A Placement Algorithm for FPGA Designs with Multiple I/O Standards

Jason Anderson, Jim Saunders, Sudip Nag, Chari Madabhushi, and Rajeev Jayaraman

Xilinx, Inc.
2100 Logic Drive
San Jose, CA 95124 USA
{janders, jims, sudip, chari, rajeev}@xilinx.com

Abstract. State-of-the-art FPGAs possess I/O resources that can be configured to support a wide variety of I/O standards [1]. In such devices, the I/O resources are grouped into banks. One of the consequences of the banked organization is that all of the I/O objects that are placed within a bank must use „compatible“ I/O standards. The compatibility of I/O standards is based on each standard's supply and reference voltage requirements. For designs that use more than one I/O standard, the constraints associated with the banked organization lead to a constrained I/O pad placement problem. Meeting these constraints with a minimal deleterious effect on traditional objectives like minimizing wirelength turns out to be quite challenging. In this paper, we present a placement algorithm that operates in the context of these constraints. Our approach uses a combination of simulated annealing, weighted bipartite matching and constructive packing to produce a feasible I/O placement. Results show that the proposed algorithm produces placements with wirelength characteristics that are similar to the placements produced when pad placement is unconstrained.

1 Introduction

Increasingly fast system clock speeds and modern bus and low-voltage applications have resulted in the proliferation and acceptance of new I/O standards. To keep pace with these developments, programmable logic vendors have recently introduced FPGAs with flexible I/O resources that may be configured to operate according to a wide variety of I/O standards [1]. For example, the XILINX® Virtex™-E FPGA family has I/O resources, called SelectI/O™ resources, that are capable of supporting 20 different I/O standards.

In the Virtex-E FPGA, multiple I/O blocks are grouped together into banks. As a result of the underlying hardware architecture associated with the banked I/O organization, there are restrictions regarding the I/O standards that may be combined together in a single bank. FPGAs are commonly used in applications where they communicate with several devices and buses. Consequently, it has become commonplace

R.W. Hartenstein and H. Grünbacher (Eds.): FPL 2000, LNCS 1896, pp. 211–220, 2000.
© Springer-Verlag Berlin Heidelberg 2000

for a single FPGA design to use multiple I/O standards. This yields a constrained placement problem as a user's I/O objects must be placed in a way that does not violate the rules regarding the I/O standards that can be used together in the same bank. In this paper, we use the term „I/O block“ to refer to a physical I/O resource or slot. We use the term „I/O object“ to refer to an I/O instance in a user's design.

The difficulty of the constrained I/O placement problem is suggested by the example depicted in Figure 1. The figure shows three different placements of a design's I/O objects and core logic. I/O objects with different shading are incompatible and cannot be placed together in the same bank. We assume that there is a single I/O bank per chip edge. Figure 1(a) depicts the placement that might be achieved in the absence of the constraints associated with the banked I/O organization. Figure 1(b) depicts a good constrained I/O placement in which one group of compatible I/O objects is spread between two banks. The placement in Figure 1(c) would be the result if we placed I/O objects using the naïve approach of taking each group of compatible I/O objects and placing them together in a single bank. Notice that the naïve placement in Figure 1(c) has many more long connections than the placement in Figure 1(b), which suggests that the naïve placement is an inferior placement. A human designer would need to have an intimate knowledge of a circuit's connectivity to be able to make intelligent decisions about how I/O objects should be allocated to banks. Clearly, the difficulty of this problem warrants the development of an algorithm to provide an automatic solution.

In this paper, we present a novel placement algorithm that has been developed for FPGA designs that use multiple I/O standards. Commercial CAE tools must be robust enough to deal with such problems and to our knowledge, this paper represents the first published solution. Our approach uses a combination of simulated annealing, weighted bipartite matching, and constructive packing to generate an I/O placement that does not violate the I/O „banking“ rules. Our algorithm is currently being used in the XILINX placement tools for Virtex and Virtex-E FPGAs.

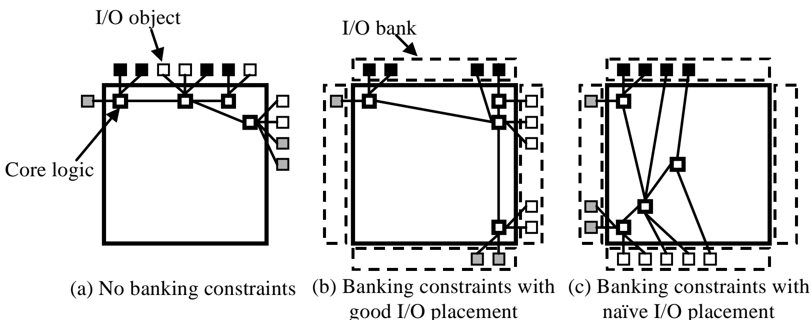

Fig. 1. Difficulty of constrained I/O placement problem

2 Background

In this section, we provide background on I/O standards and the architecture targeted by our algorithm. Following this, we discuss simulated annealing-based placement.

2.1 I/O Standards and Banking Rules

The I/O standards supported by current FPGAs differ from each other in several ways. Some I/O standards require the use of a differential amplifier input. When such standards are used, an external reference voltage, *Vref*, must be provided to the amplifier by the user. Using the differential amplifier allows I/O voltage swings to be reduced, which results in faster switching. A second characteristic is that some standards require a specific supply voltage, *Vcco*, to power the I/O blocks. Figure 2 shows the voltage requirements for some of the I/O standards supported by the Virtex-E FPGA. Notice that *Vref* requirements are associated with input I/O objects; whereas, both input and output I/O objects may have *Vcco* requirements. As shown, bidirectional I/O objects of a particular standard have both the input requirements and the output requirements of that standard.

I/O Standard	Direction	*Vref* Req.	*Vcco* Req.
Peripheral Component Interface (PCI)	Input	Not Req.	3.3V
	Output	Not Req.	3.3V
	Bidirectional	Not Req.	3.3V
Gunning Transceiver Logic (GTL)	Input	0.8V	Not Req.
	Output	Not Req.	Not Req.
	Bidirectional	0.8V	Not Req.
High-speed Transceiver Logic Class I (HSTL_I)	Input	0.75V	Not Req.
	Output	Not Req.	1.5V
	Bidirectional	0.75V	1.5V

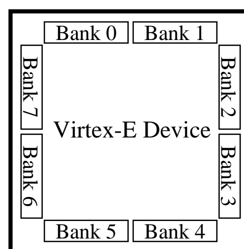

Fig. 2. Some I/O standard voltage requirements (left); organization of Virtex-E FPGA (right)

In the Virtex and Virtex-E FPGAs, the *Vref* and *Vcco* voltages are supplied externally and connect to special pins that serve groups of I/O blocks, called *banks*. All of the I/O blocks in a bank are served by a single *Vref* voltage and a single *Vcco* supply voltage. Virtex-E FPGAs have two I/O banks per chip edge or eight banks in total, as shown in Figure 2. Clearly, the banking of *Vref* and *Vcco* voltages leads to restrictions regarding the I/O standards that may be combined within the same bank: two I/O objects that, because of their I/O standards, require different *Vref* or different *Vcco* voltages cannot legally be used together in the same bank. For example, using the data in Figure 2, we see that an input I/O object that uses the GTL standard cannot be placed in the same bank as an input I/O object that uses the HSTL_I standard because these two standards require different *Vref* voltages.

In Virtex-E, each bank has multiple *Vcco* pins and multiple *Vref* pins. All of the *Vcco* pins in a bank are dedicated pins that cannot be used for user I/O signals. They must be connected to the same supply voltage. On the other hand, the *Vref* pins in a bank may be used to accommodate user I/O signals, if the bank does not contain any user I/O object that is configured to use a standard that needs a reference voltage.

The notions of *Vref* and *Vcco* voltage requirements should make the rationale for the banked I/O organization apparent: if it were possible to configure each I/O block independently, each I/O block would need to have access to separate user-supplied

Vref and *Vcco* voltages. This would greatly increase the number of pins that are committed to receiving *Vref* and *Vcco* and would limit the number of pins available for user I/O signals. The banked organization provides a reasonable trade-off between I/O flexibility and the number of I/O blocks available to the user. We expect that any FPGA supporting multiple I/O standards will employ a banked I/O organization.

2.2 Simulated Annealing-Based Placement

The first step of our placement algorithm uses simulated annealing [2]. Simulated annealing has been applied effectively in many combinatorial optimization problems. Recently published work has shown that simulated annealing produces good results in the FPGA placement domain [3]. A simulated annealing-based placer begins with a random placement of logic blocks and I/O objects. Following this, the random placement is improved iteratively, by choosing pairs of logic blocks or I/Os to swap. The „goodness“ of each swap is evaluated using a cost function. The cost function used is typically designed to minimize estimated wirelength and timing cost. Swaps that decrease placement cost are always accepted. However, swaps that increase cost may or may not be accepted, depending on probabilities. By accepting some swaps that increase cost, the algorithm permits a limited amount of hill-climbing which gives it an opportunity to free itself from local minima.

3 A Placement Algorithm for FPGA Designs with Multiple I/O Standards

The flow of our placement algorithm is as follows: We begin by applying simulated annealing to place a user's I/O objects and core logic blocks. In Section 3.1, we describe a new annealing cost function that contains a component that is directed at resolving the banking rule violations that may be present for designs that use multiple I/O standards. After simulated annealing, we greedily improve the I/O placement using a weighted bipartite matching approach, as described in Section 3.2. Following this, if the I/O placement has no banking rule violations, the placement algorithm terminates. Otherwise, we use a constructive packing algorithm to assign I/O standards to banks in a feasible way. Our packer is described in Section 3.3.

3.1 Simulated Annealing-Based Placement

The annealing cost function we employ takes into account wirelength, timing cost, and the banking violations that result from illegal I/O placements. The function we use is:

$$PlacementCost = \alpha \cdot WirelengthCost + \beta \cdot TimingCost + \gamma \cdot BankingViolationCost \qquad (1)$$

where α, β and γ are scalar constants that reflect the importance of each term. During placement, when core logic blocks are moved, only the values of the wire-length cost and the timing cost may be affected; whereas, I/O movements may affect the values of all three terms of equation 1.

We estimate wirelength using a metric that is based on each net's bounding box. Timing cost is determined in conjunction with user-specified timing constraints. Timing analysis and slack allocation [4] are used to determine a delay slack for each source-to-sink connection. Connection slacks are then translated into a cost function that represents the importance of placing a particular source and sink close together.

The banking rule violation cost in equation 1 is determined by summing the violations for each bank:

$$BankingViolationCost = \sum_{i \in B} BankCost_i \tag{2}$$

where B is the set of all I/O banks. Violations within a bank may occur as a result of *Vref* conflicts or *Vcco* conflicts. A *Vref* conflict occurs when a bank contains I/O objects that, because of the standards they use, would require multiple *Vref* voltages to be applied to the same bank. *Vcco* conflicts are defined similarly. The cost of a bank, *i*, is the sum of the *Vref* conflict cost and the *Vcco* conflict cost:

$$bankCost_i = vrefConflictCost_i + vccoConflictCost_i\ . \tag{3}$$

Banks that contain no I/O objects have no conflicts and are assigned a cost of zero.

To compute the *Vref* conflict cost for a bank, *i*, we first define the notion of a bank's *prevailing Vref*. The prevailing *Vref* for a bank is simply the *Vref* voltage requirement that is most common in the bank. That is, it is the *Vref* voltage that is required by the greatest number of I/O objects in the bank. More formally, the prevailing *Vref* for a bank *i* is:

$$prevailingVref_i = v \mid v \in VREF, NIO_{v,i}^{VREF} = max_{v' \in VREF}\,(NIO_{v',i}^{VREF}) \tag{4}$$

where VREF represents the set of *Vref* voltages used by the I/O objects in the design and $NIO_{v,i}^{VREF}$ represents the number of I/O objects in bank *i* that use I/O standards requiring a *Vref* voltage of *v*. When determining the prevailing *Vref* for a bank, we break ties between multiple *Vrefs* arbitrarily. Banks that do not contain I/O objects that require a *Vref* have a *Vref* conflict cost of zero. Otherwise, for a bank, *i*, the *Vref* conflict cost is given by:

$$vrefConflictCost_i = \sum_{v' \in VREF, v' \neq prevailingVref_i} NIO_{v',i}^{VREF} + NIO_i^{VREF_BLOCK} \tag{5}$$

where $NIO_i^{VREF_BLOCK}$ represents the number of I/O objects in bank *i* that are currently placed in I/O blocks that can receive a user-supplied *Vref* voltage. Recall that in the Virtex and Virtex-E FPGAs, the I/O blocks that receive *Vref* voltages are not dedicated and can be used for user I/O signals if the bank containing them does

not contain any I/O objects that require a *Vref* voltage. In essence, equation 5 states that the *Vref* conflict cost for a bank is the number of I/O objects in the bank that require a *Vref* other than the bank's prevailing *Vref* plus the number of objects placed in I/O blocks that can receive user-supplied *Vref* voltages.

Although not described in this section, we cost *Vcco* violations similarly to *Vref* violations. That is, we compute a prevailing *Vcco* voltage for each bank and from this, compute the number of *Vcco* conflicts for each bank.

3.2 I/O Placement Improvement

Following the simulated annealing-based placement, we enter an I/O placement improvement phase where we use a weighted bipartite matching formulation to improve the placement of I/Os relative to the core logic. In general, the connectivity of I/O objects to core logic is much greater than the connectivity between I/Os. Consequently, we can assume that the cost of placing an I/O object in an I/O block is independent of where other I/O objects are placed as long as the core logic is fixed. Thus, the cost of placing an I/O object in an I/O block can be represented using a static cost function, which allows us to apply weighted bipartite matching to the problem.

A bipartite graph, *G(V,E)*, is a specific type of graph with the property that the vertex set, *V*, can be partitioned into two vertex sets, V_1 and V_2 such that each edge $(u,w) \in E$ indicates that $u \in V_1$ and $w \in V_2$ [5]. In a weighted bipartite graph, each of the edges has an associated weight or cost. A matching, *M*, is a set of edges such that no two edges in *M* have a common vertex. The weight of a matching, *M*, is simply the sum of the weights of the edges in set *M*. In the weighted bipartite matching problem, the goal is to find a matching with minimum weight [6][7].

In our case, the first vertex set, V_1, corresponds to the set of I/O objects being placed. The second vertex set, V_2, corresponds to the set of the available I/O blocks. That is, set V_2 represents the potential placement locations for the objects in set V_1. There is an edge from each vertex in set V_1 to every vertex in set V_2. The cost assigned to each of these edges is:

$$edgeCost_{i,j} = \Delta \cdot \alpha \cdot WirelengthCost + \Delta \cdot \beta \cdot TimingCost + \rho \quad \textbf{(6)}$$

where $i \in V_1$ and $j \in V_2$ and $\Delta \cdot \alpha \cdot WirelengthCost$ and $\Delta \cdot \beta \cdot TimingCost$ represent the changes in wirelength cost and timing cost, respectively, if I/O object *i* were moved from its current position to the I/O block *j*. Let *b* represent the bank containing I/O block *j*. The final term, ρ, is defined as follows:

$\rho = \infty$: if I/O object *i* requires a *Vref*, *v*, and $prevailingVref_b \neq v$.

$\rho = \infty$: if I/O object *i* requires a *Vref*, *v*, and $prevailingVref_b$ has no value.

$\rho = \infty$: if I/O block *j* can receive a *Vref* and $prevailingVref_b$ has a value.

$\rho = 0$: all other cases.

For clarity we describe the values of ρ only for *Vref* violations. The purpose of the ρ term is to maintain an I/O placement that does not violate the I/O banking constraints.

After formulating the problem, we find a minimum cost matching, *M*, which corresponds to an assignment of I/O objects to I/O blocks. If the cost of the matching is non-infinite, then the I/O placement is feasible and our placement algorithm terminates. However, if the minimum cost matching solution has infinite cost, then the annealing step has failed to produce a feasible assignment of prevailing *Vrefs* and prevailing *Vccos* to banks. In this case, we execute a constructive packing step to remedy this infeasibility. In the next section, we describe the packing step.

Our matching formulation is able to repair minor banking rule violations in the annealing placement. Consider the example depicted in Figure 3. The left side of the figure shows an I/O placement after simulated annealing. In this case there are two banks, each containing three I/O blocks. There are six I/O objects to place: three that require a *Vref* voltage of A (labelled A) and three that require a *Vref* voltage of B (labelled B). The annealing placement contains banking rule violations as there is an object labelled B in bank 1 which has a prevailing *Vref* of A. In our matching formulation, each of the objects that require *Vref* A will have an infinite cost edge to all of the I/O blocks in bank 2. Similarly, each of the objects that require *Vref* B will have an infinite cost edge to all of the I/O blocks in bank 1. For this simplified case, the minimum cost matching will have non-infinite cost and it will appear similar to that shown on the right side of Figure 3, which is free of rule violations.

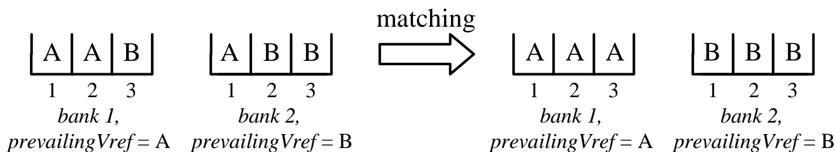

Fig. 3. Example that shows how matching can repair minor banking rule violations

3.3 Constructive Packing

We enter a constructive packing step if the minimum cost matching found in the previous step has infinite cost. In the packing step, we view each bank as a „bin“ and pack I/O objects into bins using a simple bin packing algorithm. The goal of this step is to re-assign a prevailing *Vref* and a prevailing *Vcco* voltage to each bank. We expect that annealing and matching will produce a feasible I/O placement in the majority of cases. The packing step will generally be applied only in very difficult cases, which correspond to designs that use many different *Vref* and *Vcco* voltages.

The bin packing algorithm we use is given in Figure 4. Each of the bins in our formulation has an associated prevailing *Vref*, a prevailing *Vcco,* and a capacity. We begin by initializing each of the bins. This involves setting each bin to be empty and setting the prevailing *Vref* and prevailing *Vcco* voltage of each bin to be unassigned. Next, we sort the I/O objects according to their expected packing „difficulty“. For

example, the group of I/O objects that require both a *Vref* voltage and a *Vcco* voltage impose the most constraints and therefore these will be packed first.

After sorting the I/O objects, we take each object, i, in turn and sort the bins in decreasing order of their affinity to object i. The affinity of a bank, b, to an I/O object, i, that requires a *Vref* voltage, v, and a *Vcco* voltage, o is:

$$affinity_{i,b} = NIO_{v,b}^{VREF} + NIO_{o,b}^{VCCO} \ . \tag{7}$$

Equation 7 says that the affinity of a bank to an I/O object, i, is the number of I/O objects in the bank that require the same *Vref* voltage as object i plus the number objects in the bank that require the same *Vcco* voltage as object i. When computing affinity, we take the values of $NIO_{v,b}^{VREF}$ and $NIO_{o,b}^{VCCO}$ from the annealing placement. By doing so, we establish a preference for assigning prevailing *Vref* and prevailing *Vcco* voltages to banks in a way that is similar to the annealing placement. The effect of this is that we mitigate the potential damage to the quality of the annealing result.

```
bankList ← list of all bins (banks).
Initialize bins in bankList.
ioList ← sorted list of all I/O objects to place (in order of decreasing packing difficulty).
For each I/O object, i, in ioList
        Sort bankList in order of decreasing affinity to I/O object i.
        For each bank, b, in bankList
                If object i can be added to bank b then
                        Add object i to bank b.
                        break.
        If object i could not be packed into any bank then
                Error case: I/O objects could not be packed.
```

Fig. 4. Algorithm for packing I/O objects into banks

After sorting the bins according to their affinities, we employ a greedy approach where we take each bin in turn and check whether the I/O object being packed can be added to the bin. To determine if an I/O object can be added to a bin, we compare the *Vref* and *Vcco* requirements of the I/O object to the prevailing *Vref* and *Vcco* voltage associated with the bin. We do not permit any intermediate illegal packing configurations: for an object to be packed in a bin, it must be compatible with the bin's prevailing *Vref* and *Vcco* assignment. We also do not permit a bin's capacity to be exceeded. When an I/O object is added to a bin, the bin's prevailing *Vref* and prevailing *Vcco* may be affected. Each bin's prevailing voltage values may change from unassigned to a specific value only once; prevailing voltage values will never change from one specific value to another specific value. If we find an I/O object that could not be packed into any bin, automatic placement is deemed unsuccessful. For such cases, we recommend that the user pre-place I/O objects in a feasible manner.

After the packing is complete, we discard its assignment of I/O objects to banks (bins) and use only its prevailing *Vref* and prevailing *Vcco* voltage assignments. We then re-execute the I/O improvement phase of Section 3.2 using these new prevailing voltage values.

4 Experimental Results

To evaluate the quality of result produced by our placement algorithm, we apply it to the problem of placing designs that require multiple I/O standards. The designs that we use in our experiments are Virtex-E customer circuits of various sizes with various I/O standard requirements. We evaluate placement quality using the metric of estimated wirelength. Placement wirelength is estimated by summing the half-perimeter bounding box of the pins of each net.

In this experiment, we place each design twice. We first place each design in an unconstrained manner, ignoring all banking constraints on I/O object locations. Following this, we use our algorithm to place each design, ensuring that banking constraints are strictly obeyed. We then compare the wirelength of these two placements. The aim of this experiment is to investigate how well our algorithm is able to deal with the constraints associated with the banked I/O organization. Another way to view the experiment is from the hardware architecture viewpoint: it addresses the question of whether the banked I/O organization has a deleterious effect on placement wirelength.

The characteristics of our circuits and the experimental results are given in Table 1. Column 2 of the table shows the size of each circuit in terms of the number of slices and I/O objects it uses. Each Virtex-E slice contains two four-input look-up-tables, two flip-flops, as well as other specialized circuitry. Column 3 of the table shows the number of different *Vref* and *Vcco* voltages used by each circuit. The number of *Vref* and *Vcco* voltages used by each circuit reflect the difficulty of each problem as they relate directly to additional constraints on the I/O placement.

Table 1. Characteristics of the benchmark circuits and experimental results

Circuit	# slices/ # I/Os	# Vrefs/ # Vccos	Additional moves (constrained) (%)	Est. wirelength (unconstrained)	Est. wirelength (constrained)	Additional wirelength (%)
Circ1	593/130	2/1	15.2%	4564	4990	9.3%
Circ2	2792/133	1/2	4.7%	28187	28347	0.5%
Circ3	6816/162	2/2	2.3%	113348	122335	7.9%
Circ4	11177/254	2/2	2.2%	235208	236392	0.5%
Circ5	4608/202	2/2	8.3%	36689	36496	-0.5%

We use the same simulated annealing schedule for both the unconstrained and the constrained placement runs. However, because of the non-deterministic nature of the annealing algorithm and the differences in the cost functions used in the two runs, the number of moves made in each of these runs is slightly different. Column 4 of Table 1 shows the percentage increase in the number of annealing moves made in the constrained placement run versus the unconstrained run. Columns 5 and 6 of the table show the wirelength of the unconstrained and constrained placement runs, respectively. Column 7 shows the amount of additional wirelength needed when banking constraints are obeyed versus when banking constraints are ignored. The constructive

packing step discussed in Section 3.3 was not necessary to generate a feasible I/O placement for any of the designs considered.

The results in Table 1 show that our placement algorithm deals very effectively with the constraints imposed by the banked I/O organization. Specifically, the results show that the quality of placements produced when banking rules are obeyed is not significantly different than the quality of placements produced when banking rules are ignored. For all of the circuits considered, adherence to banking rules did not impact circuit wirelength by more than 10%.

5 Conclusions

The key contribution of this paper is to present a placement algorithm for FPGA designs that use multiple I/O standards. The proposed algorithm is unique in that it combines simulated annealing, weighted bipartite matching and bin packing heuristics. The simulated annealing step places a user's core logic and I/O objects using a cost function with a component that is directed at removing I/O banking rule violations. Following simulated annealing, I/O placement is improved using a weighted bipartite matching approach. If annealing and matching fail to produce a feasible I/O placement, the algorithm enters a constructive packing step where I/O objects are packed into banks in a feasible way. Experimental results show that the proposed algorithm deals with the placement constraints effectively, with minimal impact on total estimated wirelength.

References

1. Xilinx Inc., „Virtex™-E 1.8V Field Programmable Gate Arrays," *Product Data Sheet*, http://www.xilinx.com, 2000.
2. S. Kirkpatrick, C. Gelatt and M. Vecchi, „Optimization by Simulated Annealing," *Science*, May 13, 1983, pp. 671 – 680.
3. A. Marquardt, V. Betz and J. Rose, „Timing-Driven Placement for FPGAs," *Proc. ACM/SIGDA Int. Sym. on Field Programmable Gate Arrays*, 2000, pp. 203 – 213.
4. J. Frankle, „Iterative and Adaptive Slack Allocation for Performance-driven Layout and FPGA Routing," *Proc. of the 29th ACM/IEEE Design Automation Conference*, 1992, pp. 536 – 542.
5. T. Cormen, C. Leiserson and R. Rivest, „Introduction to Algorithms," *McGraw-Hill Book Company*, New York, 1994.
6. R. Tarjan, „Data Structure and Network Algorithms," CBMS-NSF Regional Conference Series in Applied Mathematics, *Society for Industrial and Applied Mathematics (SIAM)*, Philadelphia, 1983.
7. M. Fredman and R. Tarjan, „Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms," *Journal of the Association for Computing Machinery*, Vol. 34, No. 3 July 1987, pp. 596 – 615.