Compression-Relaxation: A New Approach to Performance Driven Placement for Regular Architectures*

Anmol Mathur C. L. Liu
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801

Abstract

*We present a new iterative algorithm for performance driven placement applicable to regular architectures such as FPGAs. Our algorithm has two phases in each iteration: a compression phase and a relaxation phase. We employ a novel compression strategy based on the longest path tree of a cone for improving the timing performance of a given placement. Compression might cause a feasible placement to become infeasible. The concept of a slack neighborhood graph is introduced and is used in the relaxation phase to transform an infeasible placement to a feasible one using a mincost flow formulation. Our analytical results regarding the bounds on delay increase during relaxation are validated by the rapid convergence of our algorithm on benchmark circuits. We obtain placements that have 13% less critical path delay (on the average) than those generated by the Xilinx automatic place and route tool (*apr*) on technology mapped MCNC benchmark circuits with significantly less CPU time than* apr*.*

1 Introduction

With the advent of advances in VLSI technology, the size of modules in integrated circuits is becoming smaller and the density of modules on a chip is increasing. Consequently, intra-module delays are becoming smaller and total delay in the circuit is being dominated by delays in the interconnections between the modules. The communication bounded nature of total circuit delay, along with more stringent performance requirements due to more aggressive design style, have made *performance driven layout* an important area of study. To meet the needs of a fast expanding electronics industry, high-performance chips must be designed in a short period. Accordingly, a design flow which incorporates timing analysis and verification into the physical design process is desirable.

The problem of performance driven placement has been studied extensively [1, 2, 4, 6, 9]. The traditional approaches to this problem can be broadly classified into *path-based methods* [4, 9], and *net-based methods* [1, 2, 6]. In this paper, we present a new algorithm for performance driven placement that is applicable to architectures with a regular structure. Thus, the placement problems for Field Programmable Gate Arrays (FPGAs), restricted cell based designs and some Wafer Scale Integration (WSI) architectures fall within the scope of this paper. Our algorithm uses an iterative approach. In each iteration there is a *compression phase* and a *relaxation phase*. The compression phase attempts to make the placement *delay feasible* by compressing the long paths that cause some of the primary output signals to arrive too late. However, the compression phase may produce an infeasible placement with some of the slots occupied by two (but no more than two) modules. This allows the compression phase more flexibility and often allows it to achieve the required decrease in delay. If the compression phase produces an infeasible placement, we require the relaxation phase to obtain a feasible placement. In that case, the relaxation phase carries out a *performance driven reconfiguration* of the infeasible placement to produce a feasible placement. We use the concept of a *slack neighborhood graph* to achieve a provably good reconfiguration, in the sense that the delays in the critical paths are guaranteed not to increase beyond a certain bound. The neighborhood graph is computed using the slacks in the current placement. It captures the freedom of movement, without violating the timing constraints, of the various modules.

Our approach can be viewed as a combination of both path-based and net-based approaches, since the compression phase is path-based while the relaxation phase is net-based. Also, our algorithm simultaneously places the input-output pins and the logic modules. Most iterative placement algorithms proposed in the literature [3] used very local transformations, such as pairwise swap, to move from one configuration to another. Our algorithm makes more global transformations, using flows in weighted slack neighborhood graphs to ensure that the new placement is good. Further, since the flow-based reconfiguration step of our algorithm transforms an infeasible placement (with overlapping modules) into a feasible one, this step can also be used as the back-end for some other performance-driven placement algorithms [9].

2 Problem Formulation

The underlying architecture that is used to implement the given circuit is assumed to have the following two properties:

- **Discreteness**: It is assumed that there are predefined slots in the underlying architecture on which the modules are to be placed. This allows the use of graph based techniques that would not be applicable if the modules can be placed anywhere in the placement

*Work partially supported by NSF under grant MIP 92-22408 and by Fujitsu Company.

area.

- **Regularity**: It is assumed that the modules are to be placed in identical slots.

FPGAs, identical standard cell based architectures and WSI arrays of blocks having a regular structure are the major architectures that satisfy the above requirements. Under these restrictions, the underlying architecture can be viewed as a set of discrete, identical slots: $S = \{S_1, S_2, \ldots, S_m\}$, along with routing resources (routing channels, pre-placed interconnections, programmable switches, etc.). Another characteristic common to all these architectures is that it is easy to estimate interconnection delay without performing detailed routing because of regularity. Also, in the case of FPGAs, all the routing resources are already present, making the prediction of routing delay easier. We assume that the input and output pins are restricted to the periphery of the grid of slots, while the other modules are placed in the *core* of the grid.

The input to the performance driven placement problem is a circuit, C, consisting of a set of modules $\{M_1, M_2, \ldots, M_n\}$, interconnected to implement the required functionality. The topology of the circuit can be abstracted as a *circuit graph*, $G_C(V_C, E_C)$, where each module of C is represented by a vertex in V_C, and an interconnection between two modules is represented by an edge in E_C between the corresponding vertices. In the subsequent discussion, we will often use M_i to denote the *vertex* in V_C representing the ith module. Let

$$PI = \{x_1, x_2, \ldots, x_k\}$$

be the set of primary inputs to the circuit and

$$PO = \{f_1, f_2, \ldots, f_l\}$$

be the set of primary outputs. As part of the problem specification, the arrival times of the signals at the primary inputs, $A(x_i)$, the required arrival times, $R(f_i)$, at the primary outputs and the delays $d(M_i)$ associated with the modules, are also specified. The propagation delay of modules is assumed to be "block-oriented", that is, it is assumed that the latest arriving signal determines the output. These assumptions are introduced only for simplicity of delay computations and are not inherent in the proposed algorithms. The interconnection delay computations in our algorithm use the *general delay model* that allows different wire segments belonging to the same output net to have different delays. This is in contrast with the traditional net based delay models that associate a constant delay with a net, irrespective of its topology. Using the general delay model is specially important when modeling interconnection delays in architectures that have routing resources with widely differing conductivities and capacitances, or when there are programmable switches in the interconnecting paths (as in FPGAs).

A placement is an injective mapping, ϕ, from V_C to the set of slots in the placement area. A placement is said to be *delay feasible* if it satisfies all the timing constraints. The aim of the performance driven placement problem is to find a delay feasible placement that is easily routable. In this paper, we call a placement *(physically) feasible* if no slot is occupied by two or more modules, otherwise it is said to be *infeasible*.

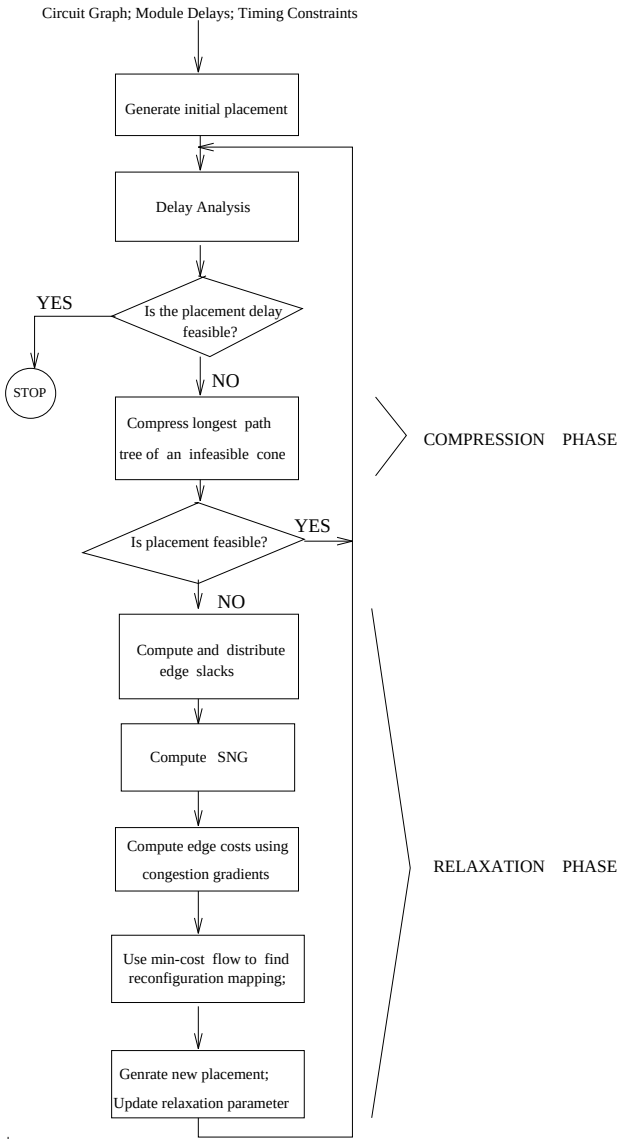

Figure 1: Flowchart for the iterative flow-based performance driven placement algorithm.

3 Overview of the Algorithm

Our algorithm starts with an initial placement generated by placing the modules uniformly in the core of the grid, and the primary inputs and outputs on the periphery. A delay analysis is performed to identify the primary outputs where the required-time constraints are violated. Our algorithm selects, among these outputs, the one where the difference between the arrival time and required time is maximum and attempts to decrease the arrival time at this output in a two phase process consisting of a *compression phase* and a *relaxation phase* (see Fig. 1). In the compression phase, the *longest path tree* of the cone of this output is extracted, and paths in this tree are shortened by decreasing the interconnection delays on the edges in the tree. This is accomplished by moving modules and inputs towards the slot occupied by the primary output. This process is guided by the signal flow direction and is path-based. In the compression phase up to two modules can occupy the same slot (making these slots "overcrowded"). This allows the compression phase to employ a more aggressive compression strategy that would not have been possible if the new placement was constrained to be a feasible one.

If the compression phase produces an infeasible placement, the original modules occupying the overcrowded slots need to be relocated. In the relaxation phase, relocation is carried out simultaneously for all the modules using mincost flow in the slack neighborhood graph in such a way that the delays do not increase by too much. Details of the relaxation phase are described in Section 5.

We show one iteration of our algorithm for the circuit in Fig. 2 in Fig. 3. Fig. 3(a) shows the placement at the beginning of the iteration, along with the longest path tree for primary output f_1. The compression of this longest path tree results in an infeasible placement shown in Fig. 3(b) (overcrowded slots are shown in black). This figure also shows the slack neighborhood graph. Mincost flow is used to find the node disjoint paths shown in Fig. 3(c) and these are used to reconfigure the placement to the feasible placement shown in Fig. 3(d). Some of the details in Fig. 3 will be explained in the subsequent sections.

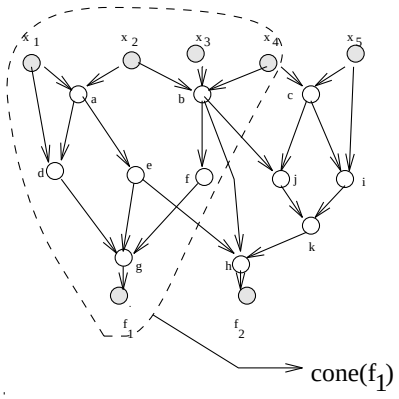

Figure 2: The boolean circuit used in the illustration of an iteration of our algorithm in Fig. 3

4 The Compression Phase

The compression phase attempts to move the current placement closer to a delay feasible placement by reducing the arrival times at the outputs. A delay analysis is carried out on the current placement to compute the actual arrival times of the signals at the various outputs and the outputs where the timing requirement is violated are identified. Associated with each such output is a set of long paths ending at the output, that are responsible for the violation of the timing requirement. We define the *cone* of a primary output f_i to be the set consisting of f_i and all its predecessors. Thus, all the nodes that can have an effect on the arrival time of the signal at f_i are in $cone(f_i)$. The algorithm attempts to satisfy the timing requirement at f_i by moving modules in the cone to new slots so as to decrease edge delays, a process referred to as *compressing the cone at* f_i. It should be noted that in the process of compressing $cone(f_i)$, up to two modules can be placed in the same slot. This ensures that compressing of the cone has enough flexibility to guarantee a substantial decrease in the critical path delay of the cone. If there are several outputs at which the arrival time exceeds the required time, the cone of the output with the maximum violation of the required time is chosen for compression.

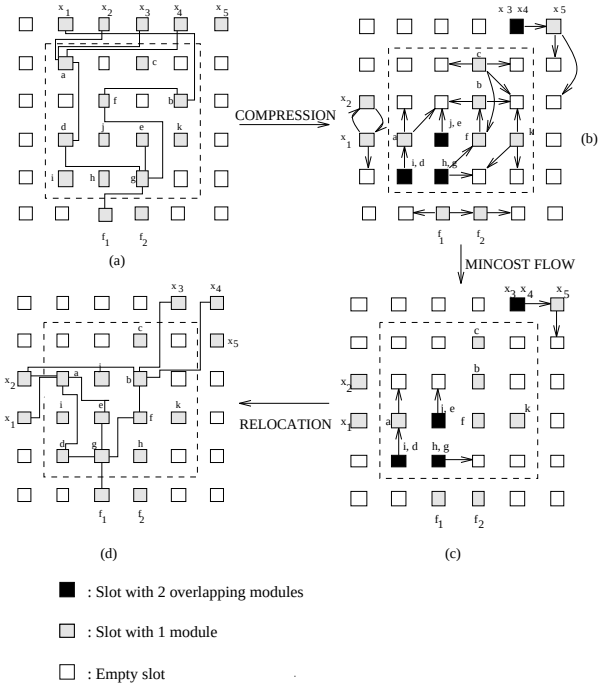

Figure 3: An iteration of our algorithm on the circuit in Fig. 2. (a) The initial placement and the longest path tree of $cone(f_1)$. (b) The infeasible placement obtained after compressing the longest path tree of $cone(f_1)$. The edges of the slack neighborhood graph are shown. (c) The edges with non-zero flow in the mincost flow computed on the transformed SNG. (d) The new placement obtained after relocating modules along the node disjoint paths in (c).

The following steps are used to accomplish the compression of a cone, $cone(f_i)$, that violates the required-time constraint:

1. For each node x in $cone(f_i)$, find a longest path from x to f_i. These paths form a tree which will be referred to as the *longest path tree* (see Fig. 3(a)). This tree can be found easily in linear time. Note that there may be several different longest paths trees of a cone. Our algorithm chooses an arbitrary longest path tree of the cone being compressed. Intuitively, the longest path tree captures most of the critical paths in the cone, and is at the same time structurally simple. This makes the compression using longest path trees very effective and relatively simple.
2. The nodes in the longest path tree are processed in topological order starting from the tip of the cone, and an edge (M_i, M_j) in the longest path tree is compressed by moving module M_i closer to the slot occupied by module M_j. The amount an edge is compressed is proportional to the current delay of the edge (long edges have more potential for compression). When module M_i is moved to shorten edge (M_i, M_j), the entire subtree rooted at M_i is displaced by the same amount to ensure that displacing M_i does not increase the arrival time of the signal at M_i. The compression algorithm also keeps track of the compression achieved so far on the unique path from M_j to the root of the longest path tree, thus allowing it to adaptively decrease the amount of compression forced on edge (M_i, M_j) if enough compression has already been achieved on the path from M_j to the root of the longest path tree. Fig. 4 shows an example of edge compression. Note that the compression algorithm makes certain that an IO pin will not be relocated to a slot in the core of the grid, and a logic module will not be relocated to a slot in the periphery.

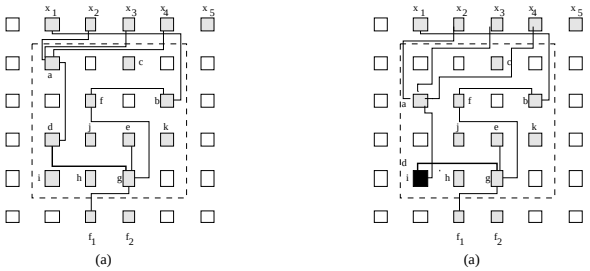

Figure 4: Compression of the edge from module d to module g. (a) Longest path tree and placement before compression. (b) Placement after compression of the edge. Notice that module a also moves down.

Consider the compression phase for the example shown in Fig. 3. Notice that the compression of the longest path tree shown in Fig. 3(a) causes the inputs x_1 and x_2 to move much closer to the output f_1. Also, modules pairs (i, d), (h, g), (j, e) and (x_3, x_4) overlap in the resulting placement.

The compression algorithm attempts to make the task of the relaxation phase easier by relocating modules to empty slots, if they are available at the right distance to achieve the required compression. Further, an attempt is made to spread out the infeasible slots to make the relocation in the relaxation phase easier.

5 The Relaxation Phase

If the compression phase generates a placement with overlapping modules, then the relaxation phase is used to get back to a feasible placement. The input to the relaxation phase is an infeasible placement with some slots containing two modules. During the relaxation phase the placement is reconfigured by moving modules from the overcrowded slots to restore feasibility. In this section we introduce the concept of a *slack neighborhood graph* that provides a systematic framework for doing *performance driven reconfiguration*.

Given a particular placement, we compute the slack of each edge in the circuit graph which is a measure of the amount by which the delay on the edge can be increased without violating any of the timing constraints. Since a delay increase can be translated into an increase in the length of the interconnection corresponding to the edge, the slack can be interpreted as an upper bound on the amount by which the length of the interconnection can be increased, without violating the timing constraints. This has been the basis of several performance driven placement algorithms that compute upper bounds on the lengths of the nets and use them to constrain the placement so that it is timing feasible [1, 2]. However, if the underlying architecture is regular, then it is possible to go one step further and translate the bounds on the amounts by which the interconnections can be lengthened into bounds on the mobility of the modules to which these wires are connected. Further, these bounds on the distance that a module can move without violating any timing constraints imply that a module can only be moved to certain slots in the neighborhood of the slot it currently occupies. Informally, this set of neighboring slots to which a module can be moved without violating any timing constraints is said to be its *slack neighborhood*. The graph in which the adjacency relation reflects these slack neighborhoods is referred to as the *slack neighborhood graph* (see Fig. 3(b)).

5.1 Computing Slacks

Using the delay model described in Section 3, we associate a delay $d(e)$ with each edge e in the circuit graph G_C. We use a function of the semi-perimeter of the bounding box as an estimate of the edge delay. It is assumed that the signals arriving early wait for the later arriving signals. So, if the inputs to a module M_i are denoted by $IN(M_i)$ then the arrival time at the output of M_i is given by

$$A(M_i) = \max_{M_j \in IN(M_i)} \{A(M_j) + d(M_j, M_i)\} + d(M_i).$$

Similarly, the required times at the outputs of the modules is given by

$$R(M_i) = \min_{M_i \in IN(M_j)} \{R(M_j) - d(M_j) - d(M_i, M_j)\}.$$

The arrival times and required times can be computed by the method of dynamic programming using a forward and backward topological sweep of the DAG G_C, in time linear in the size of the circuit graph.

The *slack*, $s(e)$, at an edge $e = (M_i, M_j)$ is defined as the difference between the required and arrival times at the *input* to M_j on edge (M_i, M_j). Thus,

$$s(e) = (R(M_j) - d(M_j)) - (A(M_i) + d(M_i, M_j)).$$

Notice that this definition of the edge slacks does not allow us to increase the delay at all the edges by an amount equal to their slack without incurring the violation of some timing constraints. So, we need to distribute the slacks at the primary outputs among the edges of the circuit graph. This is accomplished using a variant of the zero slack algorithm [8]. The distributed edge slack at edge e is denoted by $ds(e)$.

5.2 The Relaxation Parameter

As observed earlier, in the early iterations, we allow the modules higher mobility than that dictated by the slacks on their interconnections. This enables the algorithm to explore more freely placement configurations that would otherwise have been unreachable from the initial placement. This expansion in neighborhood is accomplished by using a *relaxation parameter* (λ) that is added to the slacks of all the interconnections. The edge slacks with the relaxation parameter added are referred to as the *relaxed slacks* ($rs(e)$). Thus,

$$rs(e) = ds(e) + \lambda.$$

The value of the relaxation parameter is decreased in successive iterations according to a *shrinking schedule*. However, if the relaxation phase in an iteration is not able to reconfigure an infeasible placement, the relaxation parameter is increased to expand the slack neighborhoods. In our experiments we halved the relaxation parameter from one iteration to the next.

5.3 Constructing the Slack Neighborhood Graph

Given the current placement (ϕ) and the relaxed slacks of all the edges in the circuit graph, we can use them to define the *slack neighborhood* for each slot $S_i = \phi(M_i)$ as

$$\nu(S_i) = \{S_j \in S - S_i | d'(e) - d(e) \leq rs(e) \text{ for all edges } e \text{ incident to module } M_i\},$$

where $d'(e)$ is the delay in edge e when the module M_i is moved from S_i to S_j, everything else remaining the same. So, the slack neighborhood of a slot contains all the slots to which the module in the slot can be moved without increasing the delay in any of the edges incident to the module by an amount more than that dictated by the relaxed slacks of the edges.

These slack neighborhoods can be used to define a slack neighborhood graph (SNG) $G_N(V_N, E_N)$, where there is a vertex in V_N for each slot in S, and the there is an edge (S_i, S_j) if and only if $S_j \in \nu(S_i)$ (We are using S_i to represent the vertex in G_N corresponding to S_i). Fig. 3(b) shows the SNG for the placement generated after compressing $cone(f_1)$. Note that the slots occupied by two modules have no edges directed into them and empty slots have no edges directed out.

The computation of the slack neighborhoods can be carried out with varying levels of accuracy. A crude approximation can be computed by using Manhattan distances to estimate the increases in the delays of various interconnections when module M_i is moved from slot S_i to slot S_j. This is what is done in our current implementation. A more accurate computation would do the local rerouting necessary when M_i is moved from S_i to S_j and use the actual increase in propagation delays in the interconnecting wires to find the neighborhoods.

5.4 Congestion Gradient and Costs

One of the important objectives of the placement step is to ensure routability. We use a congestion metric as a guide to preferentially direct the reconfiguration process to move modules into areas with lower congestion. The motivation for this being that a placement with low *congestion gradient* (or a placement with more uniform congestion) is more likely to be routable.

The computation of congestion gradients requires the division of the given chip area into subareas or *precincts*. These precincts may be defined by dividing the chip area into rectangular subregions. The congestion in a precinct is the number of occupied slots in the precinct and the congestion gradient between precincts is the difference in their congestion. The congestion gradient measures the variation in congestion in various directions and is used to define a cost for each edge in the slack neighborhood graph. The cost of an edge $e = (S_i, S_j)$ in the SNG is given by

$$c(e) = cong\_grad(S_i, S_j) + is\_occ(S_j),$$

where $cong\_grad(S_i, S_j)$ is the difference in congestion between the precinct containing S_j and the precinct containing S_i, and $is\_occ(S_j)$ is 1 if S_j is occupied and 0 if it is empty. Thus, the cost of an edge is higher if it goes into a highly congested area or if it goes to an occupied slot.

5.5 Flow Based Reconfiguration

Reconfiguring the infeasible placement can be accomplished by finding node disjoint paths that start at overcrowded slots and end at empty slots in the slack neighborhood graph. We use a min-cost flow algorithm on a transformed version of the slack neighborhood graph to find these paths. It should be noted that the structure of the SNG and the distribution of the empty slots and overcrowded slots will determine whether a sufficient number of such node disjoint paths exist. If a sufficient number of node disjoint paths do not exist, then we need to increase the relaxation parameter (making the SNG denser), or make the compression less aggressive reducing the number of overcrowded slots.

The nodes corresponding to the overcrowded slots and those corresponding to the empty slots serve as the sources and sinks, respectively, in the flow network. Let V_s and V_t represent the sets of sources and sinks in G_N, respectively. G_N can be transformed into a flow network, G'_N, with one source and one sink by the addition of a dummy source vertex, s, and a dummy sink vertex, t, along with zero cost, infinite capacity edges of the form (s, v) and (w, t), for all $v \in V_s$ and $w \in V_t$.

Now that we have a weighted flow network, G'_N, we can use a mincost flow algorithm [10] to find node disjoint paths in G'_N from the sources to the sinks. We set the capacity of all the nodes in G'_N other than s and t to 1. The following theorems (stated without proof due to space limitation) establish the correctness of the flow based algorithm

for computing the reconfiguration that maps the infeasible placement to a new feasible placement.

Theorem 1. (**Integral Flow Theorem**) *Given a flow network with integer capacities, there exists a mincost maximum flow in which the flows through all the edges are integral values.*

If we restrict our attention to integral flows, the minimum flow through any path having non-zero flow is one unit. Further, all nodes have capacity 1, so no two paths with non-zero flow can share a node (except s and t). These observations yield the following theorem:

Theorem 2. (**Disjoint Paths Theorem**) *In any integral feasible flow f of value W in G'_N, the edges in E_N with non-zero flow can be decomposed into W vertex disjoint paths. Further, each path starts at a vertex $x \in V_s$ and terminates at a vertex $y \in V_t$.*

Thus, the disjoint paths in the SNG computed by the mincost maximum flow start at an overcrowded slot and end at an empty slot with all intermediate slots being singly occupied. Hence, we have the following theorem:

Theorem 3. (**Reconfiguration Theorem**) *Let f be an integral maximum flow of value $|V_s|$ in G'_N, then the reconfiguration function $\rho : V_N \rightarrow V_N$ defined as*

$$\rho(x) = \begin{cases} y & \text{if } f(x,y) = 1 \\ x & \text{otherwise} \end{cases}$$

transforms the infeasible placement into a feasible one.

Fig. 3(c) shows the node disjoint paths computed in the SNG shown in Fig. 3(b). The path (given by the coordinates of the slots; coordinates increase from left to right and from top to bottom) $(5,2) \rightarrow (4,2) \rightarrow (3,2)$ causes module i to be moved to slot $(4,2)$ and module a to be moved to the empty slot $(3,2)$. The node disjoint paths computed using the mincost maxflow algorithm define a reconfiguration function that transforms the current infeasible placement into a new feasible placement. It should be noted that the more local reconfiguration patterns used in other algorithms, such as pairwise swap or fixed length cycles[3], are special cases of the reconfiguration patterns generated by out algorithm. In fact, many of the traditional iterative improvement algorithms for placement can be subsumed within our approach. Since the flow is biased using congestion gradients the resulting placement is likely to be routable. Due to lack of space, we omit the results showing that the relaxation phase results in a bounded increase in delay. Details can be found in [7].

6 Experimental Results

We implemented our algorithm on a Sparc 10. We will refer to the package implementing our algorithm as *sysdias* (an acronym for *sys*tolic-*dias*tolic, reflecting the compression and relaxation phase in each iteration). *Sysdias* was tested on a set of MCNC benchmarks and compared to the *apr* (Automatic Place and Route) tool provided by Xilinx for their 3000 series FPGAs. The MCNC benchmarks were technology mapped and then placed and routed using *apr* to obtain the arrival times at the primary outputs. These arrival times were used as the required times at the outputs for our algorithm. Our placement algorithm was allowed to continue even after it generated a placement that satisfied the required times at all the outputs by decreasing the required times, until it was no longer able to achieve significant delay reduction in an iteration. The placement produced was then routed using *apr* to compute the actual critical path delay. The results of these experiments are shown in Table 1. The critical delay after routing for the placement generated by *sysdias* is less than that for the placement generated by *apr* by 13 percent, on the average. For all the benchmarks *sysdias* terminated in less than 50 iterations (that took a maximum of 6.5 minutes). This is significantly faster than the simulated annealing based *apr*. The total time taken over all the benchmarks by *sysdias* is less than the total time taken by *apr* by a factor of 5. Further, the most significant gains are on the large circuits : *C880* and *alu2*, thus demonstrating the ability of *sysdias* to handle large circuits efficiently.

To demonstrate the convergence rate of our algorithm, we ran *sysdias* on some larger, randomly generated circuits with up to 800 modules. Even for such large examples the convergence rate of our algorithm is very good. Typically, the number of iterations is around 100 and the running time is no more than 10-15 minutes. The results for some of these large circuits are shown in Table 2. Since these circuits are too big to be placed and routed on the Xilinx 3000 series architecture, we have no information on the actual critical path delays for these circuits. The required times at the outputs were generated using the depth of the output as an estimate. The *Initial Delay* column gives the estimated delay of the critical path in the initial placement and the *Final Delay* column gives the (estimated) critical path delay in the final placement. Notice that when the circuit *Random4* is placed on a larger grid, the convergence is faster, possibly due to the fact that compression can be done more aggressively due to low congestion.

It is interesting to observe the manner in which the vector of arrival times at the primary outputs changes from iteration to iteration. Table 3 shows the arrival times at the outputs of a randomly generated example with 100 modules, 10 inputs and 3 outputs, being placed in a 12 by 12 array of slots. In most of the iterations there is a substantial decrease in one of the arrival times, corresponding to the cone that is compressed in that iteration. The arrival times at the other outputs might increase slightly due to the relaxation phase in an iteration. However, the use of slack neighborhoods for relaxation ensures that the delay increase is bounded. In fact, the sum of the arrival times at the outputs decreases from one iteration to the next, most of the time (in Table 3 the exception is from iteration 4 to iteration 5). This provides strong evidence that our iterative compression-relaxation based algorithm converges well.

7 Conclusions

We present a new algorithm for performance driven placement when the underlying architecture is regular. Our iterative algorithm has two phases in each iteration: a compression phase and a relaxation phase. We develop a novel compression strategy based on the longest path tree of a cone. We introduce the concept of a slack neighborhood graph and use it to transform an infeasible placement produced in the compression phase to a feasible one using a mincost maxflow formulation. The slack neighborhood graph approach guarantees that relaxation does not undo the delay reduction achieved during compression. Our ana-

Circuit	# modules	*apr*		*sysdias*		
		Delay (ns)	Time (min)	Delay (ns)	# iterations	Time (min)
cordic	48	56.5	1.3	45.0	25	0.8
count	85	71.5	4.7	60.2	32	1.5
bw	61	32.1	3.0	30.0	20	1.2
f51m	58	92.7	1.8	80.4	21	1.2
frg1	91	79.7	3.4	70.0	32	2.3
comp	103	90.9	7.7	77.0	40	4.3
term1	161	92.9	24.3	70.6	45	5.4
C499	144	79.6	9.5	70.0	33	4.3
C880	210	128.3	28.5	115.2	42	5.5
alu2	232	192.3	91.1	177.0	48	6.5

Table 1: Comparison of critical delay after routing for *apr* and *sysdias*.

Circuit	# modules	Grid Size	Initial Delay (ns)	Final Delay (ns)	# iterations	Time (min)
Random1	400	25 x 25	620	350	70	8.4
Random2	550	25 x 25	710	500	50	6.2
Random3	700	25 x 30	600	450	88	12.4
Random4	800	30 x 30	1230	700	103	15.0
Random4	800	40 x 40	1450	650	90	12.4

Table2: Results for the placement of large randomly generated circuits.

Iteration #	Arrival time (ns)		
	Out1	Out2	Out3
0	120.5	220.0	130.6
1	130.2	150.0	125.0
2	90.0	130.0	130.7
3	95.2	135.6	90.3
4	90.2	120.0	90.3
5	100.3	115.0	100.4
6	90.0	110.0	85.0

Table 3: The arrival times at primary outputs in successive iterations.

lytical results regarding the bounds on delay increase during relaxation are validated by the rapid convergence of *sysdias* on benchmark circuits.

References

[1] T. Gao, P. M. Vaidya, C. L. Liu, A New Performance Driven Placement Algorithm, *Proc. ICCAD, 1991, pp. 44–47.*

[2] T. Gao, P. M. Vaidya, C. L. Liu, A Performance Driven Macro-Cell Placement Algorithm, *Proc. 29th DAC, 1992, pp. 147–152.*

[3] S. Goto, An Efficient Algorithm for the Two-Dimensional Placement Problem in Electrical Circuit Layout, *IEEE Trans. Circuits Syst., Vol. CAS-28, Jan. 1981, pp. 12–18.*

[4] M. A. B. Jackson, E. S. Kuh, Performance-Driven Placement of Cell Based ICs, *Proc. 26th DAC, 1989, pp. 370–375.*

[5] S. Kirkpatrick, C. D. Gelatt, Jr., M. P. Vecchi, Optimization by Simulated Annealing, *Science, 13 May 1983, Vol. 220, No. 4598.*

[6] M. Marek-Sadowska, S. P. Lin, Timing-Driven Placement, *Proc. ICCAD, 1989, pp. 94–97.*

[7] A. Mathur, C. L. Liu, Compression-Relaxation: A New Approach to Performance Driven Placement for Regular Architectures, *Manuscript, 1994.*

[8] R. Nair, C. L. Berman, P. S. Hauge, E. J. Yoffa, Generation of Performance Constraints for Layout, *IEEE Trans. CAD, Vol. 8, Aug. 1989, pp. 860–874.*

[9] A. Srinivasan, K. Chaudhary, E. S. Kuh, RITUAL : A Performance Driven Placement Algorithm for Small Cell ICs, *Proc. ICCAD, 1991, pp. 48–51.*

[10] R. E. Tarjan, Data Structures and Network Algorithms, Chap. 8, *SIAM, 1983.*