Iterative Improvement Based Multi-Way Netlist Partitioning for FPGAs

Helena Krupnova, Gabriele Saucier
Institut National Polytechnique de Grenoble, CSI
46, Avenue Felix Viallet, 38031 Grenoble cedex, FRANCE
{bogushev,saucier}@imag.fr

Abstract

This paper presents a multi-way FPGA partitioning method. The basic idea is similar to one proposed in [12], but instead of using the replication and re-optimization, it takes force of the classical iterative improvement partitioning techniques ([4],[14]). The basic effort consists in guiding the classical algorithms in their solution space exploration. This was done by introducing the cost function based on the infeasibility distance of the partitioning solution and carefully tuning the basic parameters of classical algorithms such as definition of size constraints for feasible moves, handling solutions stack, selecting best cluster to move, etc. The proposed method obtains results comparable to the best published results ([12],[16]), and even outperforms them for biggest benchmarks.

1. Introduction

Partitioning problem was extensively addressed in the previous research. Te best known and most cited is the iterative improvement partitioning algorithm of Fiduccia and Mattheyses (FM) [4]. Two basic extensions of this algorithm are the algorithm of Krishnamurthy [8] which improves the tie-breaking in the FM algorithm by introducing the higher-level gains and look-ahead technique for the cell moves, and the Sanchis' algorithm [14], which is the extension of two previously mentioned bipartitioning algorithms to the multi-way partitioning.

The behavior of the FM based partitioning technique was largely studied in the past. Recently published works [5], [7], [2], [17] discuss the impact of different optimization parameters on the behaviour of the FM algorithm. These parameters are generally the method of finding the initial partition, selecting the cell to be moved (look-ahead heuristics), clustering approaches, organization of the data structures (LIFO, FIFO gain buckets), cell locking strategies, number of runs, number of passes, etc.

The mostly referenced works for multi-FPGA partitioning are [9], [10], [11], and [12]. Proposed in [9] method (k-way.x) is based on recursive bipartitioning alternating with improvements by calling the FM algorithm ([4]). In [11], the previous method was extended with functional replication possibilities (implementation program called r + p.0). Finally, in [12] is introduced a recursive paradigm (PROP) combining partitioning, replication and optimization. Another known approaches are the network-flow based method FBB-MW of [16], the "local ratio-cut" and set covering approach (SC, LRSC) of [3], the WINDOW ordering, clustering and dynamic programming-based method WCDP of [6].

In this paper, we present a multi-way FPGA partitioning method which is similar to one proposed in [12]. But instead of using the replication and re-optimization, it takes force of the classical partitioning techniques of [4],[14]. We show that the FM based techniques being adopted to the FPGA partitioning allow to obtain results comparable to the best published results ([12],[16]). The basic effort consists in guiding the FM and Sanchis' algorithm in their solution space exploration. This was done by playing with classical parameters of the FM algorithm, such as number of passes, mechanism of selecting cells to move, data structures, etc. In addition, an important role is given to the general organization of the algorithm, adopted infeasibility-based cost functions for selecting the best solution, and the strategy of applying the appropriate size constraints to allow/disable the cell moves aiming at concentrating the search process in the region of the solution space with greatest probability to find the feasible solution.

This paper is organized as follows. The next section presents the FPGA partitioning problem formulation. Then, section 3 describes the proposed method starting from the overall organization of the partitioning algorithm. After that, implementation details, such as creation of initial partition, cost function for selecting the best solution, size constraints on cell moves, etc. are explained. Finally, experimental results are presented and some conclusions are given.

2. Problem definition

The FPGA partitioning problem is defined as implementing a given digital circuit as a set of subcircuits each of which can be implemented as a single FPGA device. A digital circuit is represented as a hypergraph $H_0 = (\{X_0, Y_0\}, E_0)$, where X_0 and Y_0 denote respectively the interior and terminal node sets, $X_0 \cap Y_0 = \emptyset$, and E_0 is the set of nets. Each interior node x_i may be weighted by a size parameter $S(x_i)$ representing the number of target technology cells needed to implement the given node. The size of the circuit is the sum of sizes of all interior nodes: $S_0 = \sum S(x_i)$. A k-way partition of H_0 implies an assignment of the nodes in X_0 and Y_0 to a set of k non-overlapping hypergraphs $P_j = (\{X_j, Y_j\}, E_j)$. During the partitioning process, the terminal nodes of the original hypergraph Y_0 are assigned to terminal node sets Y_j of one or more of the resulting partition's components: $Y_0 \subset \bigcup_{j=1}^{k} Y_j$. Each of the interior nodes of the original hypergraph is assigned to the interior node set of exactly one component hypergraph, $\bigcup_{j=1}^{k} X_j = X_0, \forall_{i \neq j} X_i \bigcap X_j = \emptyset$, thus partitioning X_0 into $\{X_1; X_2; ...; X_k\}$.

Each FPGA device D_i is characterized with parameters $D_i = (S_{MAX}, T_{MAX})$, where S_{MAX} represents the size in number of basic cells of the corresponding FPGA technology, and T_{MAX} represents the number of terminals. Additional constraints to consider may be another FPGA resources such as the number of flip-flops, number of tri-state lines (Xilinx target), etc. In our experience, these constraints are rarely critical. They can be handled in a similar way as the size constraint but omitted from the discussion for the reasons of simplicity. The value of S_{MAX} is defined as $S_{MAX} = S_{ds} * \delta$, where S_{ds} is value found in the FPGA vendor data sheet, and δ is a user specified value corresponding to the desired filling ratio. δ is often chosen to be smaller than 1.0 (0.9 for example) to guarantee the successful routing by the vendor place and route tool. We consider that all the subcircuits in the partitioning are implemented with the same device type. There always exists a lower bound M on the number of devices required to implement the given circuit $M = MAX(\lceil \frac{S_0}{S_{MAX}} \rceil, \lceil \frac{|Y_0|}{T_{MAX}} \rceil)$.

It is said that a hypergraph $P_j = (\{X_j, Y_j\}, E_j)$ meets constraints of the device $D_i = (S_{MAX}, T_{MAX}) : P_j \models D_i$ if and only if $\sum_{i=1}^{n_j}(S(x_i)) \leq S_{MAX}$ and $|Y_j| \leq T_{MAX}$, where n_j is the number of interior nodes of the hypergraph P_j.

A k-way partition is called *feasible* if each P_j satisfies the relation $P_j \models D_i$ for some $j = 1, 2, ..., k$. A k-way partition is called *semi-feasible* if $\forall_{1 \leq j \leq k-1} P_j \models D_i$, and the k-th subset P_k does not meet constraints. The subset which does not meet constraints in a semi-feasible partition is called the *remainder* and denoted as R_k. Finally, a k-way partition where more than one subsets don't respect the device constraints is called *infeasible*.

We define the problem of k-way FPGA circuit partitioning as follows: *find a feasible partition with minimum k number*.

3. Method description

We start by mentioning the recursive paradigm used in [9]. The algorithm is composed of $(k-1)$ iterations. The first iteration applies a bipartitioning procedure to the given circuit, and the subsequent iterations apply the same procedure to the remainder. Each iteration produces a bipartition with at least one subcircuit that meets constraints and the remainder which probably does not meet the constraints. Then, an optimization is applied to force the remainder meet the constraints. The iterations stop when the remainder subcircuit meets constraints.

The weakness of the above algorithm is its greedy character. At the later steps there is no possibility to modify blocks created at the previous iterations. In addition, more times the remainder is cut, more new I/O pins appear. At the early iterations, produced blocks have good (100%) filling and small number of I/Os. At the following iterations, 100% filling becomes impossible because I/Os are saturated more quickly than the logic resources.

To overcome the greedy tendency, the replication and re-optimization techniques were proposed in [12]. Each time the remainder was cut, logic was replicated to reduce the number of cut nets, and then optimization was applied to eliminate the size increase due the replication. The re-optimization technique depends on the concrete software flow and the tight integration with the vendor tools. The functional replication possibility depends on whether such functional information is available in the used input format. In this paper we describe a method which does not use the replication and reoptimization and improves the partitioning results by intensively applying classical partitioning strategies.

The optimization objective of the classical bipartitioning problem is to reduce the cutset size of the partition. In multi-FPGA partitioning the basic objective is reducing the number k of FPGA blocks. The basic obstacle for producing the optimal number of blocks is the I/O pin constraint: when nets are cut, the number of I/Os increases. So, reducing the number of cut nets is the way to improve the quality of partitions and to minimize the number of blocks. But these two objectives are not directly related. This was stated also in [9]. In some cases, it may happen that net with zero-gain changes the number of I/Os of block to/from which it is moved. This requires to bring closer together the optimization objective, which is minimization of the number of blocks in a partition, and the optimization mechanism, which is the reducing of the number of cut nets in a cut set.

The way to reduce the number of blocks in final partition using the described above recursive paradigm is to reduce the size and I/O number of the remainder at each step. To do this, we introduced specified cost functions and adopted an appropriate solution space exploration strategy, which will be described in the following sections.

3.1. Overall algorithm organization

In [9], the iterative improvement was called between the remainder and the block produced at the last step. The possibility to optimize the remainder by exchanging cells with blocks produced at the first $(k-1)$ steps was not exploited. As was noted in [9], the ratio $|Y_k^R|/|X_k^R|$ tends to increase during the recursive bipartitioning process. Initially, few signals are cut, and devices produced at first iterations have non saturated I/O pins. Being involved into the optimization process at later stages, these blocks may provide an additional opportunity to reduce the terminals number of the remainder. From the other side, some devices produced at the previous iterations may have free space. Involving them in partitioning at the later stages may allow to move cells from the remainder to these blocks, thus reducing the remainder size.

By these reasons, at each iteration we introduced the improvement pass involving all the blocks of the partition. We used the Sanchis multi-way partitioning algorithm [14] which allows cells to move between all the blocks of the partition.

The following improvement strategy was adopted. The iterative improvement is initially called only for two lately partitioned blocks R_{k+1} and P_{k+1}. This because these blocks are most likely to improve the cutset in the beginning. When the optimization possibilities between these blocks are exhausted, improvement pass for all the blocks may sometimes greatly improve the solution.

From the other side, when the number of blocks k increases, the probability to meet a solution where $(k-1)$ of k blocks respect constraints decreases. For problems where the big number of blocks k is expected, we have chosen another strategy. After the bipartitioning of the remainder, the iterative improvement pass is called first for two lately partitioned blocks, as in the previous case. Then, it is called between the remainder block and block with the smallest size P_{MIN_size}. At the next step, the iterative improvement is called between the remainder and minimum I/O block P_{MIN_IO}, and finally, between the remainder and maximum free space block P_{MIN_F}. Free space of the block is estimated on the base of the occupation of both logic cells and I/Os : $F = \sigma_1 * \frac{S_{MAX}-S_i}{S_{MAX}} + \sigma_2 * \frac{T_{MAX}-|Y_i|}{T_{MAX}}$, where σ_1 and σ_2 are two coefficients. During the experimental evaluation $\sigma_1 = \sigma_2 = 0.5$ was set. This strategy allows to increase filling of badly filled on the previous iterations blocks and as a consequence to decrease the size and I/O number of the remainder.

Thus, we separated the partitioning cases in two groups - the first one with small expected number of blocks, which does not exceed the constant $M \leq N_{small}$, and the second one with big expected number of blocks $M > N_{small}$. N_{small} was empirically defined and set to 15. For each group we adopted the best suited improvement strategy. The iterative improvement involving all the blocks is applied only for the first group. The improvement passes between the remainder and selected blocks P_{MIN_size}, P_{MIN_IO}, P_{MIN_F} are applied for both groups. When $k = M$ is reached, an additional FM call is performed for all pairs of blocks R_k-P_i for $i = 1$ to $(k-1)$.

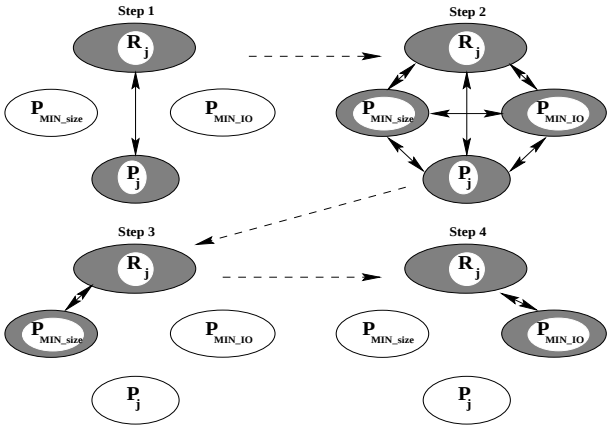

Figure 1. Call of the iterative improvement passes

The described process is illustrated in Figure 1 for partitions with $M \leq N_{small}$. Suppose, that two lately created blocks are $\{P_j, R_j\}$. Blocks involved in the iterative improvement algorithm are shadowed in Figure 1. At step 1, the iterative improvement involves blocks R_j, P_j, at step 2, the iterative improvement is called for all the blocks, then it is called for the remainder block and the smallest size block, and finally at step 4 it is called for the remainder and minimal I/O pin block.

Algorithm 1 represents the general k-way partitioning algorithm we use. It calls two functions : **Bipartition()** and **Improve()**. The first one is used to create the initial partition, and is described in section 3.2. The second one, **Improve()** corresponds to the call of the Sanchis' iterative improvement algorithm and its implementation details are described in sections 3.3-3.7.

Algorithm 1

Input: $H_0(\{X_0, Y_0\}, E_0)$, $D_i(S_{MAX}, T_{MAX})$
Output: k, $P_1, P_2, ..., P_k$
$k = 0$;
$R_k = H_0$;
$M = MAX(\lceil \frac{S_0}{S_{MAX}} \rceil, \lceil \frac{Y_0}{T_{MAX}} \rceil)$;
proceed=TRUE ;
while (proceed) **do**
    $k = k + 1$;
    $\{R_k, P_k\}$ = **Bipartition**(R_{k-1});
    Improve(R_k, P_k);
    if $(M \leq N_{small})$
        Improve$(P_0, P_1, ..., P_k, R_k)$;
    end if
    Improve(P_{MIN_size}, R_k);
    Improve(P_{MIN_IO}, R_k);
    Improve(P_{MIN_F}, R_k);
    if $(k = M)$ and $(M \leq N_{small})$
        for $(i = 1; i \leq (k-1); i++)$
            Improve(P_i, R_k);
        end for
    end if
    if $(R_k \models D_i)$
        proceed=FALSE;
    end if
end while

3.2. Initial partition creation

It was observed that randomly created initial partition may lead to poor results ([9]). As we need a semi-feasible partition as an initial solution, and this is not guaranteed by the random method, a constructive method has been chosen to create the initial partition. Two passes corresponding to two different methods are applied, and the best solution of two passes is selected as an initial solution. The first one is a greedy nodes merge similar to one proposed in [1]. Initially, two seed nodes are selected in a hypergraph. The first node is one with the biggest size, and the second node is one which has the maximal distance from the first node found by breadth-first search. After the seed nodes for two blocks are selected, at each step, one node is add to each block. The merge candidate is chosen on the base of the cost function ([1]): $Cost_{(i+j)} = \frac{S_{(i+j)}}{T_{(i+j)}}$, where $S_{(i+j)}$ and $T_{(i+j)}$ are correspondingly size and I/O pin number if two nodes i and j are merged. Creating two blocks in the same time slightly alleviates the greedy tendency of the algorithm [1]. This tendency manifests by absorbing at the first steps nodes with good cost. Merge for each block stops when constraints are saturated for both blocks and block with biggest size is selected as P_k. The unassigned cells are merged with another block which forms the remainder R_k.

During the second pass, ratio cut objective function [15] is used to obtain an initial solution. The first initial seed point is selected as first block of the partition, and the rest of nodes as second block. Nodes are moved one by one to the first block, and each time the ratio of the partition is estimated : $R_{i,j} = \frac{C_{i,j}}{S(P_i) * S(P_j)}$, where $C_{i,j}$ is the size of the cut set. The same operation is performed starting from the second seed point. Partition with smallest ratio and having at lest one block satisfying constraints is selected at the end. Best solution of these two algorithms ([1],[15]) is retained as an initial partition for the iterative improvement algorithm.

3.3. Infeasibility distance cost function

The classification of partitioning solutions in feasible, semi-feasible and infeasible (section 2) is illustrated graphically in Figure 2. The X-axis corresponds to the number of I/O pins, the Y-axis to the size in number of logic cells. The shadowed feasible region is delimited by the device constraints S_{MAX}, T_{MAX}. Each partition block is represented as a point in a 2-dimensional space, and a set of points represents a partitioning solution. A point inside the rectangle corresponds to the feasible block, and a point outside the rectangle - to the infeasible block. Figure 2a represents a 4-block feasible solution. Figures 2b and 2c represent correspondingly 3-block semi-feasible and 4-block infeasible solutions.

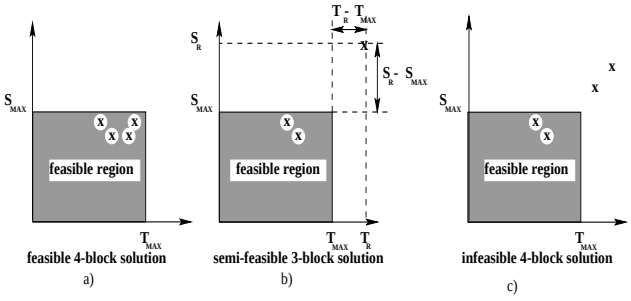

Figure 2. Feasible, semi-feasible, infeasible solutions examples

In the cost function of [9], only the net number was taken into account. Here we introduce the *infeasibility distance*-based cost function. We define the infeasibility distance for partition block P_i as : $d_i = \lambda^S * d_i^S + \lambda^T * d_i^T$, where $d_i^S = \frac{S_i - S_{MAX}}{S_{MAX}}$ is the size infeasibility distance of block P_i if $S_i > S_{MAX}$, and $d_i^S = 0$ otherwise. Similarly for the I/O number: $d_i^T = \frac{T_i - T_{MAX}}{T_{MAX}}$ is the I/O infeasibility distance of block P_i if $T_i > T_{MAX}$, and $d_i^T = 0$ otherwise. λ^S and λ^T are the coefficients which express the importance of the corresponding component in the final cost function and are determined on the experimental basis. The infeasibility distance of a solution is a sum of the distances of all the blocks. Obviously, the infeasibility distance of the feasible solution equals 0.

An example of the infeasibility distance is presented in

Figure 2b for the remainder block. The infeasibility distance expresses how far is the given solution from the desired feasible solution. Comparing two semi-feasible solutions by the infeasibility distance, one which has the remainder block closer to the feasible region is preferred. The smaller is the infeasibility distance, the easier will be to make the remainder feasible. Because the I/O constraint is usually the most critical during the partitioning, coefficient λ^T should be more significant than the coefficient λ^S ($\lambda^S = 0.4$ and $\lambda^T = 0.6$ were used in the experimental evaluation).

When the expected number of blocks is high (lower bound M of the partition is big), $d^S_{R_k}$ will remain relatively big during significant number of iterations. Small variation of $d^S_{R_k}$ will not be sensitive in different solutions. In addition, if the expected number of blocks is high and correspondingly the number of iterations is also high, the difference of the number of I/Os on the remainder T_{R_k} at different iterations may vary very few. Thus, the cost function is not sensitive to the fact that the first produced blocks have good or bad filling. Producing unfilled blocks at numerous iterations will finally impact the resulting number of blocks. To eliminate this drawback, we introduced an additional component $\lambda^R * d^R_k$ in the final cost function : $d_k = \sum_{i=1}^{k}(d_i) + \lambda^R * d^R_k$, where d^R_k is the *size deviation penalty* which is defined as $d^R_k = \frac{S_{AVG}}{S_{MAX}}$ if $S_{AVG} > S_{MAX}$ and $d^R_k = 0$ if $S_{AVG} \leq S_{MAX}$. The S_{AVG} component is determined in a following way. At step k, $k+1$ blocks are already created, and the remainder block should be partitioned at least $M-k$ times. Then $S_{AVG} = \frac{S(R_k)}{M-k+1}$ represents the average size of blocks produced if the remainder will be partitioned in minimal theoretical number of blocks. This component of the cost function will give preference to solutions where the size of the remainder is sufficiently small to fit in minimal theoretical number of devices. It penalizes solutions with unfilled blocks and big size of remainder. The coefficient λ^R is set to 0.1 during the experimental evaluation.

3.4. Selecting best solution

Each time the FM algorithm pass is performed, best solution of pass is retained as a starting point for the next pass. When two solutions are compared during a pass, the following parameters in their lexicographical order determine the better solution: (f, d_k, T^{SUM}, d^E_k), where

- f is the number of feasible blocks (if $f = k$ a feasible partitioning solution is found);
- d_k is the solution cost represented by the infeasibility distance defined in section 3.3;
- $T^{SUM} = \sum_{i=1}^{k} |Y_i|$ is the total number of I/Os of all the blocks;
- d^E_k is the external I/O balancing factor which will be defined later.

The number of feasible blocks has the highest priority because it is determinant for finding the feasible partition. Next, the importance is given to the infeasibility distance cost. Between two solutions with the same number of infeasible blocks and the same infeasibility distance, one which has smaller I/O number is preferred.

The last component is introduced to take into account the assignment of external I/O pins (terminal nodes of the initial circuit) to the devices. This component is needed for I/O-critical designs where the number of external I/Os influences the final result ($\lceil \frac{S_0}{S_{MAX}} \rceil \leq \lceil \frac{Y_0}{T_{MAX}} \rceil$). If external I/O distribution among the created blocks is not taken into account during partitioning, the following scenario may happen. First blocks have few I/Os, and few external I/Os are assigned to them, the following blocks have more and more external I/Os, and at the final iterations the number of external I/Os of the remainder becomes the obstacle for its feasibility. The external I/O balancing factor is calculated as the deviation of the number of external I/Os from the average number of external I/Os computed on the base of minimal theoretical number of blocks : $T^E_{AVG} = \frac{|Y_0|}{M}$. The balancing factor is: $d^E_k = \sum d^E_i$, where $d^E_i = \frac{T^E_{AVG} - T^E_i}{T^E_{AVG}}$ if $T^E_i < T^E_{AVG}$, and $d^E_i = 0$ if $T^E_i \geq T^E_{AVG}$. By T^E_i is denoted the number of external primary I/Os assigned to block i.

3.5. Solution space exploration

Because only semi-feasible solutions are accepted as intermediate solutions between the Algorithm 1 steps, there is no interest to explore solutions which are far from the semi-feasible solutions (having several blocks largely exceeding sizes and / or several blocks with very poor filling). To make the solution space exploration efficient, the explored space should be delimited within the reasonable margins. From one side, strict limits may lead to trapping in a local minimum, from the other side, relaxed limits may lead to wasting time.

In classical FM algorithm, strict size constraint is imposed, and the equal-size partitions are only allowed. In the case of the multi-FPGA partitioning, equal size constraint is necessarily restrictive. It is sufficient that size constraint is respected.

We introduced several heuristics concerning the size of blocks for efficient solution space exploration.

- Allow size-violating moves for non remainder blocks only if the theoretical minimal number of blocks M is not yet reached. When $k > M$, there should be enough free space in the devices to provide a freedom of moves without violating sizes.

- Delimit exceeding of size of the non-remainder blocks. This delimiting should be more restrictive when swapping is performed between two blocks (to give preference to moves "from" the remainder).
- No upper limit is imposed on the moves "to" the remainder. To prevent the remainder from growing too much, lower bound size limitation is imposed on small-size blocks.
- No constraint is imposed on I/O pin violation during the iterative improvement algorithm.

In Figure 3a feasible move region is shown as unbounded horizontally rectangle. It is defined as $S_{MAX} * (1 - \varepsilon_{min}) \leq S_i \leq S_{MAX} * (1 + \varepsilon_{max})$, where ε_{min} and ε_{max} are empirically identified coefficients ($\varepsilon_{min} > \varepsilon_{max}$). From the experimental observations, ε_{min} in the case of iterative improvement between 2 blocks should be more strict, otherwise clusters have a tendency to move "to" the remainder. This leads us to introduce specific coefficients for 2-block (ε^2_{min} and ε^2_{max}) and multi-block passes (ε^*_{min} and ε^*_{max}). This corresponds to Figures 3a and 3b. The following values of the coefficients were determined after an experimental evaluation: $\varepsilon^*_{max} = \varepsilon^2_{max} = 1.05$, $\varepsilon^*_{min} = 0.3$, $\varepsilon^2_{min} = 0.95$. For the remainder block there is no upper limit for cluster move : $\varepsilon^R_{max} = \infty$.

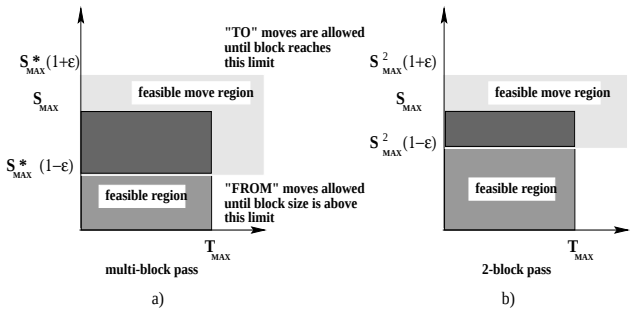

Figure 3. Feasible space for cell move

3.6. Solutions stack

To make the solution space exploration more efficient, we store a fixed number of best solutions during the first FM execution in a solution stack. Current solution is compared with the "head" and "tail" of the stack and probably inserted in the stack. A serie of FM passes is then performed starting from each of these solutions.

As the FM pass is started from the semi-feasible solution, solution stack will contain always only semi-feasible solutions. Sometimes it may happen that infeasible solution with more than one block exceeding the constraints has better infeasibility cost than the best semi-feasible solution. Exploring space around such infeasible solution may allow to escape from the local minimum. To take a benefit of infeasible solutions, we use two solution stacks in parallel - one for semi-feasible solutions, and another one for infeasible solutions. After finishing passes starting from semi-feasible solutions, a serie of passes is performed starting from each infeasible solution in the infeasible solutions stack. At the end, the best result of all the passes is selected. Used solution stacks depth is set to $D_{stack} = 4$. Thus, at most $2 * D_{stack} + 1$ initial solutions is explored during the iterative improvement algorithm call : the first solution, D_{stack} semi-feasible solutions and D_{stack} infeasible solutions.

3.7. Selecting best cluster to move

According to the FM method, best cluster to move is one which has the highest gain defined as number of nets which disappear from the cutset if the cell is moved. If multiple cells have the same gain, [8] proposed the tie-breaking strategy based on the upper-level gains. The effect of higher-level gains was explored in [7] and the conclusion was that higher level gains requires more execution time and does not have significant impact on the solution quality. In the case of multi-way FPGA partitioning, where the net gain is already not directly related with the optimization objective, the impact of higher-level gains becomes even less important.

In the approach described here, the direction of cell move becomes significant. Moves "FROM" the remainder block are preferable than moves "TO" the remainder block. In other words, moves balancing sizes are preferable. This idea comes from the ratio-cut approach ([15]).

We use one bucket structure per move direction. Thus, for k-block partitioning $k*(k-1)$ buckets are maintained. A heap structure is used to keep track of best cell to move, like in [14]. We use 2-level gains and among the clusters with the same gain is selected one which equilibrates block sizes ($MAX(S_{FROM} - S_{TO})$). As the block size reaches boundaries of the feasible move region (section 3.5), bucket corresponding to the move direction "TO"("FROM") the given block is removed from the heap.

4. Experimental results

The experimental evaluation of the described method was done on a set of MCNC Partitioning93 benchmarks proposed in [13] mapped on Xilinx families XC2000 and XC3000. The benchmark data is presented in Table 1. For each benchmark is given the number of primary I/O pins and size in number of CLBs for XC2000 and XC3000 technologies.

Circuit	#IOBs	#CLBs Map to XC2000 families	#CLBs Map to XC3000 families
c3540	72	373	283
c5315	301	535	377
c6288	64	833	833
c7552	313	611	489
s5378	86	500	381
s9234	43	565	454
s13207	154	1038	915
s15850	102	1013	842
s38417	136	2763	2221
s38584	292	3956	2904

Table 1. Benchmark circuits characteristics

Partitioning experiments were performed using the devices XC3020 ($S_{ds} = 64, T_{MAX} = 64$), XC3042 ($S_{ds} = 144, T_{MAX} = 96$), XC3090 ($S_{ds} = 320, T_{MAX} = 144$) and XC2064 ($S_{ds} = 64, T_{MAX} = 58$). The filling ratio was set to $\delta = 0.9$ for experiments with XC3020, XC3042, and XC3090 devices, and to $\delta = 1.0$ for experiments with XC2064 device. Lower bound M on the number of blocks in Tables 2 - 5 was calculated on the base of these δ values.

Partitioning results were compared with previously published results of [11], [12], [16], [3] and [6] in terms of number of produced devices. Presented in this paper method is named FPART in Tables 2-5. All the results of the FPART algorithm were obtained with the following fixed values of the parameters $\sigma_1 = 0.5, \sigma_2 = 0.5, N_{small} = 15, \lambda^S = 0.4, \lambda^T = 0.6, \lambda^R = 0.1, \varepsilon^*_{max} = \varepsilon^2_{max} = 1.05, \varepsilon^*_{min} = 0.3, \varepsilon^2_{min} = 0.95$.

Circuit	Partitioning into XC3020 devices						M
	k-way.x (p,p) [11]	r+p.0 (p,r,p) [11]	PROP [12] (p,o,p)	(p,r,o,p)	FBB-MW [16]	FPART	
c3540	6	6	6	6	6	6	5
c5315	9	8	9	8	8	9	7
c6288	16	16	12	12	15	15	15
c7552	10	10	9	9	9	9	9
s5378	11	10	11	9	9	9	7
s9234	10	10	9	9	8	8	8
s13207	23	23	21	19	18	18	16
s15850	19	19	17	16	15	15	15
s38417	46	48	44	44	41	39	39
s38584	60	60	60	56	54	52	51
Total	210	210	198	188	183	180	172

Table 2. Results comparison on XC3020 device

As shown in Tables 2 and 3, PROP method succeeds to find the results below the lower bound in a number of cases (c6288 for XC3020 device, c3540 and c6288 for XC3042 device). This may not be possible for methods which do not employ the reoptimization. Thus, the overall result of the (p,r,o,p) method remains the best in the Table 3, and is very close to the lower bound theoretical result (82 vs. 81 devices). The proposed method, FPART, and FBB-MW

Circuit	Partitioning into XC3042 devices						M
	k-way.x (p,p) [11]	r+p.0 (p,r,p) [11]	PROP [12] (p,o,p)	(p,r,o,p)	FBB-MW [16]	FPART	
c3540	3	3	2	2	3	3	3
c5315	5	5	4	4	4	5	4
c6288	7	7	6	5	7	7	7
c7552	4	4	5	4	4	4	4
s5378	5	4	4	4	4	4	3
s9234	4	4	4	4	4	4	4
s13207	11	10	9	8	9	9	8
s15850	8	9	8	7	8	7	7
s38417	20	20	20	19	18	18	18
s38584	27	27	25	25	23	23	23
Total	94	93	87	82	84	84	81

Table 3. Results comparison on XC3042 device

method produce 7 results (of 10) equal to the lower bound in Table 3. But as the device size decreases, and the expected lower bound number of devices M increases, FBB-MW and FPART methods outperform the PROP method, especially for the bigger benchmarks (Table 2). FBB-MW method produces better result than FPART method in one case (c5315), and FPART method produces better result than FBB-MW for two largest benchmarks (s38417 and s38584).

Table 4 presents partitioning results for XC3090 device. The comparison is performed for k-way.x method [11] (called also (p,p)), r+p.0 method [11] (called also (p,r,p)), set covering method SC [3], WINDOW ordering, clustering and dynamic programming-based method WCDP [6], FBB-MW method [16] and presented in this paper FPART method.

For small benchmarks, results of FPART method are the same as results of the k-way.x and r+p.0 methods. For four bigger benchmarks, FBB-MW and FPART methods outperform SC and WCDP and produce the same results, but the r+p.0 method, which reaches the lower bound outperforms them by 1 device.

Table 5 presents the experimental results for XC2064 devices. FPART produces similar results as FBB-MW method, and outperforms WCDP, SC and k-way.x methods.

Table 6 presents CPU time results of the FPART algorithm for all the testcases presented in Tables 2-5. All the experiments were run on SUN Sparc Ultra 5 station. The number of algorithm iterations corresponds to the final number of blocks. Thus, for results with small k required CPU time is smaller.

5. Conclusions

In this paper we presented a new multi-FPGA partitioning approach. The general algorithm is organized in a recursive iterations and is similar to one proposed in [11]. But in-

Circuit	Partitioning into XC3090 devices						M
	k-way.x [11]	r+p.0 [11]	SC [3]	WCDP [6]	FBB-MW [16]	FPART	
c3540	1	1	-	-	-	1	1
c5315	3	3	-	-	-	3	3
c6288	3	3	-	-	-	3	3
c7552	3	3	-	-	-	3	3
s5378	2	2	-	-	-	2	2
s9234	2	2	-	-	-	2	2
Total	14	14	-	-	-	14	14
s13207	7	4	6	6	5	5	4
s15850	4	3	3	3	3	3	3
s38417	9	8	10	8	8	8	8
s38584	14	11	14	12	11	11	11
Total	34	26	33	29	27	27	26

Table 4. Results comparison on XC3090 device

Circuit	Partitioning into XC2064 devices					M
	k-way.x [11]	SC [3]	WCDP [6]	FBB-MW [16]	FPART	
c3540	6	6	7	6	6	6
c5315	11	12	12	10	10	9
c7552	11	11	11	10	10	10
c6288	14	14	14	14	14	14
Total	42	43	44	40	40	39

Table 5. Results comparison on XC2064 device

stead of replication/reoptimization enhancement techniques it uses the classical partitioning techniques ([4], [14]). By introducing the infeasibility distance cost function and carefully tuning the basic parameters of classical partitioning algorithms (definition of feasible move regions, handling solutions stack, selecting best cluster to move) we obtained the results comparable to the best published results ([16],[12]), and even outperforming them for the largest benchmarks.

One of the possible directions of future work may be to try to incorporate the real gain in I/O pin number of a block instead of the gain in number of cut nets into the cell gain of the FM-algorithm. This may more quickly direct the search towards finding solutions respecting the I/O pin constraint. Another enhancement possibility is to reduce time wasted in the infeasible region by stopping the FM pass if current solution moves farther away from the feasible region.

Circuit	CPU Time, sec			
	XC3020	XC3042	XC3090	XC2064
c3540	15.59	2.75	1.00	11.2
c5315	43.99	16.12	6.15	34.74
c6288	89.14	36.45	10.83	64.62
c7552	46.23	14.11	6.05	40.89
s5378	52.09	22.01	3.87	-
s9234	59.47	23.65	3.45	-
s13207	121.51	95.18	91.61	-
s15850	156.25	61.54	15.61	-
s38417	464.66	131.48	78.54	-
s38584	875.26	258.73	184.12	-

Table 6. Execution time results

References

[1] D. Brasen, J.P. Hiol, G. Saucier, "Partitioning with cone structures", *Proc. IEEE/ACM Int. Conf. on CAD.* (1993): 236-239.

[2] W. L. Buntine, L. S. Su, A. R. Newton, A. Mayer, "Adaptive Methods for Netlist Partitioning", *Proc. IEEE/ACM Int. Conf. on CAD.* (1997): 356-363.

[3] N. C. Chou, L. T. Liu, C. K. Cheng, W. J. Dai, and R. Lindelof, "Circuit partitioning for huge logic emulation systems", *Proc. 31st Design Automation Conf.* (1994): 244-249.

[4] C. M. Fiduccia, R. M. Mattheyses, "A linear-time heuristics for improving network partitions", *Proc. 19-th Design Automation Conf.* (1982): 175-181.

[5] L. W. Hagen, D. J.-H. Huang, A. B. Kahng, "On Implementation Choices for Iterative Improvement Partitioning Algorithms", *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems 16/10* (1997): 1199-1205.

[6] D. J.-H. Huang, A. B. Kahng, "Multi-way system partitioning into a single type or multiple types of FPGA's", *Proc. FPGA'95* (1995): 140-145.

[7] S. Hauck, G. Borriello, "An Evaluation of Bipartitioning Techniques", *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems 16/8* (1997): 849-866.

[8] B. Krishnamurthy, "An improved mincut algorithm for partitioning VLSI networks", *IEEE Trans. Comput. C33/5* (1984): 438-446.

[9] R. Kuznar, F. Brglez, K. Kozminski, "Cost minimization of partitions into multiple devices" *Proc. 30-th Design Automation Conference* (1993): 315-320.

[10] R. Kuznar, F. Brglez, B. Zajc, "Multi-way netlist partitioning into heterogeneous FPGAs and minimization of total device cost and interconnect" *Proc. 31-st Design Automation Conference* (1994): 238-243.

[11] R. Kuznar, F. Brglez, B. Zajc, "A Unified cost model for min-cut partitioning with replication applied to optimization of large heterogeneous FPGA partitions" *Proc. EURO-DAC* (1994): 271-276.

[12] R. Kuznar, F. Brglez, "PROP: a recursive paradigm for area-efficient and performance oriented partitioning of large FPGA netlists" *Proc. IEEE/ACM Int. Conf. on CAD.* (1995): 644-649.

[13] R. Kuznar, Latest Partitioning Results and Directories *http://www.cbl.ncsu.edu/kuznar/* .

[14] L. A. Sanchis, "Multiple-Way network partitioning", *IEEE Trans. Comput. 38/1* (1989): 62-81.

[15] Y.-C. Wei, C.-K. Cheng, "Ratio Cut Partitioning for Hierarchical Designs", *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems 10/7* (1991): 911-921.

[16] H. Liu, D. F. Wong, "Network-Flow-Based Multiway Partitioning with Area and Pin Constraints", *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems 17/1* (1998): 50-59.

[17] C.-W. Yeh, C.-K. Cheng, T.-T. Y. Lin, "Optimization by Iterative Improvement: An Experimental Evaluation on Two-Way Partitioning", *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems 14/2* (1995): 145-153.