Exploring Inevitable Convergence for a Genetic Algorithm Persistent FPGA Placer

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

Abstract— *Persistent CAD algorithms offer the potential to optimize power consumption for programmable chips post development and deployment. The basic idea is that algorithms continue to search for better design solutions and as better solutions are found, the design is deployed to programmable chips resulting in better performance. In this work, we further study a persistent placement algorithm for FPGAs and investigate a number of algorithm improvements attempting to delay premature convergence. Our results show that these techniques create some divergence in the solutions, but in all cases what we call "inevitable convergence occurs" in the first 2 hours of execution.*

Keywords: GA, Placement, FPGA, Persistent

1. Introduction

Field-Programmable Gate Arrays (FPGAs) are programmable Integrated Chips (ICs) that continue to gain popularity due to the challenges and costs associated with creating an Application-Specific Integrated Circuit (ASIC). One aspect of the FPGA, the programmability, means that these devices, similar to general purpose processors, can be reprogrammed. This means that even when the chip is deployed in the field, with the appropriate functionality, these devices can be updated with bug fixes, new designs, and as this work examines, more efficient designs. The last of these updates is part of what we call persistent Computer Aided Design (CAD) where optimization algorithms, which map designs to FPGAs, are run post chip deployment to try and find more optimal implementations of the design. These more optimal implementations focus on improving the power consumption of the FPGA.

We first introduced a persistent genetic algorithm (GA) for placement on FPGAs in [1]. This work showed how a genetic algorithm finds improved solutions early and the rate of these improvements slows as time increases. Our window of measurement for a relatively small benchmark was only one hour. In this work, we increase the time window of observation and use a variety of modifications to the GA to attempt to maintain diversity in the population and to delay what we call "inevitable convergence". Inevitable convergence, like premature convergence [2] [3], is the lack of diversity in a population such that new offspring are not sufficiently diverse, therefore, resulting in suboptimal solutions. In the case of persistent CAD, convergence is inevitable and is likely suboptimal. The goal, therefore, is to identify convergence (or measure population diversity [4], [5]) as well as avoid convergence. In this work we attempt techniques to deal with the later and leave the identification of convergence as future work.

To study inevitable convergence for persistent placement we implement the following algorithmic variations:

- Parallel algorithmic threads [6]
- A partial mapped crossover breeding operator [3]
- A partial mapped crossover breeding operator with mutations

Our results show that these techniques all tend to converge in about 2 hours. In all cases, when compared against the baseline random solution it is evident that the large gap between random solutions and these techniques means that it is highly unlikely that they will ever leave their local minimums.

The remainder of this paper is organized as follows. Section 2 briefly describes FPGAs, CAD for FPGAs, and the persistent FPGA placement problem. Section 3 describes our implementation of genetic algorithm placer. Section 4 describes our experimental setup and shows results for one 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 [7] 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.

Our open source CAD flow used by VPR 5.0 [8], which is an academic FPGA tool that allows us to experimentally test algorithms and FPGA architectures consists of Odin II [9] (high-level synthesis), ABC [10] (logic optimization

and technology mapping) and tv-pack [11] (clustering). First, a digital design is created in Verilog HDL [12] and used as the input to this CAD flow, and 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.

This work focuses on the FPGA placement step, which maps the design clusters onto the FPGA. This is the second last stage and is implemented in VPR 5.0. VPR 5.0, originally, used simulated annealing (SA) for this placement, and we have implemented a genetic algorithm within this software framework.

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 ([13], [7]), which is the algorithm used in VPR 5.0 [8], min-cut ([14], [15], [16]), analytic ([17], [18]), and genetic algorithms (GAs) [19], [20], and [21].

We focus on SA and GA algorithms in this work since our GA is built off the SA in VPR. The SA uses a cooling schedule to control the acceptance of randomly selected swaps between clusters on an FPGA. Each swap of clusters will either improve or degrade the critical path (as well as other metrics), 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 [7].

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).

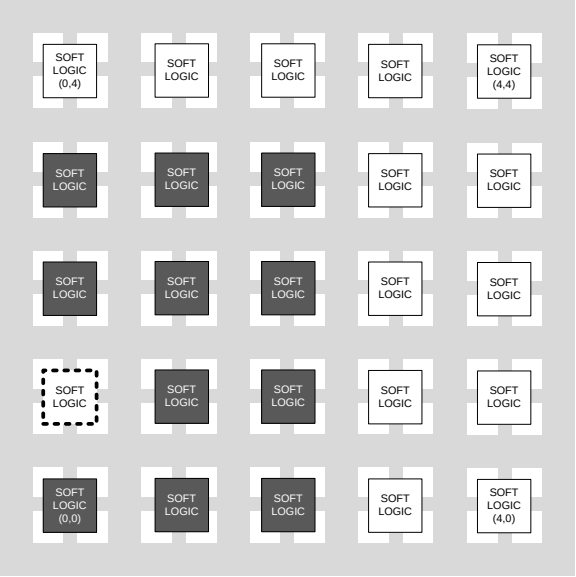


Figure 1: Shows how the R_{limit} factor affects the distance of swaps

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).

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 with power [22] consists of three components defined in [7]. 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:

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

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

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

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 power component is defined as:

$$PowerCost = \sum_{i=1}^{N} q(i) \cdot [bb_x(i) + bb_y(i)] \cdot Activity(i) \qquad (3)$$

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:

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

$$Cost = \lambda \cdot \frac{TimingCost}{PreviousTimingCost} + (1-\lambda) \cdot \left[(1-\gamma) \cdot \frac{WiringCost}{PreviousWiringCost} + \gamma \cdot \frac{PowerCost}{PreviousPowerCost} \right] \quad (4)$$

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 and the γ factor is used to control the importance of the power optimization component.

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 [23]. In our previous research [21] we confirm that these parameters are also suitable for our GA.

3. GA Framework and Modifications for FPGA Placement

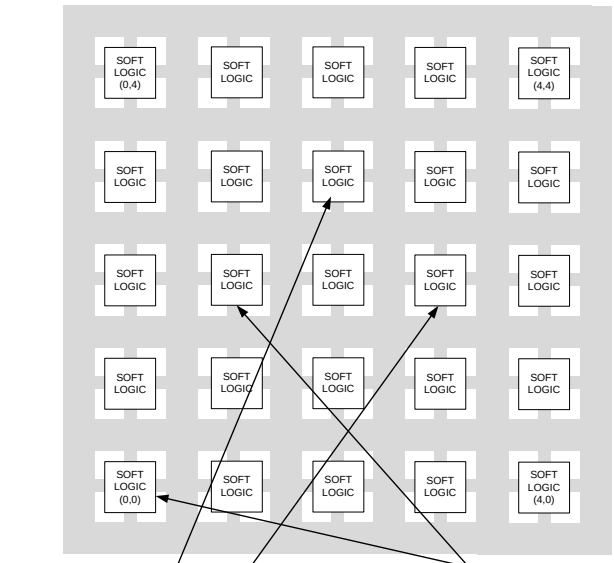
We previously built our genetic algorithm FPGA placer that includes power optimizations and describe the details of this algorithm here [21]. In this section we review some of these details, but do not cover all the details. In particular, we will describe genetic strings, the mutation operator, the crossover operator, and the parameters in the GA framework.

3.1 The Genome for Placement

Genetic algorithms (GA) and evolutionary programming algorithms have been previously implemented and explored for FPGA placement. We use a genome similar to the implementation by Venkatraman *et. al.* [19] 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.

3.2 Parameters in our GA Framework

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



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

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.

Our GA measures an individuals fitness based on the cost function shown previously in equation 4 and both λ and γ control the weighting of this cost function.

3.3 Mutation Operator

One of the operators in 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 this is similar to how the SA works, and therefore, this mutation is related to the term R_{limit}, which controls the distance between clusters 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.

R_{limit} is a parameter that changes over time. In the SA algorithm the parameter is decreased when a current set of swaps does not result in any improvement. In our GA, we

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

also schedule R_{limit} in a similar fashion, except that our algorithm is a persistent algorithm. Therefore, once R_{limit} equals 1 we then reset it to the maximum size of the FPGA array in one dimension.

3.4 Crossover Operator

The crossover operator within our GA framework is the partially mapped crossover (PMX) originally proposed by Goldberg for the traveling salesman problem [3]. This operator fits well with our placement algorithm since our string of clusters requires that each cluster appears only once in the genome string.

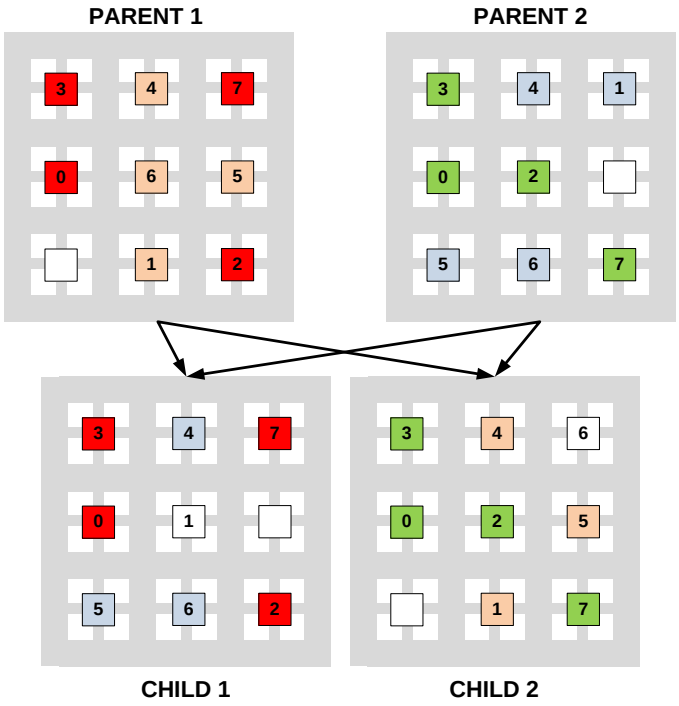


Figure 3: Sample PMX mutation

Figure 3 shows a sample crossover mutation for 2 parents generating 2 children. The figure has been color coded to show how the parent genes are crossed for the example picked genes 0, 2, 3, and 7. Note how in child 1 the gene 1 is not colored and child 2 the gene 6 is not color coded. In both of these instances, these gene locations are mapped by a series of remappings that the PMX operator achieves using a remapping list.

In our current implementation of the GA framework, two parents are chosen at random from the most fit percent of the population as specified by parameter ω to generate 2 children as part of the α new population. Within the genome, 50% of the clusters are randomly selected to stay constant from parent 1 to child 1 and parent 2 to child 2, respectively (in Figure 3 these clusters are in red. Then the PMX mapping is done on the remaining clusters to map parent 2's clusters to child 1 and parent 1's clusters to child 2 (as seen in Figure 3 in the orange and blue squares). The random swapping of components is not necessarily the best choice, and this is an area to study in the future.

4. Algorithmic Modification Results

To observe how different modifications to our persistent genetic algorithm for placement helps delay inevitable convergence we run the following experiment.

To attempt to maintain diversity we test the following modifications:

- Parallel algorithmic threads [6] - we introduce 4 threads initialized by different random seeds and execute these threads in parallel
- A partial mapped crossover breeding operator [3] - we use the PMX crossover to generate new individuals
- A partial mapped crossover breeding operator with mutations - we use the PMX crossover to generate new individuals and mutate these new individuals

Note that the β parameter is also a diversity factor, but this factor was already introduced in our first study of persistent placement with genetic algorithms, and it had very little impact. We do, however, maintain a 10% value for this parameter in our experiments.

Table 2 shows the parameters for our experiments. The first column shows the parameter, the second column shows the parameters value for the parallel GA threads, and column three shows the parameter values for the PMX crossover with

Table 3: The FPGA architectural parameters

Parameter	W	N	K	F_{cin}	F_{cout}	F_s	routing	transistor sizing
Value	144	10	5	0.18	0.1	3	uni-directional	27mwt

Table 2: Parameters for the GA parallel threads and Crossover

Parameter	Values for GA thread	Values for GA crossover
σ	200	200
ω	20% of σ	20% of σ
α	70% of σ	70% of σ
β	10% of σ	10% of σ
R_limit	Variable	Variable
$global\_swaps$	30%	30% and 0%*
$local\_swaps$	0%	0%
PMX	No	Yes
λ	0.5	0.5
γ	0.8	0.8

and without (* in table) mutations. Note that λ and γ have been experimentally determined based on out previous work.

The FPGA architectural parameters that describe the FPGA we are mapping to are shown in Table 3. For a more detailed explanation of these parameters please consult [7], but for the sake of space and unnecessary details, we do not describe these parameters here. The transistor size for these experiments is based on our results in [22]. For FPGA architects, note that W (channel width) is fixed to 144. This is done since we are only using one benchmark and the constant value considerably increases the speed of the routing algorithm. Instead of performing a binary search and increasing W by 20% (as is normally done in CAD experiments) we believe this fixed sized W is reasonable for this experiment considering that persistent CAD will, in reality, have W as a fixed parameter.

These experiments are run using the largest MCNC benchmarks [24] where these benchmarks have been converted to a netlist of clusters using an academic CAD flow. This benchmark, clma, is passed into VPR 5.0 for the same FPGA as described in Table 3. VPR 5.0 outputs the current best placement every 30 minutes and executes for 2 days. Once the algorithm is done, we then use these output placement files and run them through VPR's routing algorithm to find the final speed and power consumption metrics, which we report in the next section.

All the variations in the algorithms are run in Linux on Intel Xeon 2.4 Ghz cores.

4.1 Results

Figure 4 shows the results for our experiment. The x-axis shows the time over 2 days where we sampled the progress of the persistent placement algorithms every 30 minutes. On the y-axis, energy consumption is shown in terms of joules per clock. This metric is not truly reflective of instantaneous power consumption, but at present we do not have the capability to set the critical path in VPR 5.0.

In the figure, the upper line reflects the current best random placement result. We use this as a baseline noting that there is a significant difference between a random solution and the GA versions. Next, the orange line stabilizes after the first 30 minutes is the genetic algorithm with crossover and no mutation. Interestingly enough, in the first 30 minutes, diversity is completely eliminated since there are no mutations to introduce diversity back into the population. Depending on the random seed this happens at a different point.

The next collection of results are for the parallel threads. First, note that the lines (particularly "ga seed 99") sometimes seem to increase in energy consumption over the persistent exploration. The reason for this is due to the estimation models used to calculate the cost or fitness function at the placement level. These models are not as accurate as the models used after performing a complete placement and routing of a design (which are the results reported in this graph). In future work, it might be valuable to perform what we call a deep fitness evaluation at intervals to confirm population fitness at the placement level.

In general, the parallel approach shows that each thread finds solutions that are all in the similar energy range, but each thread itself seems to have converged relatively early in the search. Finally, the crossover and mutation operator performs similar to each of the parallel threads. The population eventually becomes dedicated towards a local minimum and cannot exit this area.

4.2 Discussion

A recent research paper by Mingjie and Wawrzynek [25] looked at the concept of tunneling to low energy (local optimal points) in SA for FPGA placement. Their claim is that once an SA finds a low energy area, no hill climbing technique can escape the localized search space. In our examples, the parallel threads and crossover mutations are in these low energy regions when we compare them relative to the random results, and our tunneling capabilities are essentially different start points (seeds).

To further persistent CAD for placement and maintain diversity longer, we will need to address the second part of the inevitable convergence problem, which is to identify when a population has converged. With this capability we can then explore if island model [26] for GAs or another solution will better suit are persistent CAD.

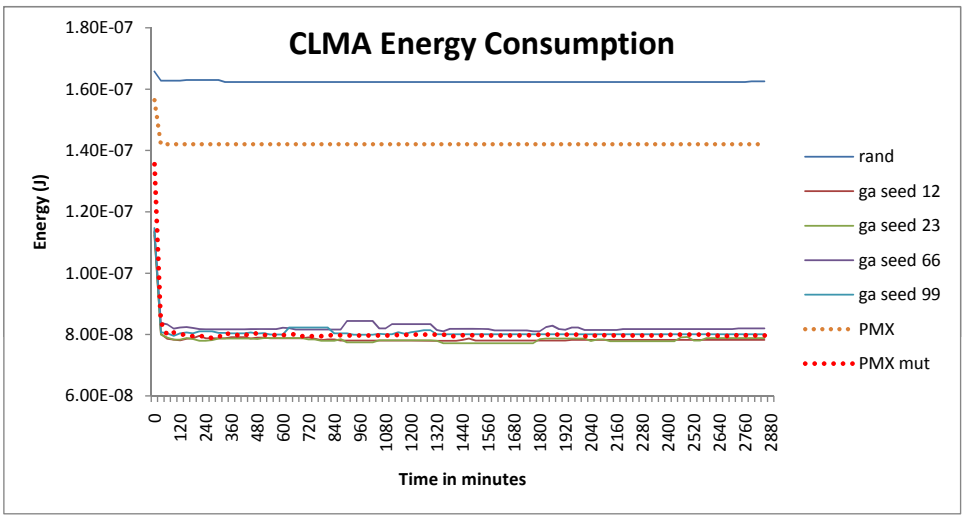


Figure 4: Energy consumption of CLMA benchmark over a 2 day execution

5. Conclusion

In this paper we explored various techniques for maintaining a divergent population in GA for persistent FPGA placement. Our results show that adding both parallel threads of GA populations and including a PMX crossover operator do not significantly impact the diversity of the population in this domain as in all cases the populations converge in a low energy search space. In future, we hope to take these techniques and incorporate them into a more complex system that will maintain divergence by searching in multiple islands of solutions and then use our techniques to mix these populations.

References

[1] P. Jamieson, "Persistent cad for in-the-field power optimization," in *ERSA*, 2010, pp. 267–270.

[2] M. Rocha and J. Neves, "Preventing premature convergence to local optima in genetic algorithms via random offspring generation," in *Proceedings of the 12th international conference on Industrial and engineering applications of artificial intelligence and expert systems: multiple approaches to intelligent systems*, 1999, pp. 127–136. [Online]. Available: http://portal.acm.org/citation.cfm?id=341506.341546

[3] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, 1st ed. Addison-Wesley Professional, January 1989. [Online]. Available: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20\&path=ASIN/0201157675

[4] Y. Leung, Y. Gao, and Z. Xu, "Degree of population diversity - a perspective on premature convergence in genetic algorithms and its markov chain analysis," *IEEE Transactions on Neural Networks*, vol. 8, pp. 1165–1176, 1997. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.33.3087

[5] E. Burke, S. Gustafson, and G. Kendall, "Diversity in genetic programming: an analysis of measures and correlation with fitness," *Evolutionary Computation, IEEE Transactions on*, vol. 8, no. 1, pp. 47 – 62, 2004. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1266373

[6] H. Mühlenbein, M. Schomisch, and J. Born, "The parallel genetic algorithm as function optimizer," *Parallel Computing*, vol. 17, no. 6-7, pp. 619 – 632, 1991. [Online]. Available: http://www.sciencedirect.com/science/article/B6V12-4GMBN2V-2/2/e78d51ea3dd47a6e6be1e93a23edead6

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

[8] 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. [Online]. Available: http://doi.acm.org/10.1145/1508128.1508150

[9] P. Jamieson, K. B. Kent, F. Gharibian, and L. Shannon, "Odin II - An Open-source Verilog HDL Synthesis tool for CAD Research," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2010, pp. 149–156. [Online]. Available: http://www.computer.org/portal/web/csdl/doi/10.1109/FCCM.2010.31

[10] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "Improvements to technology mapping for LUT-based FPGAs," *IEEE Transactions on CAD*, vol. 26, no. 2, pp. 240–253, 2007. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.68.9003

[11] A. Marquardt, V. Betz, and J. Rose, "Using Cluster-Based Logic Blocks and Timing-Driven Packing to Improve FPGA Speed and Density," in *ACM/SIGDA International Symposium on FPGAs*, Monterey, CA, 1999, pp. 37–46. [Online]. Available: http://doi.acm.org/10.1145/296399.296426

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

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

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

[21] P. Jamieson, "Revisiting genetic algorithms for the fpga placement problem," in *GEM*, 2010, pp. 16–22.

[22] 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. [Online]. Available: http://www.users.muohio.edu/jamiespa/html_papers/fpt_09.pdf

[23] 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.

[24] 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.

[25] M. Lin and J. Wawrzynek, "Improving fpga placement with dynamically adaptive stochastic tunneling," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 29, no. 12, pp. 1858 –1869, 2010. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5621041&tag=1

[26] R. E. Smith, S. Forrest, and A. S. Perelson, "Searching for diverse, cooperative populations with genetic algorithms," *Evol. Comput.*, vol. 1, pp. 127–149, June 1993. [Online]. Available: http://dx.doi.org/10.1162/evco.1993.1.2.127