Evolutionary Circuit Partitioning for Field Programmable Analog Arrays

Andreas Rummler

Technical University of Ilmenau, Department of Electronic Circuits and Systems,
PO Box 10 05 65, 98694 Ilmenau, Germany
andreas.rummler@inf-technik.tu-ilmenau.de
http://www.inf-technik.tu-ilmenau.de/~rummler

Abstract. This paper introduces the possibility of solving the circuit partitioning problem for Field Programmable Analog Arrays (FPAAs) by use of a hybrid multi-objective evolutionary algorithm. The paper starts with a short overview over the general architecture of FPAAs and a description of the appropriate partitioning task. The problem is formulated mathematically and a short review of conventional approaches is given. In the next section the used multi-objective evolutionary algorithm SPEA is described. In section 4 the local improvement operator that has been used to support the evolutionary algorithm is discussed in detail. In the last part experimental results are presented and the currently existing deficiencies of the used technique are analyzed. The article concludes with a short outlook on future work.

1 Introduction

Partitioning is a crucial task in the process of automating the design process of VLSI circuits. The task can be explained in more detail as the decomposition of a complex electronic system into a set of smaller subsystems. These subsystems can be designed, simulated and verified for correctness independently and simultaneously. The requirements of the partitioning stage of a system design process are the following: the original functionality of the system must remain intact, the interface interconnections between the subsystems should be minimized and the time required for the decomposition should only be a small part of the total design time [1].

Circuit partitioning in the VLSI design automation process is not just a single task. Dependent on the nature of the system to be designed and the concrete aim of the partitioning task there are different levels: system level, board level and chip level partitioning. The task of decomposing circuits for mapping them to FPAAs can be assigned to the latter level. The partitioning stage is only one stage of the mapping process of circuits. This process is explained very shortly in the next paragraphs.

An FPAA is an integrated programmable device similiar to Field Programmable Gate Arrays (FPGAs). FPGAs are able to reproduce the behaviour of digital circuits when programmed appropriately. FPAAs are able to do the same for

analog circuits. The development of such devices continues only slowly because they are products for several applications in small niches. FPAAs consist of several similiar blocks containing several basic elements: resistors, capacitors and P- and N-channel transistors, which are used to form analog circuits (amplifiers, filters and so on). These blocks are connected to each other by so called routing segments. The segments are interrupted by switch boxes that can be programmed to create connections between all the blocks. The physical design cycle for such devices consists of the following three steps:

1. Partitioning: The circuit to be mapped to the FPAA must be decomposed into several smaller pieces such that every piece fits into one of the blocks of the FPAA. Clearly there is the constraint that no partition may contain more elements of a particular type than available in a FPAA block. In addition there are contraints on the input and output terminals of each partition.
2. Placement: The subcircuits that have been created in the partitioning stage must be assigned to exactly one block on the FPAA. This must be performed in such a way that the third stage can be completed.
3. Routing: In this stage all used blocks on the FPAA are connected via the routing segments by programming the switch boxes. It is possible that this stage cannot be completed due to the limited routing resources (number of routing segments and switch boxes on the FPAA). In such a case the placement stage must be repeated.

In this article only the application of evolutionary methods for the partitioning stage is discussed, the other stages are subject to future work.

Over the years many algorithms for solving the circuit partitioning problem have been developed. Depending on the basic ideas of these algorithms several different classifications can be undertaken. One is the differentiation between *constructive* and *iterative* algorithms.

Constructive algorithms start with empty partitions and fill these partitions with elements during a run. An example are the algorithms presented by Kodres in [2] which start from one (or more) seed elements to form partitions. The solution quality of such a clustering algorithm is in most cases highly dependent on the initial choice of the seed element.

Iterative techniques start with an initial random solution and try to improve this solution in one or more sequential runs. The first algorithm of this kind has been introduced by Kernighan and Lin in [3] and is specialized in bipartitioning. This algorithm tries to improve an initial solution by swapping the partition assignment of two elements. The algorithm has been improved later by Fiduccia and Mattheyses in [4]. In this paper the notion of the *gain* of moving an element to another partition has been introduced. The gain has been defined as the improvement in cutsize that can be achieved by executing a move. By use of specialized data structures (bucket arrays) a single iteration of the algorithm has the complexity of $O(n)$ instead of $O(n \, log \, n)$ in the KL-algorithm. Krishnamurthy enhanced the FM-algorithm in [5] with a look-ahead strategy using *level gains*. Sanchis generalized these approaches in [6] to k-way partitioning.

An alternative classification can be made based on the computation scheme of the algorithm: deterministic and stochastic algorithms. The techniques mentioned in the last section belong to this class (although some random decisions may be made during a run). Examples for stochastic techniques are algorithms based on *simulated annealing* and, of course, evolutionary techniques. Stochastic algorithms are able to find near-optimal solutions and are quite insusceptible against being trapped in local optima unlike deterministic algorithms. The major drawback of such techniques is the amount of the necessary computational effort that is significantly higher than that of heuristic algorithms. For that reason they only became applicable by the rapid increase of computing power.

2 Circuit Partitioning for FPAAs

There are several different mathematical models to describe the (electrical) structure of a circuit. In most cases graphs are used for modeling. Depending on the required abstraction level several different graphs can be used to describe a circuit. Common are the simple clique model, bipartite, tripartite and hypergraphs. The deficiences of the clique model have been shown by Schweikert and Kernighan in [7]. Tripartite graphs have the highest abstraction level and are only used in special cases. The level of abstraction in the bipartite and the hypergraph model is the same, with the difference that the hypergraph model is easier to implement and to handle. For this reason the author concentrates on the hypergraph model.

The problem of partitioning a hypergraph can be formulated as follows: Let $H = (V, E)$ be a hypergraph with a non-empty set of vertices V and a set of hyperedges E. Each vertex $v_i \in V$ is assigned to one of the subsets (partitions) $V_1 \ldots V_k$ such that $V_i \bigcap V_j = \oslash$ for $i \neq j$. As a consequence a set of hyperedges $E_C \in E$ exists that contains all hyperedges that have at least two incident vertices that belong to different vertex subsets. This set of hyperedges is refered to as the *cutset.* All hyperedges contained in the cutset suffice the following condition: $\{e_i \leftrightarrow v_j, e_i \leftrightarrow v_k \wedge v_j \in V_m, v_k \in V_n \wedge V_m \cap V_n = \emptyset\}$. The *cutsize* $c(H) = |E_C|$ equals the number of hyperedges belonging to the cutset. The aim of partitioning the hypergraph is to find an assignment of all vertices such that the cutsize becomes minimal. Depending on whether the number of vertex subsets k is known or not, this task is refered to as partitioning of a hypergraph resp. clustering of a hypergraph.

The mapping of an electrical circuit to its appropriate hypergraph is easy: all netlist elements are transformed into vertices and all nets connecting these elements are transformed into hyperedges. Figure 1 shows an analog amplifier, figure 2 its appropriate hypergraph model. The pins for the voltage sources have been omitted in the model but (if necessary) can be transformed into vertices too. The hyperedges are depicted as straight lines or as black circles/lines.

An analog circuit normally contains several different netlist elements: transistors, resistors, capacitors etc. For this reason every vertex v_i a type attribute t_i is

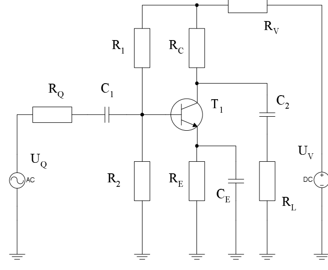

Fig. 1. example of an analog electronic circuit (amplifier)

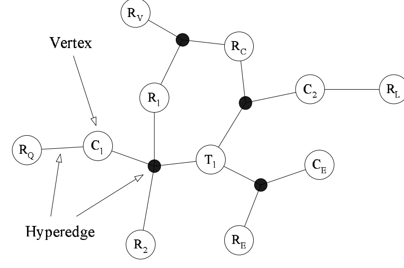

Fig. 2. hypergraph model of the amplifier from figure 1

assigned. The value of t_i is one character from an alphabet of size $0 \leq t_i \leq T-1$ for a circuit containing T different elements.

Now for the partitioning task the optimization criteria and constraints can be defined. The first and most important criteria is the minimization of the already defined cutsize. In other words the purpose is to minimize the number of hyperedges $|E_C|$ that have incident vertices belonging to different partitions. The incidence of a hyperedge e_i with a vertex v_j is denoted by $e_i \leftrightarrow v_j$. This criterion can be written as:

$$c(H) = |e_i \in E_C| \quad \Rightarrow \text{minimal} \tag{1}$$

A stochastic algorithm trying to minimize only the cutsize would produce a particular solution: all vertices assigned to the same partition. This is a very good solution in terms of the cutsize (which is zero), but of course a useless one. That is why most algorithms include a heuristic to preserve or restore the balance of the solution. Balancing a solution means that all partitions have the same size resp. contain the same number of vertices. It is important that the partitions *should* have same sizes but need not. Unlike in conventional approaches where the balance is often treated as a constraint it is possible to turn the balance into a second criterion. This criterion can be formulated as follows:

$$b(H) = \sum_{i=1}^{k} \left| \frac{n_{opt} - |V_i|}{n_{opt}} \right| \quad \text{with} \quad n_{opt} = \frac{|H|}{k} \quad \Rightarrow \text{minimal} \tag{2}$$

In addition to criterion 2 the number of pins of all partitions should be minimized. This criterion is in a sense similiar to the last criterion defined. The formulation is as follows:

$$p(H) = \sum_{i=1}^{|E|} deg(e_i) \,\forall\, e_i \in E_C \tag{3}$$

The reason for introducing this criterion is that every physical cell on the programmable array has only limited number of input/output pins. By minimizing the number used I/O pins the execution of the routing stage becomes easier.

As already stated there is a hard constraint which must be adhered. The total number of circuit elements of type t_i is $|t_i|$. If the circuit is partitioned into k subcircuits every element is assigned a partition j with $0 < j < k$. Now every partition j contains a number of elements of a specific type $|t_i(V_j)|$. Due to the fact that only a limited number of elements is physically available this number must be smaller than a particular maximum value: $|t_i(V_j)| < |t_{i,max}(V_j)|$.

3 Multiobjective Evolutionary Algorithms

The problem formulation above indicates that for solving this problem a multi-objective EA could be used. The author of this article has chosen the *Strength Pareto Evolutionary Algorithm (SPEA)* that was introduced in 1999 by Zitzler. It belongs to the class of elitist evolutionary algorithms and was chosen for this work because of its good performance compared to other similiar algorithms [8]. The algorithm is described in detail in [9], therefore only a short overview will be given here.

SPEA incorporates two populations instead of one: a normal population P and a so-called archive A. The archive stores non-dominated solutions, that have been found by the algorithm during a single evolution cycle. In every generation the individuals of the current population are compared to the ones contained in the archive and the non-dominated individuals are saved in the archive. The individuals from the archive also take part in all genetic operations to steer the algorithm towards areas in the search space with good solutions. An overview of the algorithm is shown in figure 3.

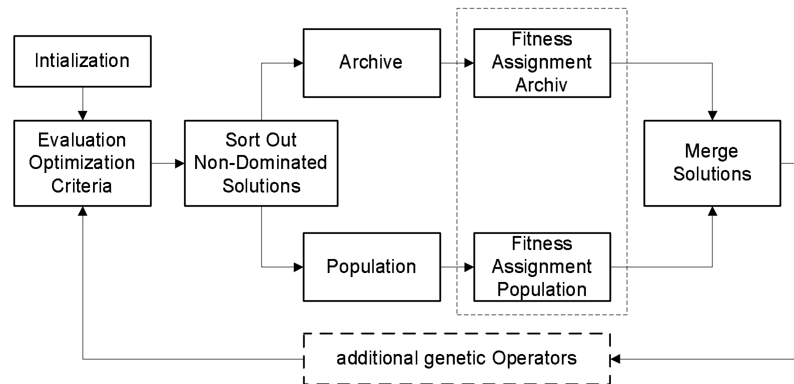

Fig. 3. schematic overview over SPEA

SPEA starts with an initial number of N_p randomly created individuals and an empty archive of maximum size N_a. For all individuals the objective function(s) are evaluated and the non-dominated solutions are copied into the

archive. Solutions that are already contained in the archive but are now dominated by newer individuals are deleted. After this step the archive contains the non-dominated individuals of both population and old archive contents. During the algorithm's runtime there is the danger of an overcrowded archive. SPEA solves this problem by applying a clustering algorithm if the number of solutions contained in the archive is greater than its maximum size. The clustering algorithm reduces the solutions that represent the pareto front. For a detailed description of the clustering algorithm one may refer to [9].

The fitness assignment is divided into two parts. First is the assignment of the fitness values to the individuals contained in the archive. This value is called *strength* S and is defined as:

$$S_i = \frac{n_i}{N+1} \tag{4}$$

The value of n_i represents the number of solutions in the current population, that are dominated by solution i. N is the total number of individuals in the current population. It applies that $S_i < 1$.

In the second step each individual j in the population is assigned a fitness value F in a similiar way:

$$F_j = 1 + \sum_{i \in A \wedge i \succ j} S_i \tag{5}$$

The fitness F_j is equal to the sum of all strength values of the individuals in the archive that dominate the solution j. It applies that $F_j > 1$. With a comparison of both inequations it can be seen that individuals contained in the population can never have a better fitness than the ones in the archive.[1]

After assigning the fitness values the solutions of both the archive and the population are merged and processed with suitable genetic operators that are problem-specific.

To form a valid evolutionary algorithm some operators are still missing: selection, recombination and mutation. The appropriate operators that have been used in this work are explained in more detail below.

The only operator of the missing ones that must not be tailored to the genetic representation is the selection operator. Here the tournament selection has been chosen (as recommended by Zitzler), although other selection techniques could be used here. In contrast the operators for recombination and mutation are always problem-specific. They must be tailored to the genetic representation that has been chosen to represent a potential solution of the particular optimization problem.

For the hypergraph partitioning problem a vector-like genetic representation presents itself. Each vertex can be assigned an integer number indicating the index of the partition the vertex belongs to. To indicate the different types of the vertices a number of vectors of integer numbers has been used. Each vector contains the partition numbers of all elements of a particular type. For the

[1] Unlike in other evolutionary algorithms in SPEA a smaller fitness value is defined to be the better one.

example from figure 2 the resulting structure of an individual is shown in figure 4. Each individual contains a number of chromosomes that again contain an integer vector with the partition numbers of the elements of the respective type.

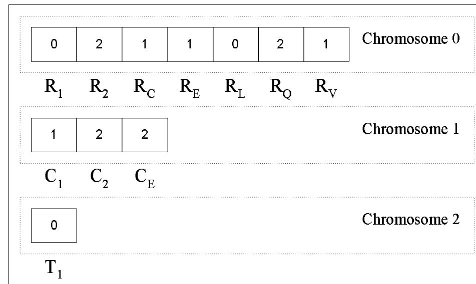

Fig. 4. genetic representation

By choosing a vector-like genetic encoding, the operators for recombination and mutation must be vector-based too. But it is not possible to use standard operators like multi-point-crossover *out of the box* because such operators could produce encodings that violate the constraints described above and therefore invalid individuals. There are two ways out of this situation: special tailored operators could be used that are not able to produce such invalid encodings or an additional repair operator must be applied afterwards. In this case the author made the decision in favour of the second alternative because of the much easier implementation. As recombination a simple multi-point-crossover has been used followed by the repair operator shown in algorithm 1.

Algorithm 1 algorithm for repairing invalid genetic encodings

```
for all element type t do
  for all partitions p do
    identify if p has space for other elements, is full or overcrowded (invalid)
    add p to one of the appropriate sets SPACE, FULL, INVALID
  end for
  for all partitions p in set INVALID do
    while p is invalid
    choose random vertex from p and move it to a random partition in set SPACE
    if the chosen random partition reaches state FULL move it to set FULL
  end for
end for
```

The sense of recombination is the improvement of attributes of the parents in their children. For this task it is necessary to identify these attributes and to use special operations to improve these attributes (directed search). In contrast mutation plays the role of an indirected search operator for exploring former unidentified regions in the search space. A recombination operator only works in a desireable manner if it is possible to identify the good attributes of the parents and apply a technique to improve them. This seems to be quite difficult in this case – the recombination together with the repair operator brings quite a big random element into the evolutionary algorithm. For this reason an ad-

ditional mutation operator has not been applied – the random element in the recombination operator already plays that part.

4 Local Improvement

For support of the multiobjective evolutionary algorithm described above a local improver has been developed. Due to the fact that the improver must be applied to a number of individuals, it must have a small runtime. Because of this reason the multi-way partitioning algorithm developed by Sanchis [6] has been used and modified for use as a local improver.

The local improvement algorithm that has been used here is a move-based algorithm. A *move* is the change of the assignment of an arbitrary vertex to another partition. The basic idea is to analyze the possible moves of all vertices for an improvement in cutsize. Only moves which improve the cutsize are executed – a typical hill-climbing strategy.

The original algorithm by Sanchis does not treat any constraints, so a constraint handling technique had to be introduced. In Sanchis' algorithm every move is valid. Due to the contraint that the number of elements of the same type in a partition is limited in this case all moves can be assigned one of two groups: moves that can be executed immediately (*valid moves*) without breaking the constraint and moves that will create a partition that violates the constraint (*invalid moves*). An example for valid and invalid moves is shown in figure 5.

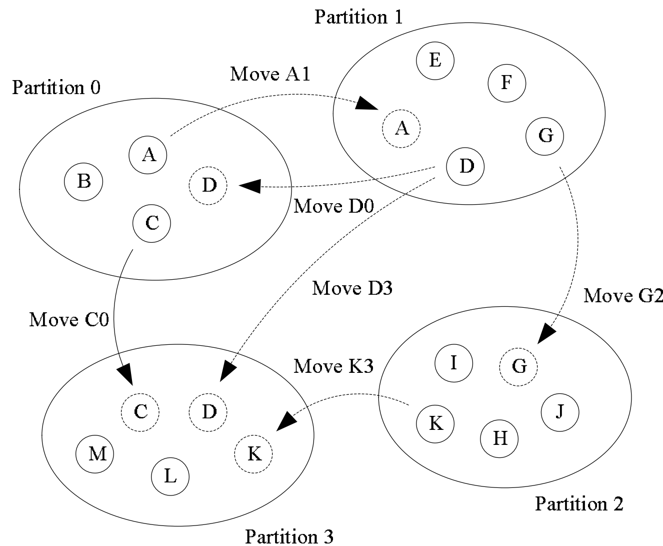

Fig. 5. valid and invalid moves of vertices between partitions

In the example in figure 5 it is assumed that every partition can contain up to four vertices. It is easy to see that moving vertex C to partition 3 (move $C3$) does not violate this constraint. The opposite is the invalid move $A1$, that invalidates partition number 1 which would contain 5 vertices. Now there are two things which can be done in this situation: forbid such invalid moves in general or move one vertex away from this partition. The first proposition does not seem to be a real alternative because in a situation where every partition contains the maximal number of elements no move could be executed. So the second variant must be applied. To revalidate the situation that was caused by move $A1$ for example the vertex D could be moved to partition 3. This leads to the definition of *move chains*. A move chain is a chain of moves where the execution of each move depends on the execution of a subsequent move to create a valid configuration. In the example several move chains are shown: $\{A1, D0\}$, $\{A1, D3\}$ and $\{A1, G2, K3\}$. The first chain is a special case where two vertices are swapped between two partitions.

In general the move chains can be of arbitrary length. In this work the length has been limited a value of 2, this means only exchanges of vertices can be performed. A case like move chain $\{A1, D3\}$ has not been incorporated because moving D *may* invalidate the configuration.

The local improvement algorithm that takes care of the constraint is shown in algorithm 2.

Algorithm 2 local improvement algorithm

```
for all individual i in collection do
  get chromosome set from i and build integer vector
  call initialization()
  call setupMoveChainSet()
  while move chain set contains more chains do
    get next move chain from chain set
    if move chain will improve the individual then
      call performMoveChain()
    else
      break while-loop
    end if
  end while
  create new chromosome set from improved configuration and assign it to i
end for
```

Every vertex v is assigned a gain vector Γ_v of the size equal to the number of partitions. Each element $\gamma_j(v)$ with $0 < j < |\Gamma_v|$ in the vector Γ_v represents the gain that is achieved by moving the vertex to partition j. The gain is defined as the improvement (the decrease) in cutsize that would be achieved if that move was executed. The gain can be both positive and negative.

In most cases a number of possible moves have the same gain, so it is difficult to find the move which is to be executed. For that reason each vertex was assigned

a second gain vector Θ_v with the same size as Γ_v. The gain $\theta_j(v)$ with $0 < j < |\Theta_v|$ represents the incidence gain that is achieved by moving the vertex to partition j. The incidence gain is defined as the improvement (the decrease) in the incidence degree of a vertex that would be achieved if that move was executed. The incidence degree is defined as the number of different partitions the neighbours of a vertex belong to. Hyperedges incident to a vertex can only be removed from the cutset if the incidence degree is equal to 1 (the vertex has only neighbours in the same partition), therefore the incidence degree must be lowered if possible.

Similiary two vectors B_e and B'_e are assigned to each hyperedge e, both with the size of the number of partitions. Each element $\beta_j(e)$ in vector B_e represents the number of neighbouring vertices belonging to partition j. The elements in vector B'_e are defined as the sum of the $\beta_j(e)$ values that do not belong to the same partition.

The values must be initialized first. This procedure is shown in algorithm 3. All values associated to vertices and hyperedges are set to 0. Afterwards the values are initialized to the correct values that are valid at the time before the improvement procedure starts.

Algorithm 3 procedure *initialization*

```
define p(v) as current partition the vertex v belongs to
for all hyperedge e in hypergraph do
  reset values for β_j and β'_j in e
end for
for all vertex v in hypergraph do
  reset values for γ_j and θ_j in v
end for
for all vertex v in hypergraph do
  assign partition number from chromosome set to v
  for all neighbouring hyperedge e in v do
    increment β_p(v)(e)
  end for
end for
```

After the initialization stage all possible move chains must be calculated. This procedure is shown in algorithm 4. The calculation is done in an iteration over all vertices. Appropriate moves are created for moving a vertex to another partitition. Dependent on the decision if a move is valid or invalid a move chain with only one or with two moves is created. Each move chain the values of the yielded gains are associated. This is quite easy in the case when the chain only contains one move: the values of $\gamma_j(v)$ and $\theta_j(v)$ are copied. For a chain containing two moves this task is a little more complicated. The first move must be executed and the gains of the neighbouring vertices must be updated. After that the gain of the second move can be calculated. The overall gain of the move chain is the sum of the gain values. Afterwards the gain values of the involved

vertices must be restored to their original values (this is not shown in algorithm 4).

Algorithm 4 procedure *setupMoveChainSet*

```
define p_c(v) as current partition the vertex v belongs to
define p_t(v) as target partition that the vertex v should be moved to
for all vertex v in hypergraph do
  for all p_t(v) in possible partitions do
    if p_c(v) ≠ p_t(v) then
      if v can be moved to p_t then
        create move chain {v ⇒ p_t}
      else
        for all vertices v_t in p_t with type equal to type of v do
          create move chain {v ⇒ p_t, v_t ⇒ p_c}
        end for
      end if
    end if
  end for
end for
```

The move chains are held in a special data structure (tree map) which guarantees storage and removal operations to be executed in $nlogn$ time. Furthermore it guarantees that its contents are sorted at every time.

The main part of the local improvement operator is shown in algorithm 5. This procedure takes a move chain as an argument and executes it. Executing a move will invalidate the values of $\gamma_j(v)$ and $\theta_j(v)$ values of the moved vertex and its neighbouring vertices as well as the $\beta_j(e)$ and $\beta'_j(e)$ of the neighbouring hyperedges. Therefore the main function of algorithm 5 is the continuous update of these values.

5 Experimental Results & Analysis

The performance of the partitioning algorithm has been tested on several hypergraphs from the benchmark suite available from [10]. Due to the fact that these graphs do not contain any type information for the vertices random type information has been generated and assigned to each vertex.

The diagrams in figures 6 and 7 show the progress of the values for the cutsize and the pincount for the hypergraph *fract* over 70 generations. The *fract* netlist contains 149 devices (vertices), 147 nets (hyperedges) and 462 pins (sum of the degree of all hyperedges). For the simulations each vertex has been assigned one type out of 4/6/8 possible types. This has been done randomly. The figures illustrate the case, where each partition is allowed to host up to four vertices of each type. With these data the minimal number of required partitions can be calculated easily.

Algorithm 5 procedure *performMove*

define p_s **as** number of the source partition of the move
define p_d **as** number of the destination partition of the move
assign new partition number to vertex v
for all hyperedge e that is incident to v **do**
  increment $\beta_{p_d}(e)$, decrement $\beta_{p_s}(e)$
  decrement $\beta'_{p_d}(e)$, increment $\beta'_{p_s}(e)$
  for all vertices v_n incident to e **do**
    for $i = 0$ to number of partitions **do**
      if $P_s \neq p_d$ **then**
        if $\beta_{p_d}(e) > 0$ **then**
          if $\beta'_{p_d}(e) = 1$ **then**
            increment $\gamma_{p_d}(v_n)$
          end if
          if $\beta_{p_s}(e) = 1$ **then**
            increment $\theta_{p_d}(v_n)$
          end if
        end if
      else
        if $\beta'_{p_d}(e) = 0$ **then**
          decrement $\gamma_{p_d}(v_n)$
        else
          if $\beta_{p_s}(e) \neq 1$ **then**
            decrement $\gamma_{p_d}(v_n)$
          end if
        end if
        if $\beta_{p_s}(e) \neq 1$ **then**
          decrement $\theta_{p_d}(v_n)$
        end if
      end if
    end for
  end for
end for

The diagrams in figures 8 and 9 originate from experiments with the hypergraph *balu*. This netlist contains 801 vertices, 735 nets and 2697 pins. Again, each partition has been allowed to host up to four vertices of each type.

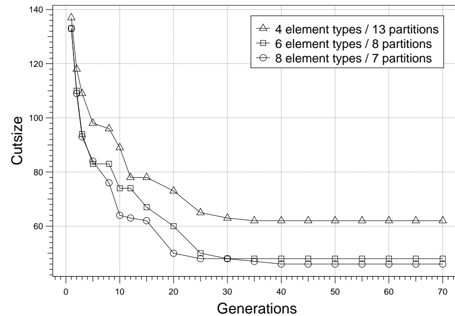

Fig. 6. Progress of the cutsize in graph *fract*

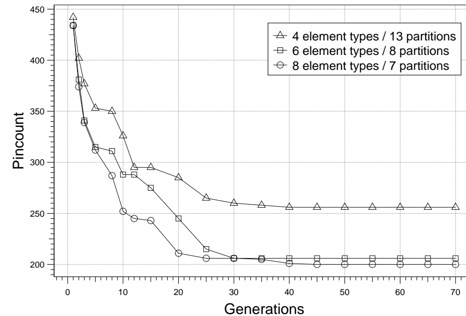

Fig. 7. Progress of the pincount in graph *fract*

In each diagram the x-axis corresponds to the number of generation and the y-axis to the cutsize and the pincount respectively. To collect the values for the curves the pareto front in each generation has been searched for the individuals containing the best values. The evolutionary algorithm has been set up with the following parameters: size of the population = 40, maximal size of the archive = 12.

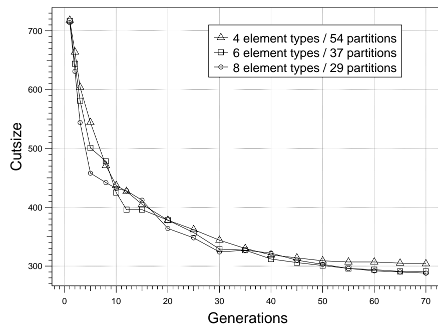

Fig. 8. Progress of the cutsize in graph *balu*

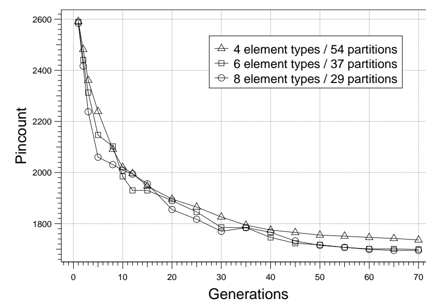

Fig. 9. Progress of the pincount in graph *balu*

By examining the result curves it can be presumed that the proposed technique works. Unfortunately there a no comparative results available. For instance the cutsize in the fract graph drops from an initial value of around 140 to 60 after convergence, but it is unknown if the optimal cutsize value is at 50 or maybe

20. In addition there are quite a lot of free parameters that can be tuned and perhaps improve the performance: In what way does the size of the population or the size of the archive affect the performance? Is it better to improve the individuals in the archive by the hillclimbing algorithm to speed up the whole EA? Are mutation operators that dynamically create and destroy partitions useful? Clearly the answering of such questions has to be done in future work.

When examining the used genetic representation carefully, it seems to a weak point. In the vector representation that was used here there are p^v possible "individuals" with v being the number of vertices in the hypergraph and p the number of partitions. A closer look reveals that there are a number of variants for the representation vector possible, that are redundant. For better explanation imagine a very simple hypergraph with three vertices A, B and C that should be splitted in two pieces. For this case there are $p^v = 2^3 = 8$ possible variants. But by counting all possiblilities by hand there are less: namely the possibilities $\{ABC\}\{\}$, $\{A\}\{BC\}$, $\{AB\}\{C\}$ and $\{AC\}\{B\}$. This is because there are variants that describe the same case. The two representation variants $\{000\}$ and $\{111\}$ describe that vertices A, B and C are assigned to partition 0 resp. to partition 1 – but the real proposition is another: all vertices are assigned to the same partition. Therefore one of both variants is redundant. Unfortunately 'relative' propositions like *"vertices A and B are assigned to another partition than vertex C"* cannot be expressed in this form of genetic representation.

Another effect that has been observed is the contraction of the pareto front over the runtime of the algorithm towards a single point. This means the diversity of points in the search space that is used for the search decreases with progressing algorithm. With this decrease the chance of exploring new areas in search space is naturally reduced and the algorithm converges too early. This behaviour may be fixed with the use of Zitzlers SPEA2 algorithm.

6 Conclusion

In this article an approach for partitioning analog circuits for mapping them to Field Programmable Analog Arrays based on multi-objective evolutionary optimization coupled with local improvement support has been presented. The algorithm has been implemented using the evolutionary algorithm toolkit *eaLib* [11], [12] and the correct functioning of the proposed technique has been verified.

A weak point in the algorithm is the genetic representation that has been chosen. This is now subject of further investigations. Eliminating the redundant variants by choosing an alternative representation may be hard. But perhaps it is possible find a technique to identify the redundant variants and map them to a region in the search space that only contains non-redundant configurations.

Another issue of future work is the merging of the local improver into a recombination operator that is able to identify good attributes in parent chromosomes and improve these attributes in child chromosomes.

The applicability of hybrid evolutionary techniques in the area of electronic circuit partitioning has been shown. For that reason future work will also concen-

trate on the application of this technique in the areas of placement and routing (as already implied).

References

1. Sherwani, N.: Algorithms for VLSI Physical Design Automation. third edn. Kluwer Academic Publishers, Boston, Dordrecht, London (1998)
2. Kodres, U.R.: Partitioning and card selection. In Breuer, M.A., ed.: Design Automation of Digital Systems: Theory and Techniques. Volume 1. Prentice Hall (1972) 172–212
3. Kernighan, B.W., Lin, S.: An efficient heuristic procedure for partitioning graphs. The Bell System Technical Journal **29** (1970) 291–307
4. Fiduccia, C.M., Mattheyses, R.M.: A linear-time heuristic for improving network partitions. In: Proceedings of the 19th IEEE Design Automation Conference. (1982) 175–181
5. Krishnamurthy, B.: An improved min-cut algorithm for partitioning VLSI networks. IEEE Transactions on Computers **33** (1984) 438–446
6. Sanchis, L.A.: Multi-way network partitioning. IEEE Transactions on Computers **38** (1989) 1384–1397
7. Schweikert, D., Kernighan, B.: A proper model for the partitioning of electrical circuits. In: Proceedings of the Ninth Design Automation Workshop on Design Automation. (1972) 57–62
8. Deb, K.: Multi-Objective Optimization using Evolutionary Algorithms. John Wiley & Sons, Chichester (2001)
9. Zitzler, E.: Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications. PhD thesis, Eidgenössische Technische Hochschule Zürich (1999)
10. : The circuit partitioning page. (`http://vlsicad.cs.ucla.edu/~cheese/benchmarks.html`) URL time: January 20th, 2002, 10^{00}.
11. Rummler, A., Scarbata, G.: eaLib - a java framework for implementation of evolutionary algorithms. In: Computational Intelligence: Theory and Applications. Volume 2206 of Lecture Notes in Computer Science., Springer-Verlag (2001) 92–103
12. : ealib. (`http://www.inf-technik.tu-ilmenau.de/~rummler/eng/ealib.html`) URL time: August 28th, 2001, 14^{30}.