University of Illinois At Chicago - Politecnico di Milano
Joint Master

High Performance Processors and System Project Report

SPartA: a novel structural algorithm for Multi-FPGA partitioning

Tutor: Marco Domenico Santambrogio

Project Author:
Alessandro Panella
UIC ID: 662147239
PdM ID: 708496

A.A. 2006-2007

Contents

1 Introduction

The design of multi-FPGA systems implies several phases, starting from specification down to FPGA implementation. One of these phases is partitioning, which is crucial to obtain good designs. This paper presents a novel approach to the partitioning problem: *SPartA* (Structural Partitioning Algorithm), which exploits the design hierarchy to obtain good performances and build the basis of a future expansion toward dynamic reconfigurability.

In section 2 the partitioning problem is described both from a global and a VLSI-oriented point of view. Section 3 deals with several different existing approaches. It starts from *classic* methods, such as Kernighan-Lin and Fiduccia-Mattheyses heuristics, and proceeds with the description of some *iterative* algorithms, namely Genetic, Tabu Search and Simulated Annealing. Then, a powerful suite of *multilevel* algorithms called METIS is described. Eventually a *structural* multi-FPGA partitioning approach is explained. Section 4 presents two forms of multi-FPGA partitioning problem. In section 5 the proposed algorithm is explained in detail, through a description the basic concepts, the pseudocode of the main procedures and some results. Section 6 describes the SPartA framework, which provides the front-end and the back-end of the core algorithm. The last section proposes some future works.

2 Problem definition

The goal of partitioning is to divide a set of interrelated objects into a set of subsets to optimize a specified objective. Partitioning is a general problem which can be encountered in a large variety of contexts. This work focuses on the partitioning of VLSI circuits, with particular attention on Multi-FPGA systems. The most used structure on which partitioning is applied is the *graph*, since it easily models almost all of the problems that require a partitioning to be performed. A graph G is an ordered pair *G:=(V, E)*, where V is the set of *vertices*, or *nodes*, and E is a set of pairs of distinct vertices, called *edges* or *arcs*. Vertices and edges can have a (vector of) *weight* associated with them.

The most general graph partitioning problem is the *k-way partitioning*, defined as follows: given a graph *G=(V, E)* with $V = n$, partition V into k subsets $V_1, V_2, ..., V_k$ such that $V_i \cap V_j = \emptyset$ for $i \neq j$ and $\bigcup_i V_i = V$. A balance constraint must usually hold; it can be expressed by the formula $|V_i| \approx n/k$ or, more precisely, $l \leq |V_i| \leq h, \forall i = 1..k$. Partitioning aims at minimizing (or maximizing) one or more objective functions. The most used cost function is the *edge-cut* - or *cutsize* - , which represents the number of edges - or the sum of weights, if the graph is weighted - whose incident vertices belong to different partitions.

The partitioning problem is known to be NP-complete. However, many algorithms have been developed that find reasonably good partitionings. Some of them are presented in section 3.

Some algorithms works with a structure that is a generalization of the traditional graph, called *hypergraph*. A hypergraph is an ordered pair $G = (V, E^h)$, defined by a set of vertices V and a set of *hyperedges* E^h, where each hyperedge is a subset of the vertex set V. The partitioning problem definition for hypergraph is almost identical to the one explained above. Some methods - e.g. *k-means* clustering - do not use graph or hypergraph. However, these methods are not usually used in VLSI design, therefore they are not described in this paper.

2.1 Partitioning in VLSI design

Partitioning is an usual issue in the design of hardware systems. To give an example, it is a frequent situation that a logic circuit need to be splitted into several parts or functional modules, thus enabling a better placing onto a physical chip, e.g. a FPGA. Moreover, a partitioning algorithm could be asked to take into account the constraints due to the limited resources of a chip, making it harder to find a good solution. Effective circuit partitioning heuristics are becoming more crucial with the increasing complexity and dimensions of VLSI designs. These trends make it necessary to partition a design into smaller, more manageable components. In the field of Multi-FPGA systems, it is straightforward that a partitioning stage is required in order to partition the circuit into the different FPGA chips.

Aggiungi 'divide and conquer', 'clustering/unclustering?' - con cit. Bi-partition.pdf

The differences among these few examples show that partitioning problems appear at different stages of VLSI design, and that the same problem can be coped with at different levels. It is possible to roughly identify two main levels of abstraction in VLSI design that, in turn, determine two ways to cope with partitioning problems.

2.1.1 Structural level

As intended here, the structural representation of a system consists of the modules that implement the functional units and the interconnections among them. The structure of the system is usually obtained through the synthesis of a HDL description (e.g. VHDL or Verilog). It is roughly equivalent to a RTL (Register Transfer Level) schematic. This structure can be also interpreted in a hierarchical fashion: the root of a tree represents a top level design, and its children are the first level modules (e.g. IP-Cores). These modules are probably in turn composed by lower level components, and so on. Going down in the hierarchy, the representation of the system becomes finer grained.

In this structure, each module at each level can be represented by a graph - or an hypergraph - and the partitioning methodology can exploit the hierarchy. The main advantage of making a partitioning at this level is that the representation of the system is still modular, thus making the communication problems simpler.

2.1.2 Logic level

At this level, the circuit is represented by a netlist of logic gates (or, if the target is a FPGA implementation, of CLBs) and can be naturally modeled trough a hypergraph. Therefore, the circuit is represented at the finest grain, making the partitioning more accurate. However, the number of nodes is very high, and, since the loss of modularity, the communication among different partitions is more problematic.

3 Existing partitioning methods

3.1 Traditional methods

Traditional approaches take into account the partitioning of a graph in two parts (*bipartitioning*). This fact could represent a strict limitation, but actually these heuristics can be applied recursively in order to cope with a k-way partitioning problem. One of the best known - and most widely extended - graph bipartitioning approach is the Kernighan-Lin algorithm (KL). The KL approach is the basis of another well known heuristic, called Fiduccia-Mattheyses algorithm (FM), which has been developed in order to solve hypergraph partitioning problems.

3.1.1 Kernighan-Lin graph bipartitioning heuristic

The KL algorithm ([1, 2]) is a heuristic that computes balanced bipartitionings of graphs, aiming to minimize the cutsize of the generated partitions.

It is an iterative-improvement algorithm, in that it begins with an initial random partition and iteratively modifies it trying to improve the cutsize.

The KL algorithm is based on a 2-loop structure, and its pseudocode is shown in 3.1. It works as follows: first, a pair of vertices belonging to different partitions is chosen such that its swap gives the largest gain or the smallest increase in cutsize. These vertices are then swapped and marked as *locked* (i.e. they are not allowed to move again during that pass). At this point the gain for each vertices pair needs to be updated. This process continue until no unlocked pair of vertices exists. After that, the index of the greatest partial sum above the performed swaps (i.e. the first k swaps that give the maximum gain) is computed. If this maximum gain is greater than zero, the correspondent swaps are actually executed, all the nodes return to be unlocked, and a new iteration of the inner loop is performed. If the maximum gain is less or equal than zero, the algorithm ends.

Algorithm 3.1: Pseudocode of Kernighan-Lin heuristic

```
Create initial partitioning;
while cutsize is reduced do
    while While valid moves exist do
        Find unlocked pair with vertices belonging to different partitions that
        most improves/least degrades cutsize when swapped;
        Swap the two nodes and mark them as 'locked';
        Update gain of nodes connected to moved nodes;
    end while
    Compute the largest partial sum and its index k;
    Keep the first k pairs swapped, unswap others;
    Unlock all nodes;
end while
```

The fact that also little worsening in cutsize are allowed in the inner loop determines one of the most relevant features of this heuristic, that is the ability to climb out of local minima. The complexity of this algorithm is quite high, $O(n^3)$.

3.1.2 Fiduccia-Mattheyses hypergraph bipartitioning heuristic

The FM variation ([3, 2, 4]) of KL algorithm has been developed to cope with hypergraphs. In FM algorithm the moves are not swaps of two vertices but single-vertex moves with a balance constraint. A gain is associated with each vertex in the hypergraph, corresponding to the sum of the different gains associated with every adjacent hyperedge. For example, if a node v is adjacent to net i and it is the only vertex of i belonging to a given partition, moving it to the other partition decreases the cutsize by the weight of i (i.e. the gain is equal to i's weight). The sum of such gains (one for each adjacent hyperedge) gives the overall gain for vertex v.

The vertex with the largest gain is moved to the other partition if this move does not violate a given balance constraint, and in that case the node is marked as locked. As in KL heuristic, after a move is executed all affected gains need to be updated. All other aspects of FM heuristic are substantially the same as KL algorithm, and are not listed here.

3.2 Iterative methods

3.2.1 Genetic and Tabu Search algorithms

During the last years, several genetic algorithms (GAs) for partitioning have been proposed. The solutions proposed in [5, 6] cope with multi-objective partitioning problems. In particular, the proposed methodology tries to find a good trade-off among cutsize, time delay, power consumption and good balance. This trade-off is searched by the use of a fuzzy logic cost function that takes into account all the mentioned objectives. Two algorithms have been developed on the basis of this multi-objective approach.

The *GA* algorithm implements a traditional GA approach. It starts with a set of initial solutions called *population* that is generated randomly. In each *generation* (i.e. iteration), each individual chromosome in the population is evaluated using a *fitness function*. Then, in the *selection* phase two of the above chromosomes are selected from the population. The individuals having higher fitness values are more likely to be selected. After that, different operators act on the selected individuals in order to generate new individuals called *offsprings*. This genetic operators are *crossover* (applied to two individuals) and *mutation* (applied to a single individual). The ways these operators are applied cause different trends in new generations: to give an example, a strong application of mutation cause the offsprings to be more *memory-less* but also increase the ability to climb local minima. In order to evaluate the quality of an individual, the fuzzy logic cost is applied as the fitness function.

The *Tabu Search* (TS) approach starts from an initial feasible solution and carries out its search by making a sequence of random moves or perturbations. A Tabu list is maintained that stores the attributes of a number of previous moves. In each iteration, a subset of neighbor solutions is generated and the best move is is chosen, according to the fuzzy logic cost function, provided it is not in the Tabu list. This prevents taking the search process back to already visited solutions. However, the Tabu list can be overridden by the *aspiration criterion*: if the found solution is the best seen so far, it is accepted regardless of the Tabu list.

3.2.2 Simulated Annealing algorithm

Simulated annealing ([7, 8]) is an iterative algorithm that continuously update a candidate solution until a stop condition is reached. The pseudocode of the algorithm is the following:

Algorithm 3.2: Pseudocode for Simulated Annealing algorithm

```
$T = T_0$
CurrentGain = CalculateGain()
while $t_{stop} > 0$ do
    AcceptMove = FALSE
    for $i = 1$ do
        randomly select vertex V to move from one partition to another
        NewGain = CalculateGain()
        if AcceptGainChange($\Delta Gain, T$) then
            CurrentGain = NewGain
            AcceptMove = TRUE
        else
            return V to original partition
        end if
    end for
    if AcceptMove then
        $t_{stop} = t_s$
    else
        $t_{stop} = t_s$
    end if
    $T = T * \alpha$
end while
```

A candidate solution is randomly generated, and the algorithm starts at high *temperature* T_0. The *gain* is computed as follows:

$$Gain = \frac{cutsize}{|A| * |B|}$$

where $|A|$ and $|B|$ are the number of vertices in partitions A and B, respectively. M is the number of *move states* per iteration. When a random vertex is selected for moving from its original partition to another, the move is accepted according to the following rule. If a move will result in an unbalanced partition, it is always rejected. If the balance is preserved and the resulting solution is improved, the move is accepted. Otherwise, the move is randomly accepted with probability $e^{-\frac{\Delta Gain}{T}}$. After each iteration, T is scaled by a *cooling factor* α, where $0 < \alpha < 1$. The algorithm stops if there have been no changes after t_s iterations.

The probability of accepting a worsening move decreases with the number of the current iteration. Therefore, at the beginning worsening moves are more likely to be accepted, and the solution evolves in a more confused fashion. Later, when T decreases, the probability to accept these moves is lower, and the solution evolves with smaller changes. This process is similar to the electrons energy jumps in a hot metal that undergoes a cooling process (annealing), from which this algorithm takes its name. The possibility to accept worsening moves provides again hill-climbing ability in the early stage of the process.

3.3 Multilevel methods

3.3.1 The Multilevel Partitioning Paradigm

The multilevel partitioning ([9, 10, 11]) was successfully introduced in the mid 1990s, and represents the best known partitioning methodology for large graphs or hypergraphs.

The main idea behind multilevel partitioning is the repeated reduction of instance size via clustering. Note that *clustering* is used in this context with a different meaning than *partitioning*. The original problem instance is reduced by constructing a clustering and then collapsing each cluster into a new vertex. This process is repeated until the size of the instance is suitable for applying an efficient and effective partitioning algorithm. Once an initial partitioning solution is computed on the smallest graph, it is projected to the upper levels and it is iteratively refined. The two phase are called, respectively, *coarsening* and *refinement* (or *uncoarsening*). These phases are depicted in figure 3.3.1

3.3.2 Metis graph partitioning algorithm

Karypis and Kumar introduced in [10] a class of efficient multilevel partitioning algorithm called *Metis*. These algorithms compute a k-way partitioning of a graph $G = (V, E)$ in $O(|E|)$ time. As said before, a multilevel partitioning algorithm is composed of three sequential phases: *coarsening phase*, *initial partitioning phase* and *uncoarsening phase*. The description of each phase is provided in the following.

Coarsening Phase During this phase, a sequence of smaller graphs $G_i = (V_i, E_i)$ is constructed from the original graph $G_0 = (V_0, E_0)$ such that $|V_i| < |V_{i-1}|$. In order for a partitioning of a coarser graph to be good with respect to the original graph, the weight of a vertex must be equal to the sum of weight of the vertices of the original graph that were collapsed to form it. Also, the edges of the new vertex are the union of the vertices that were collapsed. This rules ensure two important properties: (i) the cutsize of a

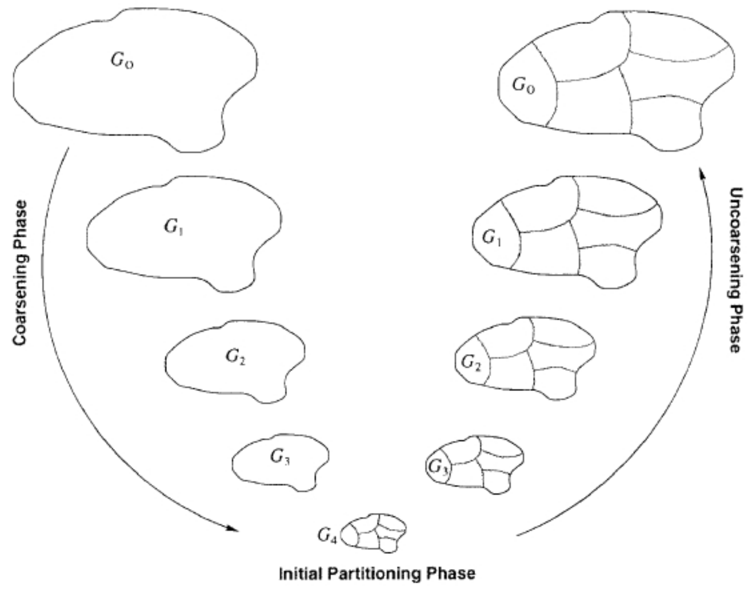

Figure 1: Various phases of a multilevel partitioning algorithm (from [10]).

given partitioning in a coarser graph is equal to the cutsize in the finer graph and (ii) a balanced partitioning of the coarser graph results in a balanced partitioning of the finer graph. This edge method can be formally defined in terms of *matchings*. A matching is a set of edges, no two of which are incident on the same vertex. The coarser graph is constructed by collapsing couple of matching nodes of the original graph. The unmatched vertices are simply copied. Since the goal is to minimize the size of the graph, a maximal matching should be found at each level. A matching is said to be *maximal* if it is not possible to add any edge without violate the matching condition. In [10] are described three ways to obtain maximal matchings, namely *Random Matching*, *Heavy Edge Matching* and *Modified Heavy Edge Matching*, listed here by increasing goodness of the result but also of execution time.

Initial Partitioning Phase This phase aims at computing a k-way partitioning of the coarsest graph such that each partition contains roughly $|V_0|/k$ vertex weight of the original graph. Two ways are described to produce such initial partitioning. The first is to keep coarsening the graph until it has only k vertices left, but it rises two problems. First, the reduction in graph's size becomes very small after some coarsening steps, making it too expensive to continue with the coarsening process. Second, the weight of the obtained vertices are likely to be quite different, making the initial partitioning highly unbalanced. Another way is to use a multilevel bisection algorithm, which produces good initial partitionings and requires small amount of time.

Uncoarsening Phase During this phase, the partitioning of the coarsest graph is projected back to the original graph, by going through all the intermediate graphs. it is important to note that, even if the partitioning of a coarser graph is at a local minimum, its projection to the finer graph may not be at a local minimum, since it has more degree of freedom. Hence, it may be possible to improve the projected partitioning at each level. Traditional KL and FM heuristics tends to produce very good results when used as refinement algorithm. Unfortunately, they produce only bipartitionig, while refining a k-way partitioning is significantly more complicated. However, two variations of FM algorithm that solve k-way partitioning problems have been developed. They are *Greedy Refinement*, which is very efficient but almost lacks any capabilities of climbing out of local minima, and *Global Kernighan-Lin Refinement*, that adds some hill-climbing abilities to the former algorithm.

3.4 Structural methods

The expression *structural methods* includes that partitioning algorithms that do not only works on a large, flattened netlists (hypergraphs), but take into account the hierarchical structure of the design. This structure is usually extracted from the HDL description of the system, and provides useful information for the partitioning process.

3.4.1 Integration of HDL synthesis and partitioning

In [12, 13] a new workflow for multi-FPGA design is introduced that integrates HDL synthesis and partitioning. This flow is composed by two main phases: (1) fine grained synthesis and functional-based clustering and (2) hierarchical set-covering partitioning. These steps are here described in detail.

Fine grained synthesis and functional-based clustering The process begins by parsing a Verilog description of the design. The synthesis aims at the creation of a three-level structural tree. The root node represents the top-level of the design. The HDL description is analyzed in order to find the *modules* that compose the design and their interconnections. The root node of the tree has one child for each module. A module is composed by a set of concurrent *processes*. Every process is a child of a module. In order to obtain clusters with a finer granularity, the process level circuit is decomposed into

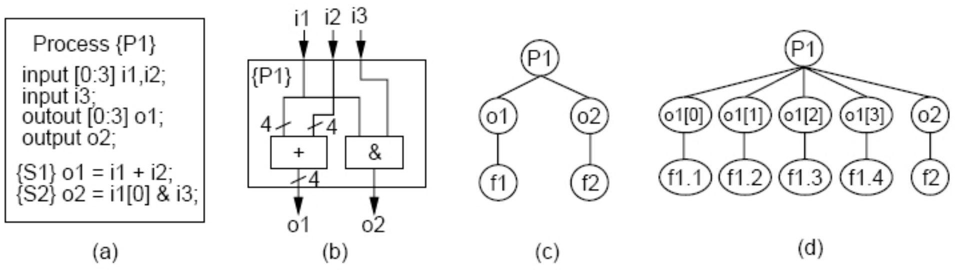

Figure 2: A process example: (a) the Verilog description, (b) the corresponding structure, (c) the structural level, (d) bit-level decomposition (from [13]).

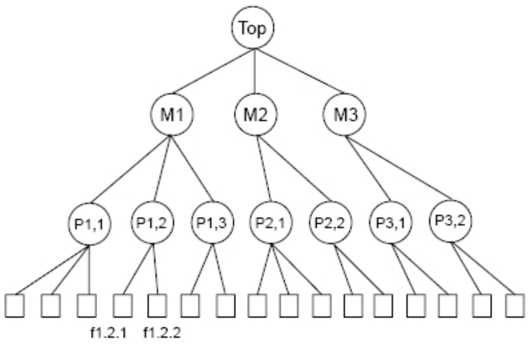

Figure 3: An example of structural tree (from [13]).

smaller clusters. A process consists of a set of statements having a set of input and output signals, so that the synthesis framework generates a logic function for each output. Furthermore, it is possible to decompose multibit logic functions into a set of single bit logic functions, thus achieving a finer granularity of functional clusters. An example of functional clustering of a process is depicted in figure 3.4.1.

The result of this phase is the creation of a hierarchical tree as shown in figure 5.3.3.

Hierarchical set-covering partitioning In this phase a hierarchical connected graph is constructed from the structural tree by mapping each single component at each level of the hierarchy. Then, a set-covering partitioning algorithm is applied. The main objective of the algorithm is to minimize the number of FPGAs (covering sets) used. The amount of logic that can be added to a covering set is limited by the number of available CLBs and I/O pins. The basic idea is to start the set-covering procedure from the highest level nodes (i.e. *module* nodes). If no more feasible covers can be found at the higher level, the process continues on the nodes at lower granularity, that have more possibilities to grant CLB and IO pins constraints. If the constraints are not violated for a component (at any level), it is added to the covering set. If this operation is impossible also at the lowest level of the hyerarchy, a new covering set is created. An example of set covering is depicted in figure 3.4.1. The algorithm ends when all nodes have benn covered. The algorithm execution requires $O(n^2)$ time.

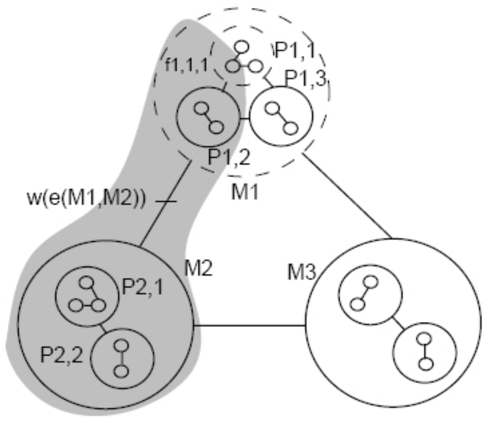

Figure 4: An FPGA covering example (from [13]).

4 Going deeper into the problem

Partitioning problems can be found in several engineering fields, therefore there is the need to circumscribe *our* partitioning problem. The field of interest is VLSI design partitioning. In particular, the goal is to provide a good partitioning algorithm for multi-FPGA design. The first version of the algorithm will cope with static multi-FPGA designs, while in a second moment a more complex algorithm will be developed, that copes with dynamic reconfigurable systems.

The next section describes two problems that can be coped with in this scenario.

4.1 Two different multi-FPGA partitioning problems

The Multi-FPGA partitioning problem can be seen from two different points of view, with different goals. In one case, the actual topology of the FPGA network is entirely provided, while in the other case the topology is not given a-priori. In the following, both cases are analyzed individually.

As it will be clear later, in our case the system to be partitioned is represented by a structural tree of the design hierarchy. The tree can be easily converted into a hierarchical connected graph. This structure is a generalization of a graph, but for the remainder of this section there are no differences between dealing with graphs or hierarchical graphs. Therefore, for the sake of simplicity and clarity, the following concepts will be treated using graph structures.

4.1.1 Topology-aware partitioning

As said above, in this case the topology of the FPGA network is received as an input by the partitioning algorithm, together with the HDL system description.

It is possible to see this problem as a matching between the graph representing the modules of the system to be implemented (i.e. the *system graph*) and the one representing the topology of the multi-FPGA architecture (i.e. the *architecture graph*). The former graph is generally much larger than the latter, so that a partitioning is required. Since the topology of the architecture is fixed, this form of the problem is subject to a large number of constraints that originate from the architectural description. On the other hand, it is obvious that there is no need to minimize the number of FPGA chips (i.e. the number of partitions) because it is imposed by the topology. Anyway, a good exploitation of the area can lead to efficient solutions.

The presence of pre-routed wires among partitions implies more constraints to be respected. Moreover, the algorithm must take into account the eventual presence of communication units to be implemented over some FPGAs, in order to permit the communication between two chips that are not directly connected.

This kind of problem do not always have a possible solution. As a matter of fact, if the dimension of the system is larger than the sum of FPGA areas, the solution is trivially impossible to be found. In that case two ways can be followed: either the designer has the possibility to use a larger multi-FPGA architecture, either dynamic reconfigurability can be considered to solve this space-constrained problem.

4.1.2 Topology-free partitioning

The absence of a given topology implies more degrees of freedom in the design of a Multi-FPGA system. This results in a lower number of constraints to be fulfilled, but also increases the number of tasks that the algorithm has to perform. Actually, the algorithm however needs some architectural informations of a single FPGA node. In particular, it needs the dimension of the used FPGA chip and the number of the maximum incoming and outgoing connections (i.e. fan-in and fan-out). On the basis of these information the procedure aims at minimizing the number of FPGA chips been used while keeping an acceptable communication throughput among the nodes.

Since the number of partitions needs to be minimized, this can be seen as a particular instance of the set covering problem. As a matter of fact, in such a problem the number of *covering sets* used to cover the graph has to be minimized. In this case, while attempting at minimizing the number of partitions, there is also the need to keep the communication volume lower than some limitations imposed by node's architecture. In order to provide a better solution, the quantity of communication can be considered as a second objective, thus resulting in a multi-objective partitioning problem.

More precisely, this approach can be described as follows. There are two inputs for the algorithm: the system to be partitioned and the description of the FPGA nodes to be used. This description basically consists of the dimension of the FPGA chip and the maximum number of incoming and outgoing connections (i.e. fan-in and fan-out) and the type of communication implemented (e.g. direct pin connection, shared-wire connection, ...). Therefore, the algorithm has to perform a mapping between the functional modules of the system graph and a library of FPGA nodes. This implies that it must also carry out some kind of place-and route tasks, in order to select the best connections among the FPGAs.

5 SPartA: the algorithm

The algorithm that is being developed deals with the second type of problem, i.e. it is *topology-free*. Moreover, it has been assumed that[1]:

1. the communication between two FPGAs is carried out by a common link with communication logic on both sides. It means that it is not a pin-to-pin connection, and therefore the FPGA's pin number does not represent a constraint;

2. the FPGAs communication logic can manage an infinite number of connections.

[1]These assumptions reflect the features of the architecture that is being developed in the other currently active branch of the project, that deals with the creation of a physical multi-FPGA platform using *Xilinx Spartan-3* FPGAs.

Before starting with the description of the algorithm, it is useful to point out what are the main objectives the algorithm tries to minimize. There are two cost functions: the amount of communication among different FPGA chips and the number of FPGA chips. The algorithm tries to minimize both objectives in a heuristic and greedy fashion.

One of the main feature of the proposed algorithm is the *hierarchical* nature of the structure on which it works. This characteristic is described in the following subsection.

5.1 The structural approach

As described in section 3, in [12, 13] the authors propose a new approach to multi-FPGA partitioning. The idea is to consider the modular structure of the HDL input design, and exploit the design hierarchy to carry out the partitioning. The methodology works on Verilog system descriptions, that are synthesized in order to produce a three-level - *modules*, *processes* and *functions* - design hierarchy. This tree is then interpreted as a hierarchical connected graph (as shown in figure 5.1), and a set covering algorithm is applied on this three-level hierarchy. The algorithm works on the basis of a *score* function, which is the linear combination of two elements: the amount of communication between the considered node and the current covering set and the ratio between the number of CLB and I/O pins of the considered node. Two parameters allow to tune these two cost functions. The node with the highest score is considered for being added to the current covering partition.

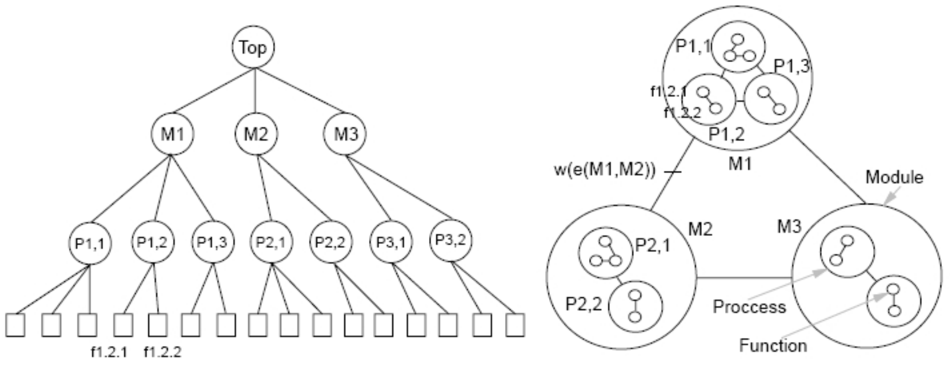

Figure 5: Example of: (a) hierarchical tree and (b) hierarchical connected graph (from [13]).

Maintaining information about design modules and hierarchy entails some important benefits. First, modules must not be broken up into a large set of basic logic elements, but can be treated as single blocks at the highest levels of the hierarchy. This implies less work to be done in order to carry out the partitioning: as a matter of fact, the possibility of adding a given module as a whole to a partition allows to avoid further partitioning activities in the lower levels of the design hierarchy. Moreover, carrying out partitioning considering modules is a good choice for the algorithm objectives as well, such as minimizing the amount of communication among partitions. This fact derives from a simple observation: in a modular design, modules represent work units, that means that the logic components inside them work in a strictly correlated way and therefore there is a lot of communication inside modules. Conversely, communication between different modules is somehow lower. In other words, usually *intra-module* communication is higher than *inter-module* communication. Therefore, the task of minimizing the amount of inter-chip communication has already be partially done by the designer when creating the HDL modular description, and the algorithm can exploit these important information. Another important advantage of dealing with modules during partitioning is that it provides a suitable base for further development of the algorithm toward multi-FPGA dynamic reconfiguration, which is the overall goal of the DReAMS project.

The basic idea of exploiting the design hierarchy has been inherited for the development of a new partitioning algorithm, called SPartA (Structural Partitioning Algorithm), as explained in the next subsection.

5.2 How the algorithm works

The algorithm deals with a hierarchical tree structure. The depth of the tree is not fixed *a priori*, as it results from the synthesis of a VHDL design (see section 6). Therefore the algorithm is required to behave dynamically with respect to the unknown number of hierarchy levels. This results in a procedure that calls itself recursively while going down through the hierarchy.

At a given moment during the execution of the algorithm a node in the tree can be in one of three different states: *uncovered*, *partially covered* and *covered*. It is partially covered when some of its children are partially covered or only some of them are covered. The algorithm stops when the node *TOP*, which represent the node at the top level of the hierarchy (i.e. the whole design) is covered.

Before describing the algorithm, an important precondition must be presented. In order to be able to carry out the partitioning, the algorithm requires that no leaf greater than the FPGA capacity is present in the tree.

Otherwise, the algorithm is not able to partition the design.

Algorithm 5.1 lists the procedure *cover(set C, int MAXCLB)* which is the main part of the heuristic. It receives as input a set of nodes C; *MAXCLB* is a macro that represents the area of the used FPGAs. The algorithm is started by invoking $cover(\{TOP\})$.

Algorithm 5.1: Procedure *cover(set C)*

```
1.  $S \leftarrow C$;
2.  while $S \neq \emptyset$ do
3.      if $curpar \neq \emptyset$ then
4.          $cur \leftarrow$ uncovered sibling in C having the highest amount
            of communication with nodes in partition curpar;
5.      else
6.          $cur = selectFirstNodeOfPartition()$;
7.      end if
8.      $S \leftarrow S \setminus \{cur\}$;
9.      if $d(cur) < MAXCLB - d(curpar)$ then
10.         $curpar \leftarrow curpar \cup \{cur\}$;
11.         $setStateCovered(cur)$;
12.     end if
13. end while
14. if $top$ is covered then
15.     stop;
16. else
17.     $Q \leftarrow$ set of the uncovered or partially covered siblings of cur;
18.     if $Q = \emptyset$ then
19.         $cover(TOP)$;
20.         return;
21.     end if
22.     $S \leftarrow$ set of the uncovered or partially covered children
        of the node in Q having the highest amount
        of communication with nodes in partition curpar;
23.     if $S = \emptyset$ then
24.         $curpar \leftarrow newPartition()$;
25.         $cover(TOP)$;
26.     else
27.         $cover(S)$;
28.     end if
29. end if
```

Algorithm 5.2: Procedure *setStateCovered(node cur)*

```
1. $cur.state \leftarrow COVERED$;
2. if All other $cur$'s siblings are $COVERED$ AND $cur \neq TOP$ then
3.     $setStateCovered(cur.parent)$
4. else
5.     Set all ancestors to $PARTIALLYCOVERED$;
6.     Set all descendant nodes' status to $COVERED$;
7. end if
```

First, the algorithm tries to add to the current partition the node in C having the highest communication with nodes in the current partition. If this partition is still empty, the node is selected using the function *selectFirstNodeOfPartition*. The pseudocode of this function is not shown, mainly because the choice of the first node to be put in an empty partition is still being investigating. In section 5.3 three different policies to select the node are described and compared.

If the node does not fit into the FPGA chip, the algorithm tries to add the second node having highest communication with nodes in current partition (or a node selected on the basis of a certain policy, if the current partition is empty) and so on, until all nodes in C have been checked out. During this process, if a node fits into the current partition, it is added and procedure *setStateCovered(node cur)* is invoked. This procedure - listed in Algorithm 5.2 - checks whether the node is the last one to be added to a partition among its siblings. In that case its parent's status needs to be set to *covered* as well, in a recursive fashion. When an end is reached (i.e. the current parent node is only partially covered) the node's status is set to *COVERED* while its ancestors' one is set to *PARTIALLY COVERED*, and all the descendant of the node are set to *COVERED*.

At this point the stop condition is checked and, if it is not satisfied, the algorithm proceeds creating a set of uncovered or partially covered siblings of the current node. If this set is empty, it means that all siblings has been covered[2] and procedure *cover* is recursively called. If the created set is not empty, another ser is created: the children of the node having the highest amount of communication with nodes in current partition. If this set is empty, it means that the algorithm has reached a leaf node that does not fit into the current partition. In this case the algorithm, instead of trying to add to the partition some children of another node, prefers to restart the procedure with a new current partition[3]. If the children set is not empty, the procedure is recursively invoked with this set as actual parameter.

5.3 Evaluation

The algorithm has been successfully implemented using C++, and some tests has already been performed. Since the framework which the algorithm is

[2]Pay attention to the fact that this does not imply that the tree has been totally covered, since the parent of these siblings can be different from TOP

[3]This is a first definition of the algorithm, and it is possible that, on the basis of some tests, some aspects will be modified.

part of (see section 6) has not yet been developed, a random tree generator (*Treegen*) has been implemented. *Treegen* is able to create random structural trees and generate a random amount of communication between nodes. The generated tree are coherent, meaning that the sum of the sizes of the children of a given node is equal to the size of the node itself. This fact is the result of a simple assumption: the sparse logic (i.e. communication logic) that can be present at a given level of the tree has been considered as embedded with nodes or - with the same effect - as a node itself.

The real partitioning algorithm (*SPartA*) takes as input two files: one describing the structure of the tree and one describing the communication between nodes. These two files must obviously be compatible, in the sense that the communication - described as weighted node pairs - must fit on the tree. The program returns two files:

- a description of the partitions and the evaluation metrics (paragraph 5.3.1), and
- a *dot* file for drawing the partitions.

5.3.1 Evaluation metrics

To evaluate the goodness of a design three metrics are considered. They are:

1. CUTSIZE (or EDGECUT): it is the overall sum of weights of connections between nodes in different partitions;
2. FILLING: it is the average percentage of occupied space in every partition;
3. SPLITS: it is the amount of splits that have been performed to create partitions. In other words it indicates how many times a module has been decomposed into submodules to fill FPGAs. This metric has been considered because, as said in section 5.1, keeping module's integrity is a good thing.

5.3.2 Selecting first node for an empty partition

The policy adopted for selecting the first node to be put in an empty partition surely affects the quality of the partitioning. It is still an open issue, in the sense that a satisfying policies exploration has not yet been carried out. However, three methods have already been developed, implemented and tested. They are described in the following.

- OPTION 0: a raandom node is selected from the considered set (S in algorithm 5.1);
- OPTION 1: the uncovered node belonging to the considered set having the highest overall communication is selected;
- OPTION 2: the uncovered node belonging to the considered set (S in algorithm 5.1) having the highest communication *with nodes in* S is selected;

The performances of these three different policies have been compared and the results are shown in the next subsection.

5.3.3 Results

In order to provide an idea of what the algorithm produces, the result of a partitioning is shown in figures 5.3.3 and 5.3.3. The first figure represents the original tree as generated by *Treegen*. The second figure depicts the result of the partitioning using OPTION 0 policy. Note that the children of nodes in partition are not showed since they do not provide further information and in that way the figure is more readable.

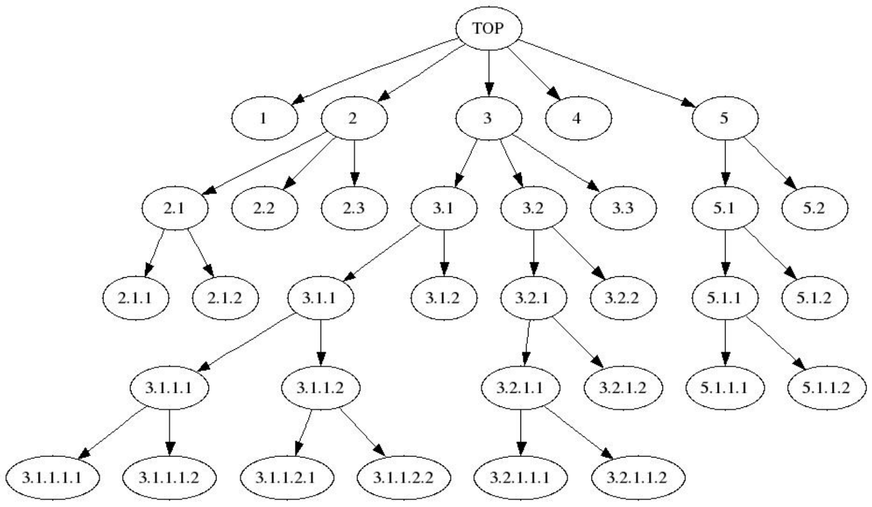

Figure 6: Example: original tree.

The three different policies to select the first node to be added to an empty partition have been tested through the use of 18 randomly generated trees having varying dimensions, partitioned using varying FPGAs sizes. The results are shown in figures 5.3.3, 5.3.3, 5.3.3. One important drawback of

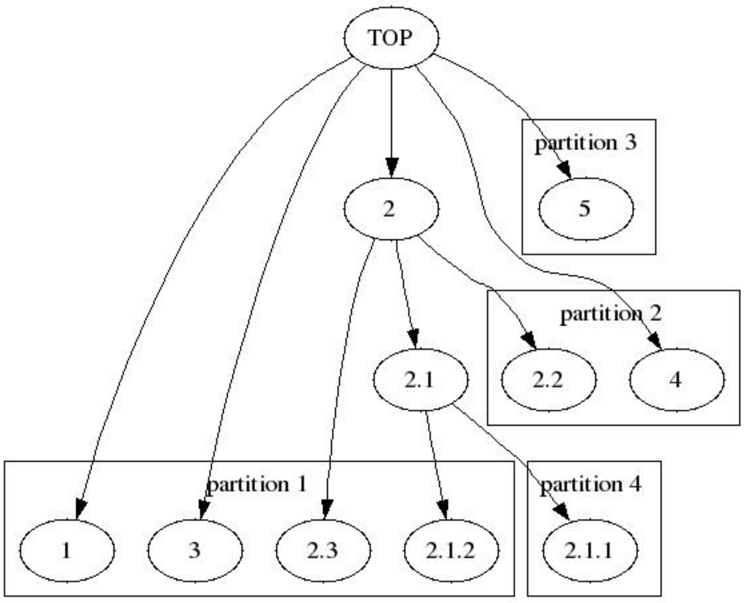

Figure 7: Example: partitioned tree.

the greediness of the algorithm is that the last considered partition is usually highly unbalanced, meaning that it contains an amount of logic depending mainly on the overall dimension of the design and the size of the used FPGAs. This issue will be addressed in the future work (see section 7). For this reason, the last partition is not considered in computing the FILLING metric.

From the figures it can be evinced that no method provides drastically better performances. In particular, for what concerns EDGECUT metric, the results are almost the same for the three policies. Considering FILLING, it can be noticed that OPTION 0 first node selection method often provides a slightly better result. Only the SPLITS metric graph shows a pretty remarkable result: the random method (OPTION 0) produces better results for large designs.

Notice that the average FILLING is very high: this means that FPGAs area is almost all used, thus preventing space wasting. This in turn implies an optimized number of FPGA used. Another result is that often only a single node is splitted several times. This means that that algorithm implicitly try to preserve module integrity, by using 'pieces' of already broken modules to fill unused FPGA areas.

The complexity of the algorithm is exponential, due both to the visible recursion of *cover* (see algorithm 5.1) and implicit recursions of subroutines needed by the implementation. However, the algorithm has been tested with

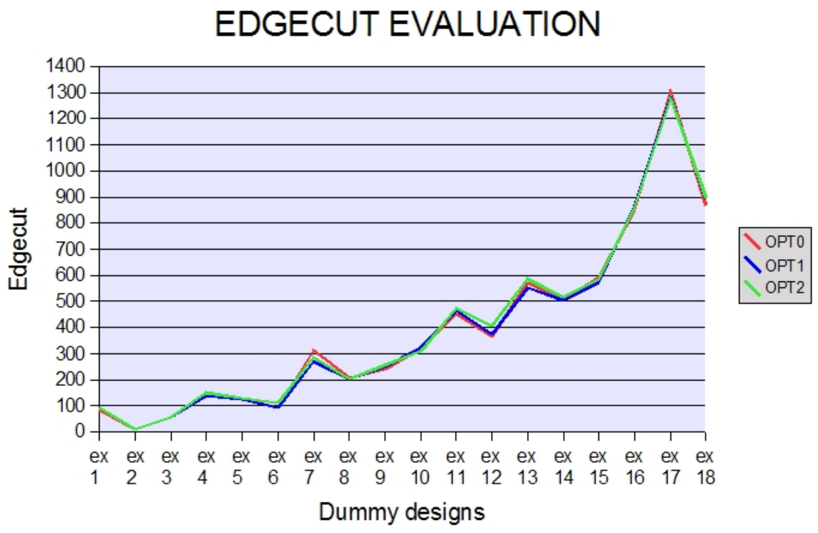

Figure 8: Test: EDGECUT evaluation.

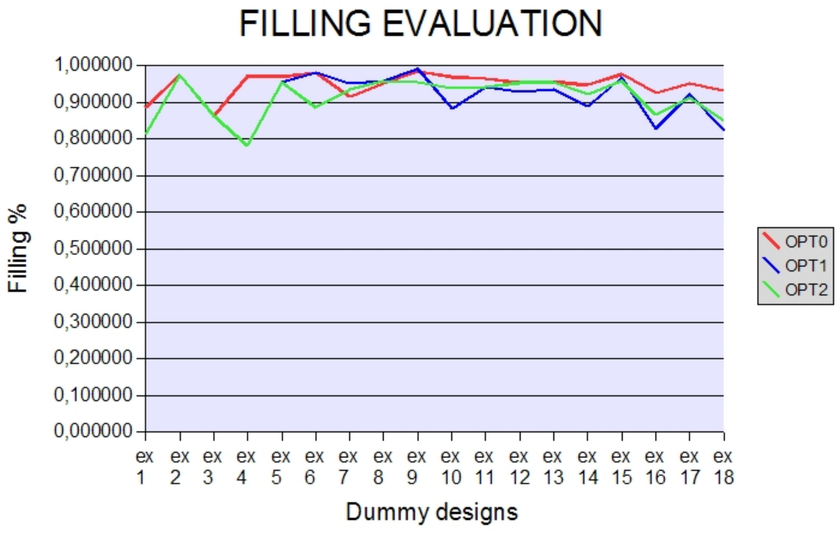

Figure 9: Test: FILLING evaluation.

graphs that could represent quite large designs: the execution time was surely acceptable (few tens of seconds).

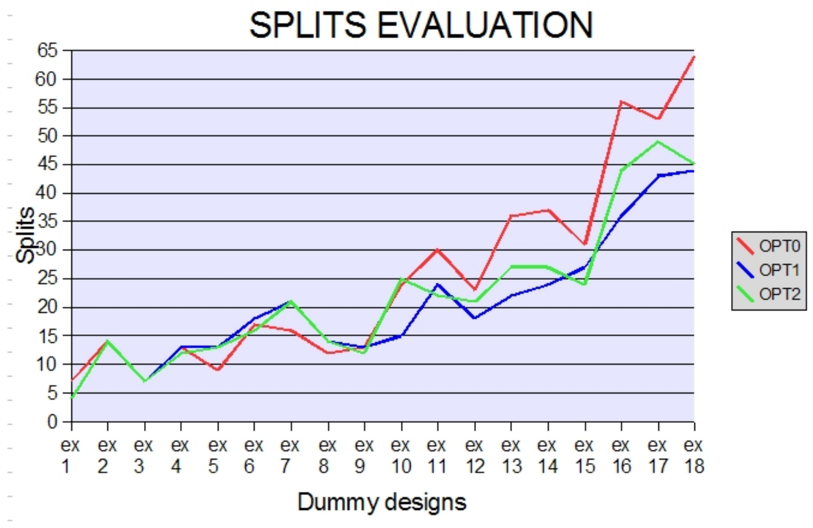

Figure 10: Test: SPLITS evaluation.

6 SPartA: the framework

In this paper a novel partitioning algorithm has been presented. However, this algorithm is thought as the core functionality of a larger framework, which embeds the real partitioning phase into a wider process. The overall framework provides the front-end and the back-end for the algorithm.

Generally, an algorithm takes some data as inputs ad produces some outputs. In our case, the input of the partitioning algorithm is represented by a hierarchical design that has to be provided in a simple textual format. The output consists of a data structure containing the information about the partitioning. This implies - as said - that the algorithm needs to be contained into a huger component. That component must be able to read the design given by the user and produce a suitable - possibly sinthetizable - output. The SPartA framework is the solution to this problem.

The framework takes as input a VHDL description of the system to be partitioned and, with the help of existing synthesis tools, produces a structural design description that can be handled by the partitioning algorithm. After that, the algorithm processes the data structure and provides the partitioning information. The output of the algorithm is then re-converted into VHDL files. Every file corresponds to a single partition (i.e. FPGA). The flow of the SPartA framework is depicted in figure 6.

At the moment, only the SPartA core algorithm has been implemented. The rest of the framework is one of the main future goals (see section 7).

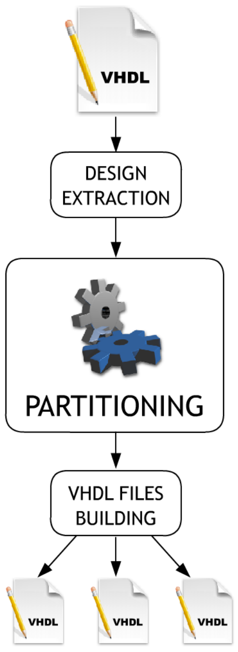

Figure 11: The SPartA framework.

7 Future work

The future work of this project includes both algorithm improvements and extensions. For what concerns improvements, several points of weakness have been highlighted in the document. The balancing of the last partition is one of the main problem that can be addressed. Moreover, the space of methods for the selection of the first node in an empty partition is far from being exhaustively explored. Another improvement will concern the general node selection policy, that is: what node has to be considered for being added to a partition (in the case it is not empty)? Actually, the node with the highest communication is taken into account. Probably a more refined score function which considers more factors could provide better results. In particular, *closeness metrics* will be considered for node's selection. That score function could also assume a parametric nature, allowing the designer to stress a particular objective rather than another one.

For what concerns algorithm expansions, the most effort will be addressed to the development of the SPartA framework, described in section 6. Another expansion concerns *topology-aware partitioning*, that is the form of the problem explained in section 4 that has not been yet taken into account.

References

[1] B. W. Kernighan, S. Lin, “An efficient heuristic procedure for partitioning of electrical circuits,” *Bell Systems Technical Journal*, vol. 49, no. 2, pp. 291–307, February 1970.

[2] S. Hauck, “Multi-fpga systems,” Ph.D. dissertation, University of Washington, 1995.

[3] C. M. Fiduccia and R. M. Mattheyses, “A linear-time heuristic for improving network partitions,” in *DAC ’82: Proceedings of the 19th conference on Design automation.* Piscataway, NJ, USA: IEEE Press, 1982, pp. 175–181.

[4] D. A. Papa and I. L. Markov, “Hypergraph partitioning and clustering,” in *Approximation Algorithms and Metaheuristics*, T. Gonzales, Ed. CRC Press, 2006.

[5] S. M. Sait, A. H. El-Maleh, and R. H. Al-Abaji, “General iterative heuristics for vlsi multiobjective partitioning,” in *Circuits and Systems, 2003. ISCAS ’03. Proceedings of the 2003 International Symposium on*, 2003, pp. 497–500.

[6] ——, “Evolutionary algorithms for vlsi multi-objective netlist partitioning,” *Engineering applications of artificial intelligence*, vol. 19, no. 3, pp. 257–268, April 2006.

[7] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, “Optimization by simulated annealing,” *Science, Number 4598, 13 May 1983*, vol. 220, 4598, pp. 671–680, 1983.

[8] T. W. Manikas and J. T. Cain, “Genetic algorithms vs. simulated annealing: A comparison of approaches for solving the circuit partitioning problem,” Department of Electrical Engineering, The University of Pittsburgh, Tech. Rep. 101, May 1996.

[9] C. J. Alpert, J.-H. Juang, and A. B. Kahng, “Multilevel circuit partitioning,” in *DAC*, 1997, pp. 530–533.

[10] G. Karypis and V. Kumar, “Multilevel k-way partitioning scheme for irregular graphs,” *Journal of Parallel and Distributed Computing*, vol. 48, pp. 96–129, 1998.

[11] ——, “Multilevel k-way hypergraph partitioning,” in *DAC*, 1999, pp. 343–348.

[12] W.-J. Fang and A. C. H. Wu, "Integrating hdl synthesis and partitioning for multi-fpga designs," *IEEE Des. Test*, vol. 15, no. 2, pp. 65–72, 1998.

[13] W.-J. Fang and A. C.-H. Wu, "Multiway fpga partitioning by fully exploiting design hierarchy," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 5, no. 1, pp. 34–50, 2000.