New Genetic Operators to Facilitate Understanding of Evolved Transistor Circuits

Martin Trefzer, Jörg Langeheine, Johannes Schemmel, Karlheinz Meier
University of Heidelberg
Kirchhoff-Institute for Physics
Im Neuenheimer Feld 227, 69120 Heidelberg, Germany
martin.trefzer@kip.uni-heidelberg.de, +49 (0)6221 54-9838
http://www.kip.uni-heidelberg.de/vision/projects/eh/

Abstract

In this paper new genetic operators are introduced that inherently avoid floating terminals and broken routes while evolving transistor circuits on a CMOS field programmable transistor array (FPTA). They are designed to facilitate understanding and improve transferability of the resulting circuits. Comparators and logic gates (AND, OR, XOR) have been evolved with the proposed algorithm and the results are compared to corresponding experiments that use a straight forward implementation of the genetic operators. Furthermore, netlists are extracted from the evolved circuits and simulated with a SPICE simulator. The simulation results are compared with measurements performed on the chip.

1 Introduction

Genetic algorithms (GAs) and genetic programming (GP) are used in various implementations for analog circuit synthesis. Two fundamental approaches are widely used: First, the evaluation of the evolving circuits is done by a software simulation, which is mostly in conjunction with GP, or second, the evaluation is done by configurable hardware to which the genotype representation can be mapped. On the one hand, simulation offers full flexibility to the circuit topology while hardware substrates, e.g. analog arrays (AAs) [8] [3], suffer from inherent constraints regarding the placing and routing of the components. On the other hand, valuable evaluation time is significantly higher for software simulation than for an AA that provides the circuits output almost instantly. Therefore, the philosophy behind using hardware-in-the-loop is to be able to process more generations in less time to compensate for the flexibility of software and to obtain directly a ready to use circuit.

Evolvable hardware aims to generate circuits from which new ideas and concepts of electronic design can be derived. To achieve this goal, the produced circuits have to be robust against environmental influences and independent of the evolution system. Circuits — evolved on AAs — tend to exploit inherent characteristics of the particular substrate and therefore lack the desired properties. Hence, most evolved circuits so far are difficult to understand and it is most often impossible to transfer them to other technologies. In the case of AAs, which consist of identical functional blocks that can be interconnected and configured in various ways, this is due to commonly used straight forward implementations of the GA: The mutation operation flips random configuration bits and the crossover copies parts from one individual to another without taking the environment into account. For our representation of the genotype this means enabling or disabling random connections and random configuration of the transistors. This leads to circuits that contain a large number of floating terminals and discontinuous routing. To overcome these problems, genetic operators can be implemented that contain knowledge about the phenotype structure ([10] describes an approach of introducing knowledge to the evolution process). For example, the genetic algorithm can evolve circuits following the constraint of avoiding floating nodes. Other approaches are made using knowledge about circuit design itself (methods of automated circuit design and current-flow analysis are applied in [5] and [7]). Another possibility is to grow circuits from small units either by consecutively connecting the available inputs and outputs as described in [6], or by using a GP representation as introduced in [2].

In this paper new genetic operators are introduced that inherently avoid floating terminals and broken routes while mutating the circuit. The GA that uses the new operators is referred to as the *Turtle GA* throughout this paper. Thus, in case of the FPTA, the main feature of the *Turtle GA* is to produce circuits that are reduced to relevant components and therefore are easier to understand according to engineering criteria. SPICE netlists [9] of the evolved circuits are generated and simulated outside the FPTA. The simulation results are compared with the measurements on the chip. In order to be able to derive new design concepts and generally reusable solutions from evolved FPTA circuits, it is necessary to accurately evaluate them.

The first task is to evolve logic gates (AND,OR,XOR), which has been successfully done in previous experiments [4] with a straight forward implementation of the GA. Therefore, the logic gates — especially the AND / OR — are suitable for testing the *Turtle GA*. The second task is to evolve comparators. This is not only interesting because it is quite difficult to find good solutions, but also, contrary to the logic gates, the problem is of analog nature. Furthermore, to the authors knowledge a comparator has not yet been successfully evolved. The performance of the *Turtle GA* is compared to the straight forward implementation of the GA used in [4], which is referred to as the *Basic GA* throughout the rest of this paper.

2 Evolvable Hardware System

The evolution system [3] consists of three parts: First, the FPTA that hosts the configurable CMOS transistor array. Second, a controller for uploading the individuals to the FPTA, applying the test patterns and measuring the outputs. Third, a PC that runs the GA and configures the controller. Thus, the PC generates the analog test patterns that are to be applied to the FPTAs inputs and transfers them to the RAM of the controller. Subsequently, the individuals — representing configuration strings for the transistor array — are transferred to the controller; it configures the FPTA and measures the output of the current individual by using the previously defined test patterns as input. Once the measurement is completed, the PC reads back the results and calculates the fitness value of the corresponding individual. After the whole generation has been evaluated, the GA creates the new generation out of the current one. These components provide a real time test environment for the evolved circuits.

For an easier understanding of how the presented implementation of the GA works, a closer description of the transistor array is necessary: The array consists of 16x16 configurable CMOS transistor cells (Fig. 1). Half of the cells are designed as programmable PMOS and NMOS transistors respectively and are arranged in a checkerboard pattern. Width *W* and length *L* of each transistor is adjustable within wide ranges ($W = 1, 2, ..., 15\,\mu\text{m}$, $L = 0.6, 1, 2, 4, 8\,\mu\text{m}$). Its terminals (source, drain and gate) can be connected to one of the cells outside connections (N,S,W,E), vdd or gnd. Additionally, it is possible to directly connect the nodes (N,S,W,E) to each other, which provides routing capabilities. Owing to the four nodes avaliable for routing and terminal connections, one cell mostly serves either as transistor cell or routing cell. However, both capabilities are not separated. The array is enclosed by IO cells that can apply voltages to the border cells or measure the output voltages of the evolved circuit. Please refer to [3] for a detailed description of the FPTA.

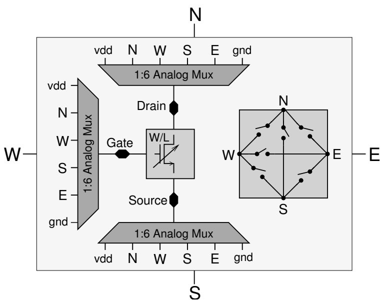

Figure 1. The block diagram of an FPTA MOS transistor cell.

3 The Genetic Algorithms

3.1 Operation Principle of the Turtle GA

Only circuits without floating nodes can be extracted into netlists and transferred to other technologies. As a matter of course, the genetic operators that are called by the *Turtle GA* have to ensure that no node or terminal remains disconnected. The *Turtle GA* is a recursive algorithm that features these properties. It 'draws'—analog to a *Graphics Turtle*—random parts of a circuit directly on the phenotype representation of the transistor array. The changes made by the *Turtle GA* are then mapped back to the genotype. For each arising open end the algorithm is recursively called until the destination node no longer represents a floating terminal or open route. Three genetic operators are available: One mutation operator (*random wires*) and two crossover operators (*implanting of foreign cells* and *logic OR*). The W/L values of the transistors are independently changed due to a given mutation rate.

3.1.1 Random Wires (Mutation)

The mutation operator randomly selects an outside node of an arbitrary cell to be the starting point for the algorithm. For such a node both, the cell and the adjacent neighbor cell, provide six possible connections: Three routing connections to the remaining outside nodes and three terminal connections to the transistor nodes. Nodes are recursively connected to (or — in erase mode — disconnected from) its

```
StartRandomWire(){

 RandomSelectTransistorCell()
 RandomSelectStartNode( return N,S,W or E )
 RandomSelectDestNode ( return N,S,W,E or
                        Gate,Source,Drain )

 if (StartNode is connected to DestNode)
  gaMode:=erase
  EraseConnection()
 elseif (not connected)
  gaMode:=create
  EnableConnection()
 endif

 if (DestNode is N,S,W or E)
  RecurseRandomWire (gaMode,DestNode)
 elseif (DestNode is transistor terminal)
  RecurseRandomTerminal (gaMode,DestNode)
 endif

 RecurseRandomWire (gaMode,CurrentNode)
}

RecurseRandomWire(gaMode,DestNode){

 if (gaMode is erase)
  RandomDecideWheterToProceed( return stop )
  if (No of node connects $=0$ or $=2$ or stop$=$TRUE)
   End recursion and return.
  else
   RandomSelectConnectedDestNode ( return N,S,W,E
                                   or Gate,Source,Drain )
   EraseConnection()
  endif
 else
  if (No of node connects>1)
   End recursion and return.
  else
   RandomSelectNotConnectedDestNode ( return N,S,W,E
                                      or Gate,Source,Drain )
   EnableConnection()
  endif
 endif

 if {DestNode is N,S,W or E}
  RecurseRandomWire (gaMode,DestNode)
 elseif {DestNode is Gate, Source or Drain}
  RecurseRandomTerminal (gaMode,DestNode)
 endif
}
```

```
RecurseRandomTerminal(gaMode,DestNode){

 for (Both remaining nodes (terminals))
  RandomSelectDestNode ( return N,S,W,E )
  if (gaMode is erase)
   EraseConnection()
  else
   EnableConnection()
  endif

  RecurseRandomWire (gaMode,DestNode)
 endfor
}


CrossImplant(){
 RandomSelectTwoIndividuals();
 RandomSelectBlockOfCells from Ind.2 (return Block;);
 InsertBlockOfCells into Ind.1 ();
 forall (BorderNodes)
  if (NoNodeConnections==1)
   StartRandomWire();
  endif
 endfor
}


CrossLogicOR(){
 RandomSelectCrossPartner();
 forall (Terminals and Routes)
  if (Connection enabled in Ind.1 OR Ind.2)
   EnableConnection() in offspring;
  endif
 enfor
}
```

Figure 2. Pseudocode implementation of the genetic operators of the Turtle GA.

arbitrary neighbor until the circuit is closed again. The basic operation principle of the mutation is described in pseudocode (Fig. 2). An example of how the mutation operator enables one transistor and corresponding routing is shown in Fig. 3. In the following, only connected transistors are shown in Fig. 3, 4 and 5.

3.1.2 Implanting a Foreign Block of Cells (Crossover)

The *implanting* crossover operator processes two stages: In the first stage, a crossover partner is selected, from which a randomly sized and positioned rectangular block of cells is copied to the current individual (Fig. 4 A+B). Since this operation in general breaks the layout of the previously intact circuit, the second stage takes care of fixing the occurring floating nodes according to the FPTAs structure as shown in Fig. 4 C. The implementation is described in Fig. 2.

3.1.3 Logic OR of Individuals (Crossover)

The *logic OR* crossover operator calculates the logic OR of the selected crossover partner and the current individual as can be seen from Fig. 5. Thus, the features of both individuals are combined. If a transistor is present in both circuits, the W/L values are taken from the current individual. In Fig. 2 the implementation is described in pseudocode.

Applied to highly diverse individuals, this results in a strong impact on the individuals structure. On the one hand, this usually changes the circuits output completely. On the other hand, since the logic OR does not destroy previous structures, it enriches the diversity of the individuals within the population and is therefore helpful in avoiding local minima.

3.2 The Basic GA

The *Basic GA* is more closely described in [4] and based on a simple genetic algorithm introduced in [1].

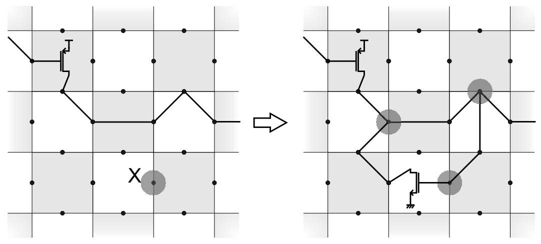

Figure 3. Principle of the Random Wires Operator. The start node is marked with an X.

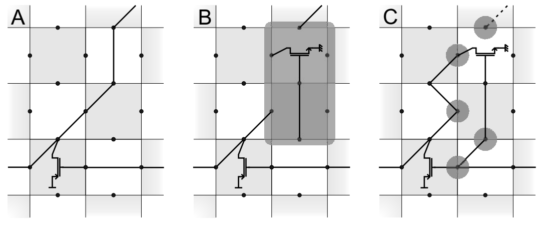

Figure 4. Principle of the implanting crossover operator.

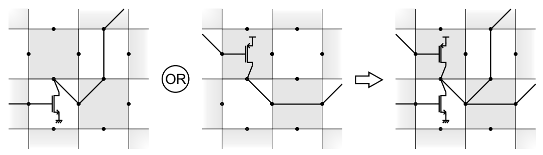

Figure 5. Principle of the logic OR crossover.

The mutation operator randomly changes every connection in every cell of an individual due to a probability given by the *mutation rate*.

The crossover operator works on cell level and inserts a rectangular block of cells of a selected crossover partner into the current individual. Size and position of the block are randomly chosen. The execution of the crossover operator is adjusted by a probability given by the *crossover rate*.

3.3 GA Parameters

Tournament selection with a tournament size of 7 is used in the presented experiments. Crossover and mutation rates are calculated proportionally to the candidates current fitness. Scaling down the rates provides high mutation and crossover probabilities for fast exploration in the beginning of evolution and low probabilities for fine tuning of the circuit when a good solution is found. The GA parameters used throughout the presented experiments are listed in Tab. 1. The '–' indicates that the operation is not available to the respective GA.

4 Experimental Setup

An area of 7×7 (8×8) transistor cells is used for all experiments. The circuits inputs are applied to the west side while the output voltage is measured on the south side. The GA parameters are set according to Tab. 1 and in all cases 30 evolution runs are carried out. All experiments have been performed with both, the *Basic GA* and the *Turtle GA*.

GA Parameter	logic gates basic/turtle	comparators basic/turtle
gen. size	50	50
no. of gen.	20000	20000
mut. fraction	0.6	0.6
mut. rate	4 . . . 0.8 %/ –	4 . . . 0.8 %/ –
rand. wires rate	– /2.5 . . . 0.5 %	– /1.25 . . . 0.25 %
cross. fraction	0.6	0.6
cross. rate	1 . . . 0.2 %	1 . . . 0.2 %
cross. block size	4x4	4x4
cross. rate (OR)	– /1 . . . 0.2 %	– /1 . . . 0.2 %

Table 1. Genetic algorithm parameters used throughout the presented experiments.

4.1 Experimental Setup for the Logic Gates

In these experiments, the task is to evolve one of the more complex logic gates, namely AND, OR and XOR. The output target voltage had to be $V_{\text{out}} = 0\,\text{V}$ (= logic zero) or $V_{\text{out}} = 5\,\text{V}$ (= logic one) depending on the computational result of V_{in1} AND (OR, XOR) V_{in2}. A set of ten curves, each consisting of 64 sample voltages, is used in the test pattern depicted in Fig. 6. The transition region is not considered during evolution in order to facilitate the search for good solutions. This is admissible, because the specification of logic gates demands a fast and correct decision depending on digital input voltages that are outside the transition region, e.g. $V_{\text{in1/2}} < 2\,\text{V}$ and $V_{\text{in1/2}} > 3\,\text{V}$. This test pattern is only used during evolution. For testing, the voltages in the transition region are measured as well to obtain the full characteristic curve of the output voltage.

4.2 Experimental Setup for the Comparators

The task is to evolve a comparator. That is, the output target voltage has to be $V_{\text{out}} = 0\,\text{V}$ if $V_{\text{in1}} < V_{\text{in2}}$ and

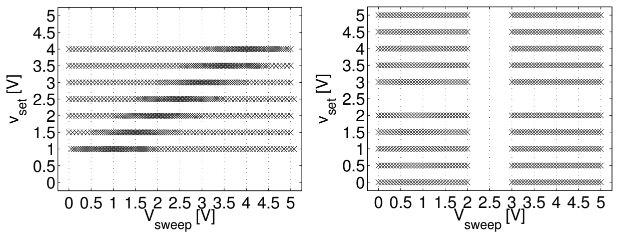

Figure 6. Input voltage pattern used for comparators (left) and logic gates (right).

$V_{\text{out}} = 5\,\text{V}$ if $V_{\text{in}1} > V_{in2}$. A set of seven curves, each consisting of 90 sample voltages, is used in the test pattern illustrated in Fig. 6. The switching points are set to $V_{\text{set}} = 1, 1.5, 2, \ldots, 4\,\text{V}$. Within a range of $V_{\text{set}} \pm 1\,\text{V}$ the density of the sample points for V_{sweep} increases towards the switching point. Thereby, a high emphasis on the transition region is achieved. The increasing density of sample points is shown in Fig. 6, left. The remaining range is covered by equally spaced sample points. For measuring the voltage characteristics, a continuous linear ramp is used for V_{sweep} (in steps of $20\,\text{mV}$) in order to facilitate calculation of RMS, offset and gain.

4.3 Fitness Calculation

Different fitness functions are used for the evolution of logic gates and comparators. In all fitness functions the range of $V_{\text{target}} - V_{\text{out}} = 0 \ldots 5\,\text{V}$ is divided into 11 intervals, the upper limits of which represent a threshold for additional penalties. The penalty schemes of both fitness functions are illustrated in Fig. 7. Finally, the fitness value is calculated as follows:

$$\text{fitness} = \sum_{i=1}^{\#\text{samples}} \text{penalty}_i. \qquad (1)$$

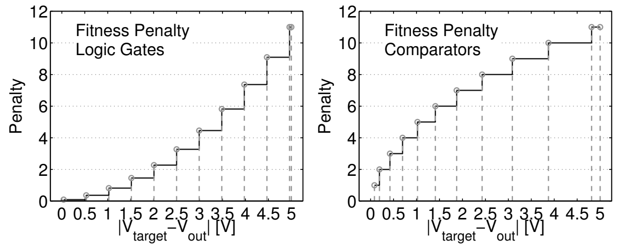

Figure 7. Fitness penalty for each sample. Left: logic gates. Right: comparators.

For calculating the penalty in case of the comparators, the absolute value of $V_{\text{target}} - V_{\text{out}}$ is taken. This makes the GA exploit even small changes of the output as long as the fitness is high and preserve found solutions if it is low. For the logic gates the absolute value of $V_{\text{target}} - V_{\text{out}}$ is quadratically weighted. Otherwise, for problems (e.g. the AND gate) with non equally distributed states, the case of all output voltages stuck at $0\,\text{V}$ (logic zero) would result in a better fitness than all at $5\,\text{V}$ (logic one). Depending on how the GA explores the solution space, this would already result in a local minimum right in the beginning.

In order to make the algorithm more stable under the influence of noise and fluctuations of the analog output, discrete fitness functions are used. The first and last intervals are set to $V_{\text{thresh}} = 0.04\,\text{V}$ and $V_{\text{thresh}} = 4.96\,\text{V}$ considering the precision of the applied voltages. Measurements have shown that a precision of at least 8 bits can be assumed for the measurements.

Additionally, in all experiments minimization of used resources is included in the fitness by adding extra penalty. In the phase of exploration minimizing the resources would be counterproductive. Hence, below a fitness threshold of 500 for the comparators and 700 for the gates an offset penalty of $1\times$ no. of used routes + $2\times$ no. of used transistors is added. The maximum offset penalty is calculated by inserting the amount of all available resources. By setting the additional penalty to the maximum above this threshold, it is ensured that a better fitness always represents a better circuit. In the presented fitness values the offset penalty is replaced by the penalty calculated from the actually used resources.

4.4 Simulation Setup

Spice netlists are extracted from the circuits that have been evolved with the *Turtle GA*. The simulations are carried out with the SPICE3 simulator, described in [9]. Basic transistor models are used for computation and the resistance of the switches is approximated by its mean value. The input voltage patterns correspond to those used for the on-chip test measurements of the logic gates and the comparators respectively.

5 Evolution Results of the Logic Gates

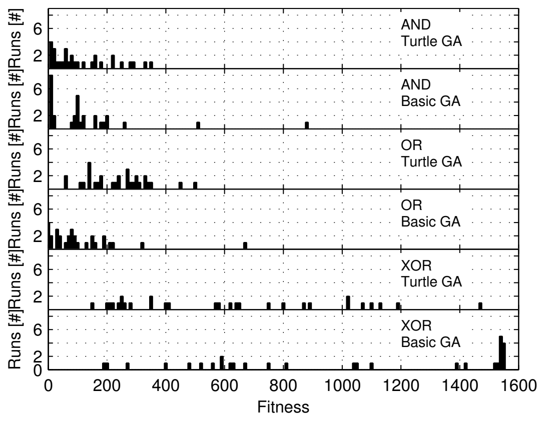

Figure 8. Results for the evolution of different logic gates using both GA representations.

The range of fitness values of the logic gates covers $0 \ldots 7040$. Each run is initialized with a random genera-

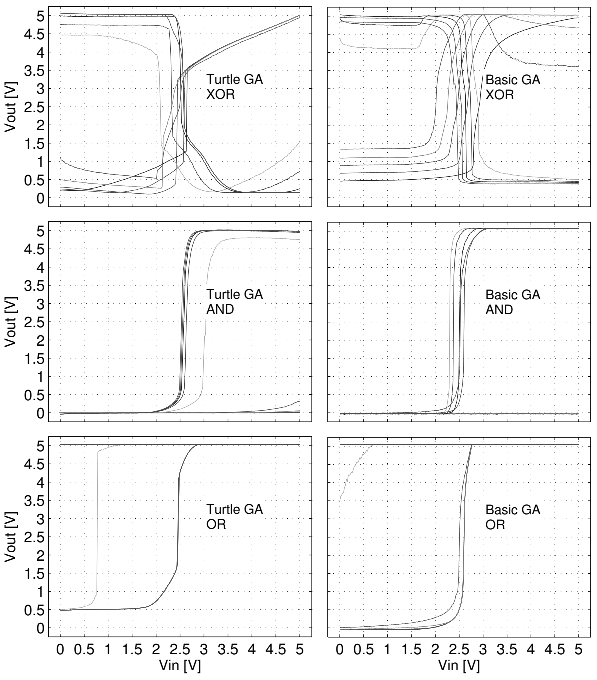

Figure 9. Voltage characteristics of the best evolved logic gates (AND, OR, XOR).

tion and most individuals start with a fitness value of about 3500. The fitness values of the experiments are shown in the histograms in Fig. 8. As observed in other experiments [4], the XOR is the most difficult of the presented logic gates to evolve. Neither of the algorithms is able to find good solutions for it. Contrary to that, both algorithms succeeded in reliably finding solutions for the AND and the OR. Considering the pure fitness values, the *Basic GA* performed slightly better in evolving OR gates than the *Turtle GA*.

For both algorithms all experiments ended with similar fitness values, where the best are in the order of 50 for the AND / OR and in the order of 200 for the XOR. As can be seen from Fig. 9, the measured output voltage schemes of the best circuits look quite similar for the *Basic GA* and the *Turtle GA*.

During evolution, the samples of the test pattern are randomly applied with a frequency of $0.25\,\mathrm{MHz}$ which corresponds to switching the output at least within $4\,\mu$s. The best logic AND / OR (Fig. 9) are also performing well under a test frequency up to $0.9\,\mathrm{MHz}$ and therefore are able to switch within $1.1\,\mu$s.

5.1 Comparison of the Results of Both GAs

With regard to the used resources, it can be seen from Fig. 11 that the circuits produced by the *Turtle GA* are substantially improved compared to the solutions found by the *Basic GA*. On average, the circuits evolved with the *Turtle GA* use only one-quarter of the transistors than those evolved with the *Basic GA* at comparable RMS error. It is expected that circuits (netlists) with less components and circuits that are proven to operate on the chip as well as in simulation can be easier converted to human readable schematics. Therefore, such circuits will be easier to understand according to engineering criteria. The first two steps are successfully done by the *Turtle GA*.

5.2 Simulation Results for the Logic Gates

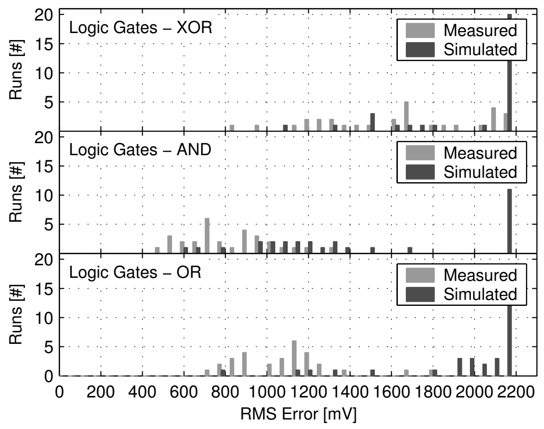

Figure 10. Comparison of the evolution results and the corresponding circuit simulations.

The simulation results of the logic gates are compared with the on-chip measurements in Fig. 10. For all evolved gates the simulation results are worse than those obtained from the measurement on the FPTA. As simulation shows, about 30% of the gates do not work at all outside the transistor array. Despite of that, the best logic gates perform at least similar in simulation and on the FPTA. In case of the AND gate the simulation results correspond nearly perfectly to the measurement. The simulation results of the voltage characteristics are shown in Fig. 11.

6 Evolution Results for the Comparators

For the comparators the range of the fitness values covers $0 \dots 6930$. The observed initial fitness of each individual is about 3500 and all runs are started with a random generation.

The fitness values of the experiments are shown in the histograms in Fig. 12. As can be seen, it is possible to

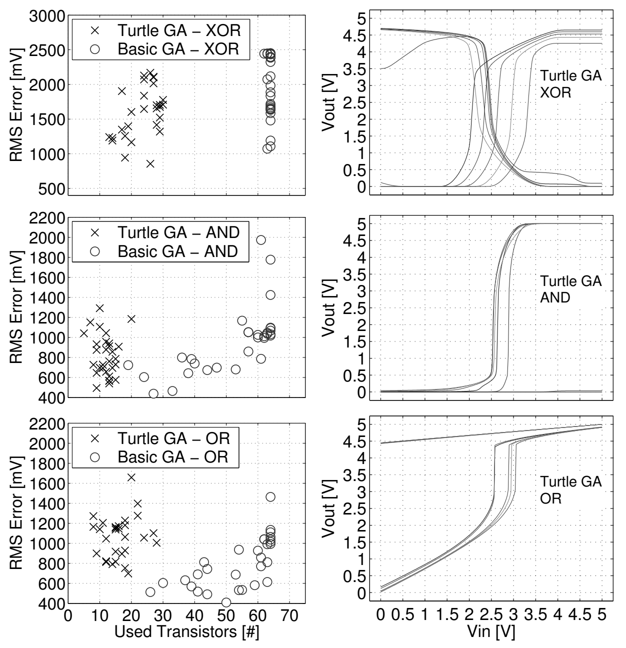

Figure 11. Left: Comparison of the RMS fitness over the number of used transistors for the logic gates. Right: Voltage characteristics of the logic gates featuring the best simulation fitness values.

evolve good comparators with either of the GAs considering the pure fitness value.

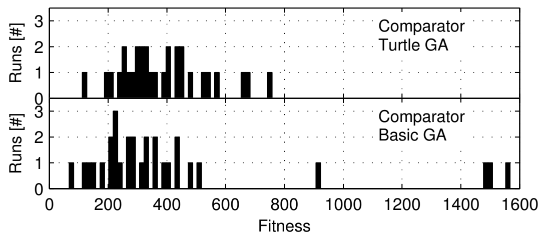

Figure 12. Results for the evolution of comparators with both GA representations.

Since all experiments ended with similar fitness values where the best are in the order of 200, the voltage characteristics look quite similar as well. The measured output voltages of the best evolved circuits are plotted in Fig. 13. The comparator evolved with the *Turtle GA* has a gain of 65 ± 33; the one evolved with the *Basic GA* of 90 ± 25. Taking the errors into account, the gain of the comparators is similar. Both have an offset of at most $20\,\mathrm{mV}$ which corresponds to the available accuracy. Gain and offset are calculated as the average value of the set of curves.

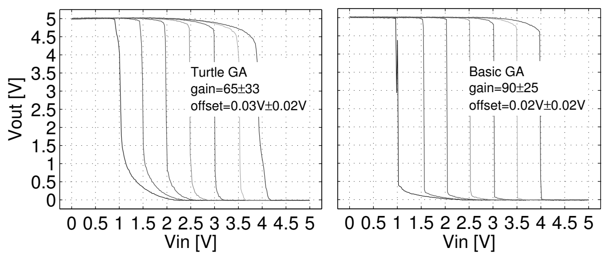

Figure 13. Voltage characteristics of the best evolution run for the comparators.

During evolution the test pattern is randomly applied with a frequency of $0.2\,\mathrm{MHz}$ which ensures a settling time of at least $5\,\mu$s. The best circuits (Fig. 13) perform equally well when using a test frequency of $0.8\,\mathrm{MHz}$ and therefore feature a settling time of $1.3\,\mu$s.

6.1 Comparison of the Results of Both GAs

Once again, with regard to the resource requirement, it can be seen from Fig. 15 that the circuits produced by the *Turtle GA* are extensively improved compared to the solutions found by the *Basic GA*. On average, the comparators evolved with the *Turtle GA* use only one-third of the transistors allocated by the *Basic GA* at equal RMS error. Thus, the *Turtle GA* achieved to minimize the number of used transistors and routes in case of the comparators and the logic gates described in Sect. 5.1.

6.2 Simulation Results for the Comparators

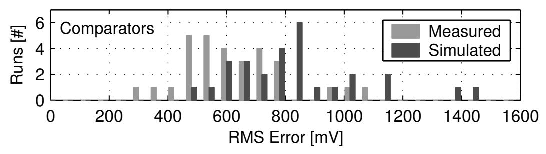

Figure 14. Comparison of the evolution results and the corresponding circuit simulations.

The histogram shown in Fig. 14 compares the RMS errors of the evolved circuits obtained directly from the measurement on the FPTA with those calculated from the simulation result. As expected, the circuits perform worse in

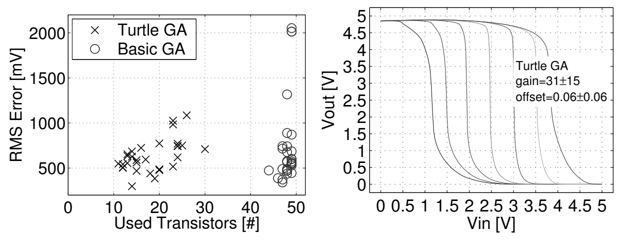

Figure 15. Left: Comparison of the RMS error over the no. of used transistors for both GA representations. Right: Voltage characteristic of the comparator featuring the best fitness in simulation.

simulation than in measurement. Two circuits with an RMS Error of about $1400\,\mathrm{mV}$ did not work at all in simulation.

However, as can be seen from Fig. 15, the voltage characteristic of the best comparator is similar to that shown in Fig. 13. The gain is still 30 ± 15, while the offset has doubled $0.06 \pm 0.06\,\mathrm{V}$. This is a promising result, because it suggests that even more complex circuits can be extracted into netlists and simulated.

7 Conclusions and Future Work

A GA with new genetic operators — the *Turtle GA* — is introduced and compared with a straight forward implementation of the GA. Comparators and logic gates have been successfully evolved with both algorithms. While the voltage characteristics of the best circuits perform equally well in both cases, the *Turtle GA* substantially reduced the required resources in all experiments; the number of used transistors decreased on average about 70%. The fact that the *Turtle GA* achieved to reduce resource allocation in both cases suggests that successful application is not restricted to a specific problem. With the help of the *Turtle GA*, floating gates and discontinuous routing can be inherently avoided in the evolved circuits. Therefore, the evolved circuits are extracted into netlists and simulated. The simulation results are compared with the on-chip measurements. The comparators performed slightly worse, whereas some of the logic gates do not work in simulation. Nevertheless, the best circuits performed well in both, simulation and on-chip measurements. Thus, it has been proven that it is possible to evolve circuits on the FPTA which can be transfered to other technologies with the new implementation of the GA. Future work will be done to find a more accurate equivalent circuit for the FPTAs cells to further improve the quality of simulations of evolved circuits. Additionally, human readable schematics will be generated to advance the understanding of the circuits according to engineering criteria.

8 Acknowledgment

This work is supported by the Ministerium für Wissenschaft, Forschung und Kunst, Baden-Württemberg, Stuttgart, Germany.

References

[1] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[2] John E. Koza, Forrest H. Bennet III, David Andre, Martin A. Keane. *Genetic Programming III: Darwinian Invention and Problem Solving*. Morgan Kaufmann Publishers, 1999.

[3] J. Langeheine, J. Becker, S. Fölling, K. Meier, and J. Schemmel. A CMOS FPTA chip for intrinsic hardware evolution of analog electronic circuits. In *Proc. of the Third NASA/DOD Workshop on Evolvable Hardware*, pages 172–175, Long Beach, CA, USA, July 2001. IEEE Computer Society Press.

[4] J. Langeheine, K. Meier, and J. Schemmel. Intrinsic Evolution of Quasi DC Solutions for Transistor Level Analog Electronic Circuits Using a CMOS FPTA chip. In *Proceedings of the 2002 NASA/DoD Conference an Evolvable Hardware*.

[5] J. D. Lohn and S. P. Colombano. A circuit representation technique for automated circuit design. *IEEE Transactions on Evolutionary Computation*, 3(3):205–219, September 1999.

[6] J. F. Miller and P. Thomson. A developmental method for growing graphs and circuits. In A. M. Tyrrell, P. C. Haddow, and J. Torresen, editors, *Evolvable Systems: From Biology to Hardware, Fifth International Conference, ICES 2003*, volume 2606 of *LNCS*, pages 93–104, Trondheim, Norway, 17-20 Mar. 2003. Springer-Verlag.

[7] T. Sripramong and C. Toumazou. The invention of cmos amplifiers using genetic programming and current-flow analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(11):1237–1252, November 2002.

[8] A. Stoica, D. Keymeulen, R. S. Zebulum, A. Thakoor, T. Daud, G. Klimeck, Y. Jin, R. Tawel, and V. Duong. Evolution of analog circuits on field programmable transistor arrays. In *Proc. of the Second NASA/DOD Workshop on Evolvable Hardware*, pages 99–108, Palo Alto, CA, USA, July 2000. IEEE Computer Society Press.

[9] T. Quarles, A.R.Newton, D.O.Pederson, A.Sangiovanni-Vincentelli. *SPICE3 Version 3f3 User s Manual*. Department of Electrical Engineering and Computer Sciences, University of California Berkeley, Ca., 94720, May 1993.

[10] G. Tufte and P. C. Haddow. Building knowledge into developmental rules for circuit design. In A. M. Tyrrell, P. C. Haddow, and J. Torresen, editors, *Evolvable Systems: From Biology to Hardware, Fifth International Conference, ICES 2003*, volume 2606 of *LNCS*, pages 69–80, Trondheim, Norway, 17-20 Mar. 2003. Springer-Verlag.