PARTITIONING FOR FPGA CIRCUITS

***Zoltan Baruch*[1], *Octavian Creţ*[2], *Kalman Pusztai*[3]**
[1] Lecturer, *Technical University of Cluj-Napoca, Romania*
[2] Assistant, *Technical University of Cluj-Napoca, Romania*
[3] PhD, Professor, *Technical University of Cluj-Napoca, Romania*

1. INTRODUCTION

In computer-aided design, partitioning is the task of clustering objects into groups so that a given objective function is optimized with respect to a set of design constraints. Partitioning is used frequently in physical design; for example, at the layout level it is used to find strongly connected components that can be placed together in order to minimize the layout area and propagation delay. In the layout synthesis process of FPGA circuits, the partitioning is used in the *placement* step, which assigns each node of the Boolean network to a specific logic block in the FPGA device. Partitioning can also be used to divide a large design into several chips to satisfy packaging constraints.

The partitioning techniques are based on a graph model of the design. Application of these basic partitionig methods requires the mapping of design structures into graph models. Each node in the graph represents a physical component, and each edge represents a physical connection between two components. The main objective of partitioning is to decompose a graph into a set of subgraphs to satisfy the given constraints, such as the size of the subgraph, while minimizing an objective function, such as the number of edges connecting two subgraphs.

The graphs can be partitioned for performance or for physical size. When partitioning for performance, we cluster graph nodes on critical paths while minimizing communication defined by the number of times control or data is passed between clusters. When partitioning for physical cost, we cluster graph nodes by the type of operations they perform while minimizing the number of wires between different clusters [1]. The difference between partitioning for performance and for physical cost can be explained by its efficiency in time and space. Partitioning for performance optimizes time utilization, while partitioning for physical cost optimizes component utilization.

In general, there are two basic partitioning techniques: constructive methods and iterative improvement methods. The constructive method partitions the graph by starting with one or more seed nodes and adding nodes to the seeds one at a time. The iterative improvement method starts with an initial partition, and then succesively improves the results by moving objects between partitions. We describe in the following section an iterative improvement method: min-cut partitioning.

2. THE MIN-CUT PARTITIONING

The min-cut partitioning algorithm (also known as the *Kernighan-Lin* algorithm) partitions a given graph $G = (V, E)$ of $2n$ nodes into two equal subgraphs of n nodes minimizing the connections between the two subgraphs. The algorithm starts with an arbitrary partition of V into two subsets V_1 and V_2. On each iteration the algorithm interchanges k pairs ($k \le n$) of vertices between two sets. It stops when no further improvement is possible.

Consider the partitioning of a graph $G = (V, E)$ with $2n$ vertices into two subgraphs G_1 and G_2 of n vertices each. The cost of each edge $e_{ij} \in E$ is denoted by c_{ij}. For each vertex

$v_i \in V_1$, we define the external cost as:

$$EC_i = \sum_{v_k \in V_2} c_{ik} \tag{1}$$

and the internal cost as:

$$IC_i = \sum_{v_m \in V_1} c_{im} \tag{2}$$

The difference between external and internal costs is denoted by $D_i = EC_i - IC_i$. Similarly, we can define EC_j, IC_j and D_j for each vertex $v_j \in V_2$.

Let c_{ij} define the number of edges between v_i and v_j. For any two vertices $v_i \in V_1$ and $v_j \in V_2$, we define the gain of interchanging v_i and v_j as:

$$gain\_cut\ (v_i, v_j) = D_i + D_j - 2c_{ij}. \tag{3}$$

The cost c_{ij} contributes to both external costs EC_i and EC_j. After interchanging v_i and v_j, the contribution of those edges to the external cost remains the same.

The *Kernighan-Lin* algorithm interchanges a favorable group of vertices instead of interchanging one pair of vertices at a time. The algorithm first arbitrarily partitions vertices into two groups of equal size. Then it computes the external costs, internal costs and the differences between these costs for all vertices. The algorithm finds a pair of vertices, one from each group, that generates the maximal gain through interchange. It stores the gain, readjusts the cost and locks the selected pair to prevent it from being interchanged again. This procedure continues until all *n* vertices in each subset are paired and a sequence of gains, $gain\_cut_1$, ..., $gain\_cut_n$, is generated. The total gain of interchanging the first *k* pair of vertices, where $1 \leq k \leq n$, is calculated as:

$$GAIN\_CUT(k) = \sum_{i=1}^{k} gain\_cut_i \tag{4}$$

The algorithm interchanges in reality only the first *k* pairs of vertices for which $GAIN\_CUT(k)$ is maximal. If for all *k*, $GAIN\_CUT(k)$ is equal to or less than zero, the algorithm stops.

This two-way partitioning algorithm can be extended to implement multi-way partitioning. Given the problem of partitioning a set *S* into *m* subsets, the multi-way partitioning algorithm executes two-way partitionings repeatedly to produce *m* subsets.

3. CONGESTION-BALANCED PARTITIONING

The only metric in the cost function in traditional partitioning algorithms, applied for the placement of FPGA circuits, is the cut size. However, the cut size alone is not a good metric for architectures with limited routing resources, such as FPGAs and CPLDs. Since the algorithm tries to place connected blocks close together, it is likely to generate a placement with congested areas, where a feasible routing is difficult to find. In other words, it is possible to obtain a partition with a small cut size, with one portion being heavily connected and the other being very sparse. For FPGA applications, min-cut based placement algorithms must be modified to take into account not only the sizes of the two portions, the size of the network crossing the cut-line, but also the distribution of interconnections within the two portions.

We describe a modified min-cut bi-partitioning algorithm, that not only balances the size of the two portions, but also evenly distributes the connections among them. We consider that multiple terminal nets are represented by a hyper-graph model. In general, to connect k terminals, $max\{k\text{-}1, 0\}$ connecting paths are needed. We define the *unbalancing number* of a net to be the number of connecting paths needed to connect all the terminals in the left portion minus the number of connecting paths needed in the right portion. The unbalancing number of a bi-partition is defined to be the sum of the unbalancing numbers of all nets. The absolute value of the unbalancing number of a bi-partition counts the difference between the numbers of connecting paths needed in the left portion and the right portion [2].

Given an initial bi-partition, we can compute its unbalancing number in $O(|T|)$ time by examining all the nets, where T is the set of all terminals. We assume that there are more interconnecting paths in the left portion than in the right portion, that is, the unbalancing number of this bi-partition is positive. If a node v is moved from the left portion to the right portion, we can compute $f(v, e)$, the amount by which the unbalancing number of the net e decreases, by examining the set of all neighbors of v, $N(v)$.

The following cost function is used to incorporate the effect of congestion distribution in a given partition:

$$Cut\_size + WEIGHT \times Unbalancing\ number$$

where *WEIGHT* is a constant. If *WEIGHT* is set to zero, then the algorithm is the same as the conventional min-cut partitioning algorithm. By setting the value of *WEIGHT* appropriately, we can control the importance of balancing the congestion.

We consider a graph $G = (V, E)$ with $2n$ vertices. The array *Locked* stores the value 0 if a vertex is available for interchange and 1 if the vertex is locked. Procedure *PART_INIT* (G, n) selects a subgraph of size n. Procedure *EXCHANGE* (V_1, V_2, v_i, v_j) interchanges vertices $v_i \in V_1$ and $v_j \in V_2$ in two subgraphs G_1 and G_2 respectively. The array *gain* stores the gain for each pair of vertices, and arrays *max1* and *max2* store the indices of these vertices. Array *GAIN* stores the accumulated gain for a sequence of n interchanges. Variables *bestk* and *bestGAIN* store the index and value of the maximal accumulated gain, respectively.

Algorithm 1. Congestion-balanced partitioning.

```
G_1 = PART_INIT (G, n);
G_2 = G - G_1;
bestGAIN = ∞;
while bestGAIN > 0 do
    unb_nr = UNB_NUMBER (n)
    bestGAIN = 0; bestk = 0;
    for i = 1 to 2n do Locked (i) = 0; endfor
    for k = 1 to n do
        gain (k) = 0; GAIN (k) = 0;
        /* Find the best pair of vertices to interchange */
        for all v_i ∈ G_1 AND Locked (i) = 0 do
```

```
                for all $v_j \in G_2$ AND $Locked(j) = 0$ do
                    $gain\_cut = D_i + D_j - 2\,c_{ij}$;
                    $unb\_nr\_new = unb\_nr - EC_i - IC_i + EC_j + IC_j$;
                    $gain\_unb = |\, unb\_nr - unb\_nr\_new \,|$;
                    if ($gain\_cut + WEIGHT * gain\_unb) > gain(k)$ then
                        $gain(k) = gain\_cut + WEIGHT * gain\_unb$;
                        $max1(k) = i$; $max2(k) = j$;
                    endif
                endfor
            endfor
            $Locked(max1(k)) = Locked(max2(k)) = 1$;
            /* Update the gain after tentative of interchanging */
            for $i = 1$ to $2n$ do
                if $v_i \in G_1$ AND $Locked(i) = 0$ then
                    $D_i = D_i - c_{i,max2(k)} + c_{i,max1(k)}$;
                endif
                if $v_i \in G_2$ AND $Locked(i) = 0$ then
                    $D_i = D_i - c_{i,max1(k)} + c_{i,max2(k)}$;
                endif
            endfor
            /* Compute accumulated gain */
            $GAIN(k) = GAIN(k-1) + gain(k)$;
            if $GAIN(k) > bestGAIN$ then
                $bestk = k$; $bestGAIN = GAIN(k)$;
            endif
        endfor
        /* Interchange $k$ pairs of vertices */
        for $k = 1$ to $bestk$ do
            $(G_1, G_2) = EXCHANGE(V_1, V_2, v_{max1(k)}, v_{max2(k)})$;
        endfor
    endwhile
```

4. CONCLUSIONS

The presented algorithm has been implemented for the *Atmel 6002* FPGA circuit. First, the description of a system, given in the *ABEL-HDL* language, has been compiled into a set of equations. Then, from this set of equations, a graph has been generated. This graph has been bi-partitioned in a top-down fashion, in accordance with a recursive application of the congestion balanced bi-partitioning.

The algorithm produces good results, in small amounts of CPU time. In addition to a reduction in the maximum cut size, we also observed that the cut sizes are distributed more uniformly than those in the placement obtained by the traditional min-cut algorithm.

REFERENCES

[1] Gajski, D. D., Dutt, N. D.: **High-Level Synthesis**. Kluwer Academic Publishers, 1992.

[2] Sun, Y.: **Algorithmic Results on Physical problems in VLSI and FPGA**. University of Illinois, 1994.