum).

These results, however, do not suggest that GAs do not have a part to be played in FPGA placement. Firstly, the approach proposed in [16] has value by using GA and SA in combination as a meta-heuristic. Also, since GAs lend themselves well to parallelization means that GA types of approaches may find significant value as the need for reduced CAD flow time increases [5].

From our perspective, we believe that the GAs for placement are operating on too fine a granularity. In the future, we hope to investigate how higher-level information, such as grouping of clusters, can be added to the GA framework to improve the results. This work presents a motivation to rethink how GA for FPGA placement can be implemented.

Table 4: Benchmark results for fixed-time execution

Benchmark	SA Critical Path in seconds	SA Power Consumption in Watts	GA_OLD Critical Path in seconds	GA_OLD Power Consumption in Watts	GA_SS Critical Path in seconds	GA_SS Power Consumption in Watts	GA_SIMPLE Critical Path in seconds	GA_SIMPLE Power Consumption in Watts
clma	$3.51E^{-8}$	2.21	$4.84E^{-8}$	3.04	$5.06E^{-8}$	2.90	$6.55E^{-8}$	2.55
spla	$2.44E^{-8}$	1.35	$3.02E^{-8}$	1.60	$3.15E^{-8}$	1.59	$3.43E^{-8}$	1.52
frisc	$2.32E^{-8}$	1.68	$3.22E^{-8}$	1.83	$3.16E^{-8}$	1.85	$4.12E^{-8}$	1.56
elliptic	$1.92E^{-8}$	1.44	$2.63E^{-8}$	1.76	$2.70E^{-8}$	1.72	$3.71E^{-8}$	1.38
ex1010	$3.09E^{-8}$	2.08	$3.82E^{-8}$	2.08	$3.77E^{-8}$	2.10	$4.56E^{-8}$	1.91
apex4	$2.03E^{-8}$	0.84	$1.95E^{-8}$	0.95	$2.14E^{-8}$	0.89	$2.22E^{-8}$	0.83
apex2	$1.87E^{-8}$	1.43	$2.12E^{-8}$	1.61	$2.06E^{-8}$	1.66	$2.78E^{-8}$	1.27
seq	$2.04E^{-8}$	1.25	$2.59E^{-8}$	1.25	$1.84E^{-8}$	1.66	$2.40E^{-8}$	1.38
misex	$1.66E^{-8}$	1.23	$1.90E^{-8}$	1.20	$1.73E^{-8}$	1.30	$2.23E^{-8}$	1.11
alu4	$1.58E^{-8}$	1.41	$1.81E^{-8}$	1.41	$1.74E^{-8}$	1.45	$1.96E^{-8}$	1.35
des	$1.49E^{-8}$	2.40	$4.07E^{-8}$	2.14	$2.72E^{-8}$	3.20	$3.29E^{-8}$	3.10
dsip	$1.13E^{-8}$	1.81	$1.72E^{-8}$	1.92	$1.19E^{-8}$	2.76	$1.94E^{-8}$	2.16
bigkey	$1.03E^{-8}$	0.77	$1.25E^{-8}$	1.40	$1.27E^{-8}$	1.40	$1.91E^{-8}$	1.06
geometric average	$1.90E^{-8}$	1.45	$2.50E^{-8}$	1.64	$2.30E^{-8}$	1.78	$2.94E^{-8}$	1.53

Table 5: Benchmark results for GAs executing over 30 generations

Benchmark	SA Critical Path in seconds	SA Power Consumption in Watts	GA_OLD Critical Path in seconds	GA_OLD Power Consumption in Watts	GA_SS Critical Path in seconds	GA_SS Power Consumption in Watts	GA_SIMPLE Critical Path in seconds	GA_SIMPLE Power Consumption in Watts
geometric average (fixed runtime)	$1.90E^{-8}$	1.45	$2.50E^{-8}$	1.64	$2.30E^{-8}$	1.78	$2.94E^{-8}$	1.53
geometric average (fixed generations)	$1.90E^{-8}$	1.45	$2.05E^{-8}$	1.64	$1.99E^{-8}$	1.72	$2.91E^{-8}$	1.55

References

[1] W. E. Donath, "Complexity theory and design automation," in *In Proceedings of the 17th Design Automation Conference*, 1980, pp. 412–419.

[2] D. H. Wolpert and W. G. Macready, "No Free Lunch Theorems for Search, REPORT SFI-TR-95-02-010," Tech. Rep., 1996.

[3] R. Venkatraman and L. M. Patnaik, "An evolutionary approach to timing driven fpga placement," in *GLSVLSI '00: Proceedings of the 10th Great Lakes symposium on VLSI*, 2000, pp. 81–85.

[4] J. Luu, I. Kuon, P. Jamieson, T. Campbell, A. Ye, W. M. Fang, and J. Rose, "VPR 5.0: FPGA CAD and Architecture xploration Tools with Single-Driver Routing, Heterogeneity and Process Scaling," in *ACM/SIGDA International Symposium on FPGAs*, Feb 2009.

[5] A. Ludwin, V. Betz, and K. Padalia, "High-quality, deterministic parallel placement for fpgas on commodity hardware," in *FPGA '08: Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, 2008, pp. 14–23.

[6] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, 1999.

[7] *Verilog Hardware Description Reference*, Open Verilog International, March 1993.

[8] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, no. 4598, pp. 671–680, May 1983.

[9] J. Kleinhans, G. Sigl, F. M. Johannes, and K. J. Antreich, "GORDIAN: VLSI Placement by Quadratic Programming and Slicing Optimization," *IEEE Transactions on Computer-Aided Design*, vol. 10, no. 3, pp. 356–365, Mar. 1991.

[10] D. Huang and A. Khang, "Partitioning-Based Standard-cell global placement with an Exact Objective," in *International Symposium on Physical Design*, Napa Valley, CA, 1997, pp. 18–25.

[11] A. E. Dunlop and B. W. Kernighan, "A Procedure for Placement of Standard-Cell VLSI Circuits," *IEEE Transactions on Computer-Aided Design*, vol. 4, no. 1, pp. 92–98, Jan. 1985.

[12] B. M. Riess and G. G. Ettelt, "Speed: Fast and Efficient Timing Driven Placement," in *IEEE International Symposium on Circuits*, 1995, pp. 377–380.

[13] K. Vorwerk, A. Kennings, and A. Vannelli, "Engineering details of a stable force-directed placer," in *ICCAD '04: Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, 2004, pp. 573–580.

[14] J. Lamoureux and S. J. E. Wilton, "On the interaction between power-aware fpga cad algorithms," in *ICCAD '03: Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design*, 2003, p. 701.

[15] P. Mazumder and E. Rudnick, *Genetic Algorithms for VLSI Design, Layout & Test Automation*. Prentice Hall, 1999.

[16] Y. Meng, A. E. A. Almaini, and W. Pengjun, "Fpga placement optimization by two-step unified genetic algorithm and simulated annealing algorithm," *Journal of Electronics (China)*, vol. 23, no. 4, pp. 632–636, 2007.

[17] K. Poon, "Power Estimation for Field-Programmable Gate Arrays," Ph.D. dissertation, University of British Columbia, 2002.

[18] P. Jamieson, W. Luk, S. J. Wilton, and G. A. Constantinides, "An energy and power consumption analysis of fpga routing architectures," in *International Conference on Field-Programmable Technology*, 2009, pp. 324–327.

[19] S. Yang, "Logic Synthesis and Optimization Benchmarks, Version 3.0," 1991, tech. Report. Microelectronics Centre of North Carolina. P.O. Box 12889, Research Triangle Park, NC 27709 USA.