Simultaneous Timing-Driven Placement and Duplication

Gang Chen
Magma Design Automation
12100 Wilshire Blvd., Ste 480
Los Angeles, CA 90025, USA
chg@magma-da.com

Jason Cong
Computer Science Department
University of California
Los Angeles, CA 90095, USA
cong@cs.ucla.edu

ABSTRACT
Logic duplication is an effective method for improving circuit performance. In this paper we present an algorithm named *SPD* that performs simultaneous placement and duplication to minimize the longest path delay. We introduce the notion of feasible region and super feasible region to improve the critical path monotonicity from a global perspective. We introduce a constrained gain graph to perform optimal incremental legalization under complex constraints. We also formulate a timing-constrained global redundancy removal problem and propose a heuristic solution. Our *SPD* algorithm outperforms the state-of-the-art FPGA placement flow (*T-VPack* + *VPR*) with an average reduction of up to 27% in longest path estimate delay and 18% in routed delay. The increase in overall runtime is less than 2% and the increase in area is less than 1%.

Categories and Subject Descriptors
B.7.2 [**Integrated Circuits**]: Design Aids – *placement and routing*

General Terms
Algorithms, Design, Performance

Keywords
Logic duplication, legalization, redundancy removal, timing-driven placement, FPGA

1. INTRODUCTION
A typical LUT-based FPGA architecture [2] contains a two-level physical hierarchy: Basic Logic Elements (BLE) and Cluster-based Logic Blocks (CLB). As described in Figure 1, each BLE contains a *K*-input LUT and a flip-flop (FF), and the LUT and FF share the same output. As described in Figure 2, each CLB contains *N* BLEs, *I* inputs and *N* outputs. Each of the *I* inputs can drive all the BLEs, and each BLE drives an output. Here *K*, *N*, and *I* are parameters described by an architecture file. The interconnect delay between BLEs within the same CLB is usually much smaller than the delay between BLEs in different CLBs.

FPGA'05, February 20–22, 2005, Monterey, California, USA.

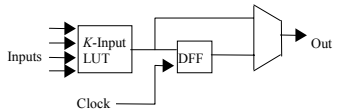

Figure 1. ***VPR*'s Basic Logic Element (BLE)**

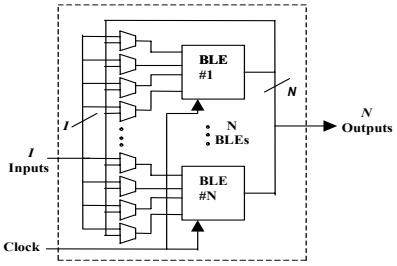

Figure 2. *VPR*'s Cluster-Based Logic Block (CLB)

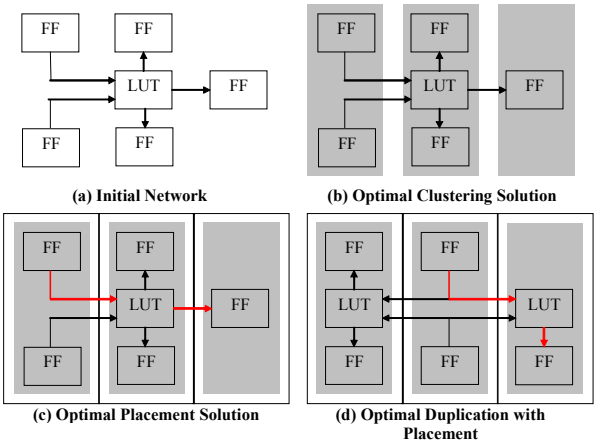

Figure 3. Impact of Duplication on Placement

Logic duplication is a common technique for improving circuit performance by duplicating one or more logic cells while maintaining the logic equivalence of the circuit. Figure 3 illustrates the impact of duplication on placement. The initial network (a) consists of five FFs and one LUT. We assume each CLB contains two BLEs, and the device is a 1 x 3 grid. We assume that the inter-cluster delay equals the Manhattan distance, and both the logic and intra-cluster delay are 0.1. The "optimal" clustering solution in (b), which consists of three CLBs and two logic levels, can be obtained from *T-VPack* [9] (which minimizes both the number of clusters and the number of levels). However, the optimal placement solution (c) on this optimal clustering has a longest path delay of 2.1, which cannot be improved by post-placement duplication. Instead, if we perform duplication together with placement, we can obtain a solution (d) with a longest path delay of 1.2.

In the past, logic duplication for timing optimization has been studied in the following contexts. First, logic duplication has been applied before placement in the logic synthesis domain. Improved circuit performance can be achieved by replicating high fanout logic gates on the critical path to isolate the critical sinks from the non-critical ones [8][13]. Lillis et al. [8] perform gate replication to improve the delay and area of a circuit under certain timing requirements. The gate replication technique complements the popular gate sizing approach in the ASIC flow. Later on, Sarrafzadeh et al. [13] present an effective heuristic algorithm for the gate duplication problem under the load-dependent delay model. They show that both the global and local (fanout partitioning) logic duplication for delay optimization problems are NP-complete.

Second, logic duplication has been applied after placement as a post processing step to further increase design performance for FPGAs. In [1], Lillis et al. propose a heuristic replication algorithm to straighten the locally non-monotone critical paths. A legalization engine based on the "ripple-move" approach in *Mongrel* [6] is used to legalize the placement incrementally. However, [1] cannot improve critical paths that are globally non-monotone yet locally monotone. The average reduction over *VPR* obtained from [1] is 7.5%. The follow-up work [5] improves [1] by incorporating two new techniques: timing-driven fanin tree embedding and replication tree. First, they introduce an optimal algorithm to solve the fanin tree embedding problem under a general cost model. Second they propose a replication tree to introduce large sub-circuits to be solved by the embedding algorithm. The average reduction over *VPR* obtained from [5] is 14.2%.

However, limited work has been done to carry out logic duplication *during* placement. Neumann et al. [11] apply logic duplication in a recursive partitioning-based timing-driven placement flow. During each recursion, they perform timing analysis, net length estimation and weight calculation, bi-partitioning and cell replication sequentially. Before cells are assigned to rows, the redundancies introduced by the replication are removed. This combined approach outperforms gate sizing by 10% on average.

In this paper we propose a novel algorithm to perform simultaneous logic duplication *during* placement for timing minimization. We introduce the notion of feasible region and super feasible region, which enable the optimization of non-monotone paths from a global perspective. We present an optimal incremental legalization algorithm under complex constraints. We also formulate a timing-constrained global redundancy removal problem and propose a heuristic to solve it by solving the local redundancy removal problems optimally. Finally, we incorporate a path counting-based net weighting scheme in our approach. The resulting algorithm, named *SPD*, outperforms the current state-of-the-art FPGA placement flow *T-VPack* + *VPR* with an average reduction of up to 27% in longest path estimate delay and 18% in routed delay. Meanwhile, our combined approach has the same runtime complexity as the existing *VPR* placement algorithm, and both the runtime and area increase are minimal.

2. INITIAL ANALYSIS

In the default FPGA architecutre used in this paper, each CLB consists of 4 BLEs and each BLE conists of one 4-input LUT and one FF. During the study of VPR's placement result on this default architecture, we confirmed two observations mentioned in [1].

Table 1. Percentage of Near-Critical Pins

Circuit	*0%*	*5%*	*10%*	*15%*	*20%*
ex5p	0.13%	1.99%	6.54%	15.88%	30.68%
apex4	0.15%	3.04%	10.14%	22.66%	39.62%
misex3	0.11%	1.40%	4.47%	11.83%	22.74%
Tseng	0.28%	1.71%	4.17%	7.09%	9.45%
alu4	0.12%	1.50%	5.53%	14.38%	26.53%
dsip	0.05%	0.17%	0.98%	2.29%	4.83%
seq	0.09%	0.73%	3.64%	9.69%	19.57%
diffeq	0.17%	1.02%	3.00%	6.35%	10.96%
apex2	0.11%	0.84%	5.93%	15.29%	27.36%
s298	0.23%	3.60%	10.55%	19.43%	30.03%
des	0.05%	0.24%	0.77%	1.93%	5.85%
bigkey	0.04%	0.15%	0.27%	0.38%	1.17%
spla	0.06%	1.17%	4.08%	10.85%	20.19%
elliptic	0.07%	1.03%	3.81%	8.01%	12.75%
ex1010	0.04%	0.78%	2.54%	7.23%	17.70%
pdc	0.04%	0.46%	2.16%	5.98%	12.83%
frisc	0.10%	0.76%	2.21%	4.67%	8.24%
s38584.1	0.04%	0.36%	1.01%	2.05%	3.24%
s38417	0.03%	0.44%	1.00%	1.99%	4.94%
clma	0.02%	0.15%	0.51%	1.88%	4.96%
Average	**0.10%**	**1.08%**	**3.67%**	**8.49%**	**15.68%**

First, the number of critical/near-critical pins is relatively small. Assuming the longest path delay is T, a pin t is critical if $slack(t) = 0$; t is $x\%$ critical if $slack(t)/T \leq x\%$. From Table 1**,** we can see that on average the percentage of critical pins is 0.10%, the percentage of 5% critical pins is 1.1%, the percentage of 10% critical pins is 3.7%, the percentage of 15% critical pins is 8.5%, and the percentage of 20% critical pins is 15.7%. It seems possible to perform a very small number of post-placement duplications to speed up the circuit by 5~10%. However, it may involve many nodes to achieve more than 10~15% speedup.

Table 2. Detour Ratio

Circuit	avg *dr(p)*	min *dr(p)*	max *dr(p)*
ex5p	3.31	2.00	10.14
apex4	3.07	2.00	5.43
misex3	3.52	1.71	10.83
Tseng	2.14	1.50	2.90
alu4	3.50	2.03	10.29
dsip	1.00	1.00	1.00
seq	5.75	2.12	12.00
diffeq	3.79	3.50	4.09
apex2	4.30	1.68	9.86
s298	6.64	4.73	7.92
des	4.36	1.48	19.00
bigkey	1.01	1.00	1.04
spla	5.34	1.86	14.86
elliptic	4.22	2.10	10.25
ex1010	6.08	2.64	21.00
pdc	3.53	1.85	15.50
frisc	4.10	2.93	5.50
s38584.1	3.43	1.25	8.25
s38417	4.89	2.33	10.00
clma	15.92	10.79	18.36
Average	**4.49**	**2.53**	**9.91**

Second, the critical paths are highly non-monotone. For a path p consisting of m nodes, $v_1, v_2, ..., v_m$, v_1 is the starting point and v_m is the ending point. The x coordinate of node v_i is $x(v_i)$ and the y coordinate of node v_i is $y(v_i)$. The Manhattan distance between any two nodes v_i and v_j is defined as $dist(v_i,v_j) = |x(v_i) - x(v_j)| + |y(v_i) - y(v_j)|$. For a node v_i, the deviation of v_i with respect to one of its input nodes v_{i-1} and one of its output nodes v_{i+1} is defined as $dev(v_{i-1}, v_i, v_{i+1}) = dist(v_{i-1}, v_i) + dist(v_i, v_{i+1}) - dist(v_{i-1}, v_{i+1})$. The

sub-path v_{i-1}, v_i, v_{i+1} is monotone if $dev(v_{i-1}, v_i, v_{i+1}) = 0$. The Manhattan distance of path p is defined as $dist(p) = \sum_{i=1}^{m-1} dist(v_i, v_{i+1})$. The minimum distance of path p is defined as $min\_dist(p) = dist(v_1, v_m)$. The path p is globally monotone if $dist(p) - min\_dist(p) = 0$. The level of a path p is defined as $level(p) = m$. The *unit_dist* is defined as the distance between two adjacent CLBs. The detour ratio is defined as $dr(p) = dist(p) / max(min\_dist(p), level(p)*unit\_dist)$. $dist(p)$ is the actual Manhattan distance of p; assuming that p_1 and p_m are fixed, $min\_dist(p)$ is the ideal Manhattan distance of p when it is globally monotone; $level(p)*unit\_dist$ is the distance of p when it is placed almost ideally; the detour ratio describes how non-monotone the path p actually is. The reason that we use the maximum of $min\_dist(p)$ and $level(p)*unit\_dist$ to compute the detour ratio is that sometimes node v_1 and v_m can be placed very close or even inside the same CLB. For the measurement of the average/minimum/maximum detour ratios in Table 2, we consider all the 5% critical paths from PI/FFs to PO/FFs. Both the average detour ratio of 4.49 and the minimum ratio of 2.53 in Table 2 show that the near-critical paths are far from monotone.

3. SIMULTANEOUS TIMING-DRIVEN PLACEMENT AND DUPLICATION

3.1 Algorithm Overview

Our algorithm uses a simulated annealing-based optimization engine [2][10] to minimize a weighted function of bounding-box wirelength and timing (weighted edge delays). At the end of each temperature, we perform logic duplication, legalization and redundancy removal iteratively. To enable the optimization of non-monotone paths from a global perspective, we introduce the notion of feasible region and super feasible region. To handle the complex constraints in commercial FPGA architectures, we introduce a constrained gain graph and perform optimal incremental legalization. To control the runtime of the duplication procedure, we limit the number of duplications for each temperature to a small number. Through experiments we find that good results can be achieved in a short runtime when this limit is logarithmic to the circuit size. In order to merge duplicated copies of the same node, we use a duplication graph representation and propose a heuristic to solve a global redundancy removal problem. For net weighting, we implement a path counting-based net weighting scheme.

In this section, we will describe the key components of our *SPD* algorithm: logic duplication, incremental legalization under complex constraints, duplication graph and redundancy removal, and path counting-based net weighting scheme.

3.2 Logic Duplication

Our logic duplication algorithm can be performed either after a full placement or during the placement at the end of each iteration (e.g. in a simulated annealing-based placer or a quadratic programming-based placer). As shown in Figure 4, a timing analysis is first performed on the current placement. Then we iteratively select a candidate node on the critical paths, duplicate or move it to a new destination location/CLB, redistribute fanouts, and immediately legalize the destination CLB to resolve any possible physical constraints conflicts. If the delay after legalization increases, both the legalization and the duplication operations will be undone. The iteration stops until there are no more candidates or the limit on the number of duplications is reached.

```
Duplication() {
        Timing Analysis
        While (!done) {
                Candidate selection
                Destination selection
                Node duplication
                Fanout partitioning
                Incremental legalization
                Undo when timing gets worse
        }
}
```

Figure 4. Overview of the Duplication Algorithm

3.2.1 Criticality-Driven Candidate Selection

Initially we put all the near-critical PO pins in a heap sorted by the slack, and then we iteratively select the most critical pin t from the heap to perform the speedup operation. The candidate node, the source node s, is duplicated and moved to a new location so as to straighten all the near-critical paths flowing through edge *(s, t)*. When two sink pins have the same criticality, we use the deviation of their source nodes to break ties. After a source node s is duplicated and legalized, we scan the input pins of the clone and put the remaining timing-critical pins into the heap.

3.2.2 Feasible Region and Super Feasible Region

After a candidate node s is chosen, we find a destination location/CLB for it in order to minimize the critical path delay. We assume node s has k critical input nodes i_1, i_2 through i_k.

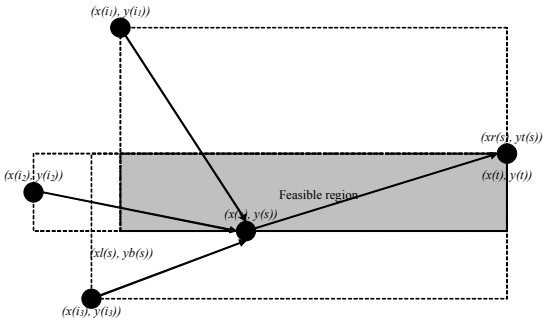

Figure 5. Example of Feasible Region

First we define a feasible region $FR(\{i_j\}, s, t)$ for node s with respect to one of its input nodes i_j and t, in which the clone can be freely placed without increasing the deviation of source node s with respect to i_j and t. Such a feasible region is simply the minimum bounding box enclosing i_j, s and t.

Next we define a feasible region $FR(\{i_1, i_2, ..., i_k\}, s, t) = ((xl(s), yb(s)), (xr(s), yt(s)))$, in which the clone can be freely placed without increasing the deviation of source node s with respect to any of its critical input nodes and output node t. Such a feasible region is the intersection of all $FR(\{i_j\}, s, t)$. If we search for the destination location inside of this feasible region, all the critical paths passing through edge *(s, t)* will be straightened.

Now we describe how to calculate *xl(s)* and *xr(s)* efficiently. We first compute the minimum *x* coordinate of all the critical input nodes of *s* is *min_x(s)*, and so on for *max_x(s)*, *min_y(s)* and *max_y(s)*. When *x(t)* is smaller than *min_x(s)*, *xl(s) = min(x(s), x(t)*) and *xr(s) = max(min_x(s),x(s))*; when *x(t)* falls between *min_x(s)* and *max_x(s)*, *xl(s) = min(x(s), x(t))* and *xr(s) = max(x(s), x(t))*; otherwise, *xl(s) = min(max_x(s),x(s))* and *xr(s) = max(x(s), x(t))*. *yb(s)* and *yt(s)* can be computed similarly. Figure 5 is an example describing the computation of a feasible region. For the largest circuit in our benchmark set, *clma*, the average size of the feasible region is around 5% of the total placement area.

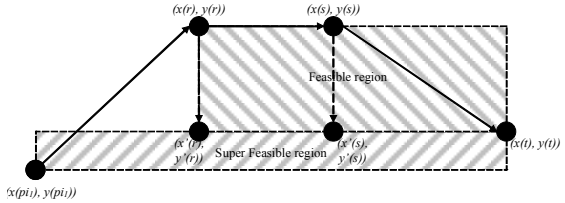

Figure 6. Example of Super Feasible Region

One drawback of [1] is that it cannot handle paths that are globally non-monotone but locally monotone. As shown in Figure 6, the path $pi_1 \rightarrow r \rightarrow s \rightarrow t$ is non-monotone, however, both sub path $pi_1 \rightarrow r \rightarrow s$ and $r \rightarrow s \rightarrow t$ are monotonous. When the delay model is linear to the Manhattan distance, such paths cannot be straightened by the approach in [1], nor by using feasible region alone. To help resolve such global non-monotone problem, we define a notion called super feasible region. A fanin cone C_v rooted at *v* is a connected sub-network which consists of only *v* and its predecessors; a critical fanin cone C_{crit_v} rooted at *v* is a connected sub-network which consists of only *v* and its timing-critical predecessors. For the primary input set of C_{crit_s}, we assume there are *l* critical primary inputs $pi_1, pi_2, ..., pi_l$, the super feasible region is defined as $FR(\{pi_1, pi_2, ..., pi_l\}, s, t)$. During the computation of *min_x(s)*, *max_x(s)*, *min_y(s)* and *max_y(s)*, we use the *x* and *y* coordinates of primary input set $pi_1, pi_2, ..., pi_l$, instead of the immediate input set $i_1, i_2, ..., i_k$. With the introduction of the super feasible region, we give priority to locations inside both regular and super feasible regions. Hence both nodes *s* and *r* will be moved as illustrated in Figure 6, and the whole path will be straightened perfectly.

3.2.3 Destination Selection within the Feasible Region

We iterate through each location inside the feasible region and choose the destination location *(x, y)* such that *Δcost(x,y)* is minimal. The *Δcost(x, y)* function at a location *(x, y)* is defined as $\Delta cost(x, y) = -\alpha * \Delta slack(t) + \beta * overflow\_cost(x, y) + \gamma * g\_path\_cost(x, y)$. α, β and γ are predefined constants. The first term *Δslack(t)* describes the timing improvement. *Δslack(t)* is defined as the increase in slack at sink pin *t* when the clone is placed at location *(x, y)*,. The second term *overflow_cost(x, y)* depicts the legality of the placement or the difficulty of the legalization. *overflow_cost(x, y)* is 0 when the destination location can accommodate the clone, otherwise is the difference between the actual usage and capacity. Priority is given to locations that can accommodate the copy of *s* without violating any physical constraints. The third term *g_path_cost(x, y)* characterizes the violation of the global monotonicity. *g_path_cost(x, y)* is 0 when the destination location is inside the super feasible region, otherwise is the minimum distance from *(x, y)* to the super feasible region.

3.2.4 Timing-Driven Fanout Partitioning

After a clone node is placed at a destination location, we perform timing-driven fanout partitioning to redistribute the fanouts to their corresponding inputs. For each fanout node *t*, we assign it to a copy of the source node such that the arrival time at *t* is minimal. This is similar to the approach described in [1].

3.3 Optimal Incremental Legalization under Complex Constraints

We perform legalization immediately after each duplication operation. If the delay on the most critical path increases, we will undo both the legalization and the duplication.

First, we describe a ripple-move-based legalization approach used in [6]. For each ripple move, we select a source location *S* with overflow and a destination location *T* with extra capacity, and find a maximum gain monotone path from *S* to *T* along which a sequence of cells are moved. To determine the maximum gain path and the cells to be moved, a global analysis based on the *gains* of individual cells is performed. Given the source *S* and the destination *T*, each cell can only be moved in at most two directions. The gain value associated with each cell move is the reduction in the cost function, and the gain value associated with each location and direction is the maximum gain value among all the cells moving in that direction. Then we can construct a *gain graph* in which each vertex corresponds to a location inside the rectangular region determined by *S* and *T*, and each weighted arch represents the maximum gain value in the direction of the arc. Since the *gain graph* is acyclic, the maximum gain path can be found by topological ordering. When the ripple move is performed on this maximum gain path, a cell is allowed to move more than once so that the final gain is equal to or better than the value determined by the maximum gain path. Figure 7 is an example of a *gain graph*.

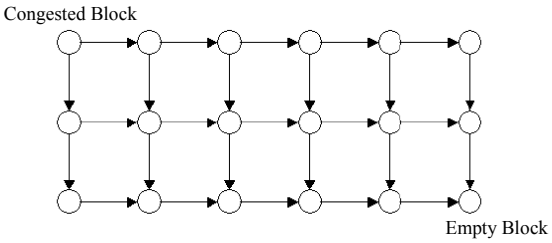

Figure 7. Example of a Gain Graph

However, real commercial architectures have complex constraints at the CLB level in addition to the capacity constraint, such as the input constraint, clock constraint, control signal constraint, etc. Since the gain graph does not consider such complex constraints at all, we introduce a new notion called constrained gain graph in this paper.

To better illustrate this, we consider a simple architecture, in which each CLB contains 2 BLEs and each CLB has 6 inputs. Assume the source CLB at location *(x(s), y(s))* has 3 BLEs, and

sink location at *(x(t), y(t))* has only 1 BLE. The incremental legalization engine tries to find a monotone path from *(x(s), y(s))* to *(x(t), y(t))* to maximize the reduction in certain cost functions. Also it needs to make sure that each involved CLB will obey the input constraint after the legalization. If we assume that $x(t) > x(s)$ and $y(t) > y(s)$, then each move along the monotone path is in the direction of either north or east. If we consider any internal CLB *c* inside the rectangle *((x(s), y(s)), (x(t), y(t)))*, there maybe at most two incoming candidate nodes from the west, two incoming candidate nodes from the south and two outgoing candidate nodes. Thus we build a 4 x 2 bipartite graph for *c*, two of the left side vertices are the incoming candidate BLEs from the west, two of the left side vertices are from the south, and both two right side vertices are the outgoing candidate BLEs inside *c*. After we move in an incoming BLE b_i and move out a BLE b_j, if CLB *c* is still in a legal configuration, we draw an edge from vertex *i* to vertex *j*, and the cost (weight) of the edge is 0.

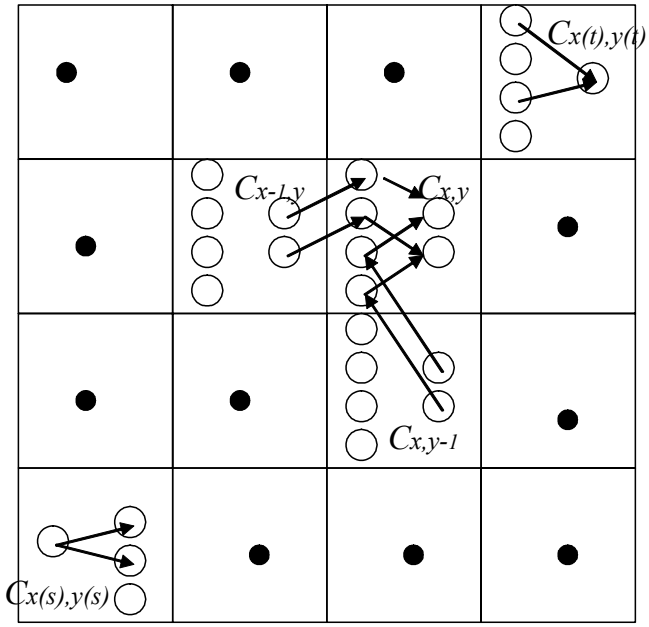

Figure 8. Construction of a Constrained Gain Graph

Figure 8 is a simple example of a constrained gain graph construction. For the purpose of illustration, we only draw the full connection of CLB $C_{x,y}$. All the edges within $C_{x,y}$ describe the input constraints on the current CLB $C_{x,y}$, and they all have a cost of 0; each vertex *i* on the left hand side is connected to its corresponding node before movement, and the weight of the edge is the reduction in the cost function when *i* is moved from $C_{x-1,y}$ or $C_{x,y-1}$ to $C_{x,y}$. Also we create one pseudo source node and one pseudo sink node.

Once the constrained gain graph is constructed, the constrained legalization problem becomes a longest-path problem, which can be easily solved with a complexity of *O(n)*. Our algorithm is optimal under any CLB level constraints.

Our algorithm is optimal for certain cost functions such as bounding box wirelength, weighted source-sink distance, etc. However, the optimality of the maximum gain path does not hold for a general cost function. For example, the timing cost is the summation of weighted delays over all the edges, and the edge delay is determined by the locations of both the source and sink pin. If there is an edge between two cells on the maximum monotone gain path, then the timing cost reduction pre-computed for the sink node would be inaccurate since the source node is moved as well. As a result, the maximum gain path is not the *"real"* maximum gain path for timing optimization under a general timing model. However, if the delay model is linear to the Manhattan distance only, the maximum gain path obtained from the ripple move is still optimal. In this paper we minimize the criticality weighted source-sink Manhattan distance during the legalization.

3.4 Duplication Graph and Redundancy Removal

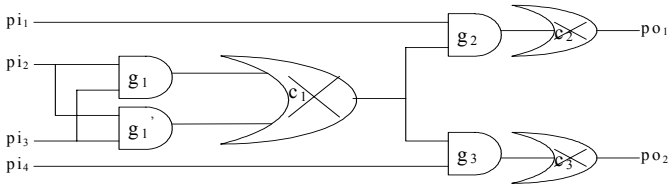

Figure 9. Illustration of a Duplication Graph

Since we perform duplication at the end of each temperature, we may use up the device capacity pretty quickly if redundancies are not removed. Before the logic duplication takes place at the end of each temperature, we need to merge duplicated copies to reduce area while maintaining the circuit performance. This is an essential step in our algorithm.

In this paper, we introduce a data structure called the duplication graph, which is the original netlist with two modifications. First, to keep track of all the copies of the same node, we introduce the notion of *choice node*, whose fanin nodes are all logically equivalent. Second, for each choice node *c*, we introduce a new net *e*. Assume *c* has *k* fanin nodes $g_1, g_2, \ldots, g_k$, and each of the fanin node has an output net $e_1, e_2, \ldots, e_k$. The source pin of net *e* is choice node *c*, and sink pins of net *e* include all the sink pins of $e_1, e_2, \ldots, e_k$. Figure 9 is an illustration of the duplication graph. Under choice node c_1, g_1' is a copy of g_1 and they are logically equivalent. Choice node c_1 drives two gates g_2 and g_3, which fanout to primary outputs po_1 and po_2 respectively. During the logic duplication step, all the duplicated copies are added incrementally to the duplication graph.

In a duplication graph $N = (C, V, E)$, each node c $\in C$ represents a choice node, each gate $v \in V$ represents a logic gate, and each directed edge $e = (c, v) \in E$ represents a wire connecting the output of choice node *c* to one input of a logic gate *v*. Each choice node *c* is a set in which each gate $g \in c$ is a logic gate with equal functionalities. For each directed edge $e = (c, v)$, the arrival time $arr\_t(v)$ at the sink pin is $min(arr\_t(g) + delay(g, v))$ for every $g \in c$.

We formulate a global redundancy removal problem under timing constraints. Under the given timing constraints, $slack(e) \geq 0$ for every edge $e \in E$. We want to find a maximum set *S*, remove every $v \in S$ and every edge $e = (c, v)$ from *N*, such that in the new duplication graph $N' = (C, V', E')$, $slack(e') \geq 0$ for every edge $e' \in E'$ and $c \neq \varnothing$ for every choice node $c \in C$.

We also formulate a local redundancy removal problem under timing constraints. For a choice node c, we assume c has m fanin gates g_1 through g_m, and n fanout gates v_1 through v_n. Under the timing constraints, each fanout gate v_i has a required arrival time $req\_t(v_i)$. We want to find a maximum subset S of c and remove every $g \in S$ from c, such that $arr\_t(v_i) \leq req\_t(v_i)$ for all the fanout gates of c. We build a m by n matrix, and define the value $matrix(i, j)$ at row i column j as the following: if the arrival time of fanout node v_j, $arr\_t(v_j) \leq req\_t(v_j)$ when v_j is driven by g_i, $matrix(i, j) = 1$; otherwise $matrix(i, j) = 0$. To solve the local redundancy removal problem, we need to select a minimum number of rows such that every column contains at least a 1. This is a unate covering or minimum set covering problem, which is NP-complete. As a result, both the local and global removal problems are NP-complete. Since we limit m to a small constant (e.g., 5) during the logic duplication, we solve the local redundancy removal problem optimally using the reduction techniques together with a branch and bound algorithm.

We propose a heuristic to solve the global redundancy removal problem by solving the local redundancy removal problem in a reverse topological order. During the traversal of the duplication graph from PO to PI, we optimally perform local redundancy removal for each choice node with multiple fanins. After a local redundancy problem is solved, we perform duplication removal and fanout partitioning together. Then we propagate the remaining time during the incremental timing update. We do not need to perform a full timing analysis during the redundancy removal process.

3.5 Path Counting-Based Net Weighting

The net-based timing-driven placers (e.g. [10]) convert timing information into net weight and optimize a weighted function of all nets. The basic idea of net weighting is to assign higher weights to timing critical nets and lower weights to non-critical nets. The net weighting scheme is both efficient and flexible enough to handle complex constraints, but most existing methods do not take into account the path information.

In this paper we implement a novel net weighting scheme [7], which accurately counts all paths (critical and non-critical) for certain types of discount functions such as $D(x, y) = a^{-x/y}$. This scheme considers path sharing, and assigns a higher weight to edges shared by two or more critical paths. For more details about path counting, please refer to [7].

4. COMPLEXITY ANALYSIS

The runtime of our *SPD* algorithm consists of two parts: placement engine and duplication/legalization engine. The complexity of our placement engine is exactly the same as *VPR*'s, which is $O(n^{4/3})$. Now we analyze the complexity of the duplication/legalization engine. For each source node, the complexity of the feasible region computation is $O(K)$. The maximum size of the feasible region is the size of the device, which is $O(n)$. For each location in the feasible region, we need to recalculate the edge delay for all input pins of node s and the sink pin t, and that is an $O(K)$ operation. Since the size of the feasible region is worst case $O(n)$, the complexity of finding the optimal destination location is $O(n)$. For legalization, assume the distance between source location and destination location is dx and dy respectively. The complexity for constructing the gain graph is $O(dx*dy*N)$, and the complexity for the maximum gain path algorithm is $O(dx*dy)$. Since dx is bounded by the width of the device, dy is bounded by the height of the device, the legalization algorithm is a worst case $O(n)$ operation. Also we perform static timing analysis during the duplication, which is an $O(n)$ operation as well. Since we limit the number of duplications to a small number (logarithmic to the circuit size) and the number of annealing iterations to another constant (~100), the overall duplication/legalization has a complexity of $O(nlogn)$. As a result, the overall *SPD* algorithm has a runtime complexity of $O(n^{4/3})$.

5. EXPERIMENTAL RESULTS

We implemented our *SPD* algorithm under the framework of our previous work *SCPlace* [3]. For the purpose of comparison, we downloaded the *VPR* 4.3 source code, architecture file and the complete set of 20 MCNC benchmark circuits used by *VPR* from [14]. We modified the architecture file to specify the number of BLEs contained in a single CLB. We compare all of the 20 MCNC circuits with the commonly used academic FPGA design flow [9]. We first run the *script.algebraic* in SIS [12], followed by *Flowmap* [4]. Then we run *T-VPack* [9] to generate an initial clustering solution. This initial clustering is then given to both *VPR* and *SPD* to perform placement. The default architecture we use assumes that each CLB contains 4 LUTs, and each LUT has 4 inputs. Our *SPD* algorithm has several different modes: *SPD-0*, *SPD-1* and *SPD-m*. *SPD-0* has zero duplication, and it is essentially the same as *VPR*; *SPD-1* performs logic duplication once after the placement is done; *SPD-m* performs simultaneous logic duplication and placement optimization. *SPD* w/ path counting utilizes the path counting-based net weighting scheme.

5.1 Post-Placement Duplication Timing Result

In Table 3, we show the impact of post-layout logic duplication on timing on the default architecture. Column 3 is the result of *SPD-0*, which is our implementation of *VPR* without any logic duplication. The result of *SPD*-0 is similar to that of *VPR*. In column 4, we perform duplication/legalization only once after detailed placement, and we achieve on average around 7% of timing improvement on the default architecture.

In Table 4, we illustrate the performance of *SPD-1* on a set of different architectures. When the CLB size is 1, the timing improvement obtained from duplication is 5%. When the size of the CLB (N) increases from 2 to 10, the timing improvement remains in a narrow range between 7 to 9%. The result shows that when the CLB size is greater than one, there is more room for duplication since the delay between BLEs within the same CLB is normally smaller than the average delay between different CLBs.

Table 3. Timing Result of *SPD-1*

Circuit	*VPR*	*SPD-0*	*SPD-1*	%
ex5p	50.45	51.66	47.76	8.2%
apex4	47.44	48.30	46.40	4.1%
misex3	51.04	48.94	45.92	6.6%
Tseng	38.85	38.55	34.80	10.8%
alu4	53.16	54.85	52.45	4.6%
dsip	38.32	38.92	38.92	0.0%
seq	51.26	53.29	49.44	7.8%
diffeq	47.73	45.26	40.93	10.6%
apex2	56.36	58.75	55.05	6.7%
s298	87.36	82.70	75.26	9.9%
des	83.88	81.71	73.60	11.0%
bigkey	41.37	41.51	40.74	1.9%
spla	72.47	72.09	66.45	8.5%
elliptic	71.07	64.48	62.32	3.5%
ex1010	97.88	95.24	87.06	9.4%
pdc	113.15	95.89	87.51	9.6%
frisc	81.39	83.81	76.02	10.3%
s38584.1	64.37	53.98	52.16	3.5%
s38417	76.63	76.84	70.18	9.5%
clma	137.20	136.56	123.93	10.2%
Average				**7.32%**

Table 4. Performance of *SPD-1* across Different Architectures

Circuit	*CLB=1*	*CLB=2*	*CLB=4*	*CLB=8*	*CLB=10*
ex5p	10.2%	5.1%	8.2%	4.1%	10.8%
apex4	5.3%	5.5%	4.1%	9.4%	7.6%
misex3	4.6%	5.6%	6.6%	7.9%	8.7%
Tseng	1.2%	5.9%	10.8%	3.9%	5.9%
alu4	2.0%	8.3%	4.6%	10.3%	6.7%
dsip	3.8%	4.4%	0.0%	5.7%	3.8%
seq	7.4%	8.3%	7.8%	6.8%	7.8%
diffeq	2.1%	12.3%	10.6%	9.6%	10.3%
apex2	2.1%	6.9%	6.7%	7.8%	8.9%
s298	0.0%	8.6%	9.9%	10.1%	8.7%
des	9.6%	6.2%	11.0%	7.4%	10.2%
bigkey	10.2%	1.2%	1.9%	3.5%	0.2%
spla	6.7%	7.7%	8.5%	11.2%	7.4%
elliptic	10.6%	6.7%	3.5%	11.2%	14.9%
ex1010	10.0%	8.5%	9.4%	10.3%	10.2%
pdc	5.9%	8.5%	9.6%	14.6%	9.5%
frisc	3.9%	7.5%	10.3%	8.9%	12.0%
s38584.1	2.3%	2.5%	3.5%	6.9%	0.8%
s38417	2.3%	11.0%	9.5%	11.3%	10.6%
clma	6.1%	9.3%	10.2%	10.3%	11.1%
Average	**5.32%**	**7.01%**	**7.32%**	**8.56%**	**8.30%**

5.2 Simultaneous Placement and Logic Duplication Timing Result

In Table 5, we compare *SPD* with *VPR* and analyze the impact of path counting [7] on timing. If we use path counting-based net weighting scheme in *SPD-0*, we can outperform *VPR* by 14% (column 4); if we perform duplication only in *SPD-m*, we can outperform *VPR* by 19% (column 6); if we integrate the path counting-based net weighting scheme with the duplication optimization, *SPD-m* w/ path counting significantly outperforms the original *VPR* result by 26%.

Table 5. Timing Result of *SPD-m*

Circuit	*VPR*	*SPD-0* w/ path counting	%	*SPD-m*	%	*SPD-m* w/ path counting	%
ex5p	50.45	44.95	12.24%	46.80	7.80%	42.31	19.24%
apex4	47.44	44.71	6.12%	43.86	8.16%	40.2	18.01%
misex3	51.04	44.15	15.61%	42.14	21.12%	37.85	34.85%
tseng	38.85	36.43	6.65%	28.71	35.32%	29.29	32.64%
alu4	53.16	45.46	16.95%	43.56	22.04%	43.44	22.38%
dsip	38.32	40.96	-6.45%	41.37	-7.37%	33.35	14.90%
seq	51.26	46.56	10.11%	43.56	17.68%	42.4	20.90%
diffeq	47.73	38.76	23.15%	36.53	30.66%	38.57	23.75%
apex2	56.36	50.97	10.58%	53.58	5.19%	45.32	24.36%
s298	87.36	90.76	-3.74%	80.56	8.44%	88.4	-1.18%
des	83.88	67.15	24.91%	71.48	17.35%	63.29	32.53%
bigkey	41.37	40.95	1.03%	38.18	8.36%	36.46	13.47%
spla	72.47	63.12	14.81%	63.72	13.73%	65.64	10.41%
elliptic	71.07	55.72	27.54%	59.89	18.67%	52.27	35.97%
ex1010	97.88	79.10	23.75%	87.82	11.46%	71.59	36.72%
pdc	113.15	76.95	47.04%	82.87	36.54%	69.68	62.39%
frisc	81.39	92.53	-12.0%	75.51	7.79%	79.33	2.60%
s38584.1	64.37	46.66	37.96%	46.72	37.78%	48.82	31.85%
s38417	76.63	70.15	9.24%	53.29	43.80%	55.2	38.82%
clma	137.20	116.8	17.52%	106.92	28.32%	99.39	38.04%
Average			**14.15%**		**18.64%**		**25.63%**

Table 6. Performance of *SPD-m* across Different Architectures

Circuit	*CLB=1*	*CLB=2*	*CLB=4*	*CLB=8*	*CLB=10*
ex5p	19.64%	6.72%	19.24%	22.22%	24.09%
apex4	2.45%	15.07%	18.01%	24.13%	30.05%
misex3	6.41%	6.30%	34.85%	21.63%	25.28%
Tseng	10.11%	12.27%	32.64%	7.65%	6.05%
alu4	18.12%	20.10%	22.38%	8.32%	31.23%
dsip	15.68%	10.34%	14.90%	-5.64%	-13.85%
seq	20.43%	15.60%	20.90%	19.88%	23.47%
diffeq	6.04%	16.38%	23.75%	44.52%	23.95%
apex2	16.83%	11.31%	24.36%	22.74%	30.26%
s298	20.12%	4.79%	-1.18%	10.46%	35.06%
des	15.82%	18.90%	32.53%	29.15%	9.89%
bigkey	34.94%	13.25%	13.47%	31.80%	-1.07%
spla	21.73%	39.47%	10.41%	26.21%	35.75%
elliptic	59.16%	30.58%	35.97%	21.94%	5.94%
ex1010	18.24%	23.97%	36.72%	43.87%	47.13%
pdc	18.71%	38.11%	62.39%	52.40%	51.83%
frisc	20.27%	17.67%	2.60%	1.40%	23.67%
s38584.1	1.73%	24.83%	31.85%	34.81%	39.07%
s38417	12.44%	55.93%	38.82%	62.79%	32.72%
clma	14.33%	33.31%	38.04%	55.09%	74.54%
Average	**17.66%**	**20.75%**	**25.63%**	**26.77%**	**26.75%**

In Table 6, we illustrate the performance of *SPD-m* w/ path counting on a set of different architectures. When the CLB size is 1, the performance gap between *SPD-m* and *T-Vpack + VPR* is 18%. When the size of the CLB (*N*) increases from 2 to 10, the timing gap between *SPD-m* and *T-Vpack + VPR* gradually increases from 21 to 27%. The result shows that even when the CLB size is relatively small (1 or 2), integrating duplication with placement has a great impact on circuit performance.

Table 7. Timing Comparison between *SPD* and *SCPlace*

Circuit	*SPD-m*	*SPD-m* w/ path counting	*SCPlace*	*SCPlace* w/ path counting	*SPD-m* + *SCPlace* w/ path counting
ex5p	7.80%	19.24%	18.14%	23.80%	21.76%
apex4	8.16%	18.01%	1.04%	14.49%	30.23%
misex3	21.12%	34.85%	24.49%	32.47%	36.02%
tseng	35.32%	32.64%	18.55%	10.65%	21.99%
alu4	22.04%	22.38%	13.46%	25.07%	23.34%
dsip	-7.37%	14.90%	11.73%	-4.49%	-5.64%
seq	17.68%	20.90%	15.29%	19.51%	27.16%
diffeq	30.66%	23.75%	27.30%	16.01%	39.33%
apex2	5.19%	24.36%	8.07%	19.34%	29.39%
s298	8.44%	-1.18%	-1.17%	7.88%	16.08%
des	17.35%	32.53%	9.14%	28.18%	37.07%
bigkey	8.36%	13.47%	-0.47%	0.03%	31.80%
spla	13.73%	10.41%	8.95%	24.35%	37.20%
elliptic	18.67%	35.97%	-7.63%	46.58%	19.89%
ex1010	11.46%	36.72%	14.93%	30.81%	49.51%
pdc	36.54%	62.39%	42.45%	67.38%	62.86%
frisc	7.79%	2.60%	5.33%	7.47%	16.30%
s38584	37.78%	31.85%	41.67%	34.71%	30.41%
s38417	43.80%	38.82%	58.17%	53.61%	48.21%
clma	28.32%	38.04%	9.73%	34.52%	47.18%
Average	**18.64%**	**25.63%**	**15.96%**	**24.62%**	**31.01%**

5.3 Timing Comparison with *SCPlace* [3]

Since *SPD* explores different clustering solutions via duplication during the placement, it is natural to compare with our recent work *SCPlace* [3], which performs simultaneous clustering and placement optimization. Without the path counting-based net weighting scheme, *SPD-m* outperforms *SCPlace* by a few percentages; with path counting, both *SPD-m* and *SCPlace* achieve similar improvement of around 26%; when all three techniques are combined, we outperform *T-Vpack + VPR* by 31%.

5.4 Area and Runtime Comparison

Table 8. Area and Runtime Comparison of *SPD*

	SPD-0		*SPD-1*				*SPD-m*			
Circuit	Area	Runtime	Area		Runtime		Area		Runtime	
ex5p	1274	65.344	1274	0.00%	65.31	-0.05%	1311	2.90%	62.81	-3.87%
apex4	1319	65.234	1320	0.08%	65.63	0.60%	1337	1.36%	72.30	10.83%
misex3	1529	86.516	1535	0.39%	86.67	0.18%	1534	0.33%	75.53	-12.70%
tseng	1473	87.516	1476	0.20%	87.00	-0.59%	1481	0.54%	95.08	8.64%
alu4	1630	81.953	1630	0.00%	81.89	-0.08%	1631	0.06%	79.78	-2.65%
dsip	2045	115.828	2045	0.00%	116.75	0.80%	2045	0.00%	116.86	0.89%
seq	2029	134.922	2030	0.05%	137.89	2.20%	2033	0.20%	127.06	-5.82%
diffeq	2036	130.391	2036	0.00%	131.02	0.48%	2039	0.15%	123.03	-5.64%
apex2	2159	149.281	2160	0.05%	152.36	2.06%	2165	0.28%	156.48	4.83%
s298	2558	146.406	2558	0.00%	148.33	1.31%	2559	0.04%	165.83	13.27%
des	2673	203.5	2673	0.00%	203.86	0.18%	2673	0.00%	206.41	1.43%
bigkey	3361	239.516	3361	0.00%	241.52	0.83%	3361	0.00%	242.44	1.22%
spla	3999	371.922	4004	0.13%	373.81	0.51%	4036	0.93%	385.83	3.74%
elliptic	4430	473.188	4476	1.04%	475.92	0.58%	4448	0.41%	484.30	2.35%
ex1010	4740	444.25	4740	0.00%	445.86	0.36%	4743	0.06%	438.41	-1.32%
pdc	5672	664.532	5674	0.04%	666.28	0.26%	5678	0.11%	638.47	-3.92%
frisc	6061	617.265	6061	0.00%	620.28	0.49%	6074	0.21%	665.47	7.81%
s38584.1	7375	819.859	7380	0.07%	822.74	0.35%	7375	0.00%	831.14	1.38%
s38417	8589	993.36	8591	0.02%	996.30	0.30%	8604	0.17%	1105.92	11.33%
clma	13673	2214.188	13674	0.01%	2208.86	-0.24%	13688	0.11%	2415.09	9.07%
Average				**0.10%**		**0.53%**		**0.39%**		**2.04%**

In Table 8, we show the area and runtime overhead of our *SPD* algorithm. Regardless of the number of iterations of logic duplication we perform, the area increase by both *SPD-1* and *SPD-m* are very small, normally less than 1%. When we perform only post-placement logic duplication in *SPD-1*, the runtime increase is negligible; even when we perform multiple iterations of logic duplication in *SPD-m*, the average runtime increase remains very small at 2%. We expected more runtime increase for *SPD-m*, but it is not the case. Our analysis of the annealing process reveals that logic duplication helps the placement reach a local minimum faster, so *SPD-m* uses a smaller number of annealing iterations than does *SPD-0* in general.

5.5 Routed Results

Table 9. Routed Delay and Track Count Comparison

	VPR		*SPD-m*		%	
Circuit	Routed delay	#tracks	Routed delay	#tracks	Routed delay	#tracks
ex5p	54.20	660	46.15	740.00	17.43%	12.12%
apex4	50.86	660	51.72	720.00	-1.65%	9.09%
misex3	52.76	638	44.50	638.00	18.56%	0.00%
tseng	41.08	483	33.92	460.00	21.08%	-4.76%
alu4	57.02	624	47.12	648.00	21.02%	3.85%
dsip	38.80	935	35.06	880.00	10.64%	-5.88%
seq	62.99	832	54.51	806.00	15.57%	-3.13%
diffeq	52.84	550	40.67	575.00	29.93%	4.55%
apex2	67.08	783	52.54	999.00	27.68%	27.59%
s298	90.56	754	82.76	841.00	9.42%	11.54%
des	88.32	960	70.76	1088.00	24.82%	13.33%
bigkey	42.60	495	39.59	660.00	7.61%	33.33%
spla	85.37	1470	64.12	1610.00	33.15%	9.52%
elliptic	75.27	1188	71.45	1332.00	5.35%	12.12%
ex1010	111.63	1292	78.95	1634.00	41.40%	26.47%
pdc	112.19	2255	89.84	2296.00	24.88%	1.82%
frisc	84.06	1596	85.90	1764.00	-2.13%	10.53%
s38584.1	61.39	1504	51.21	1316.00	19.90%	-12.50%
s38417	81.80	1650	64.91	1400.00	26.02%	-15.15%
clma	140.28	2898	125.73	3150.00	11.57%	8.70%
Average					18.11%	7.16%

In Table 9**,** we show the comparison of routed delay and track count between *SPD-m* and *T-Vpack + VPR* for the default architecture. The routed delay improvement is 18% on average but the number of routed tracks is 7% more on average. The routed delay improvement is less than the placement estimation improvement, but this is largely expected. The increase in the number of routed tracks is due to the increase in the number of nodes and nets introduced by logic duplication.

6. CONCLUSIONS

We introduce a novel simultaneous placement and duplication algorithm within our *SCPlace* [3] framework. By integrating novel techniques such as simultaneous logic duplication during placement, feasible region-based global path monotonicity optimization, advanced duplication heuristics, optimal legalization under complex constraints, duplication graph representation, redundancy removal and path counting-based net weighting, our new algorithm *SPD* produces excellent results for timing optimization. When compared with the state-of-the-art

separate FPGA design flow *T-VPack + VPR* across different architectures, our algorithm improves circuit performance by up to 27% with less than 1% increase in area and less than 2% increase in runtime. Although we test our algorithm in the context of FPGAs, the basic algorithm applies directly to ASICs and other architectures as well.

7. ACKNOWLEDGMENTS

This research is partially supported by NSF Grant CCF-0096383 and a grant from Magma Design Automation under the California MICRO Program.

8. REFERENCES

[1] G. Beraudo and J. Lillis, "Timing Optimization of PFGA Placements by Logic Replication," *ACM/IEEE Design Automation Conference*, pp. 196-201, 2003.

[2] V. Betz and J. Rose, "VPR: A New Packing, Placement and Routing Tool for FPGA Research," *International Workshop on Field Programmable Logic and Application*, pp. 213-222, 1997.

[3] G. Chen and J. Cong, "Simultaneous Timing Driven Clustering and Placement for FPGAs," *Proc. International Conference on Field Programmable Logic and Its Applications*, Antwerp, Belgium, pp. 158-167, August 2004.

[4] J. Cong and Y. Ding, "FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs," *IEEE Trans. on Computer-Aided Design*, vol. 13, no. 1, pp. 1-12, January 1994.

[5] M. Hrkic, J. Lillis and G. Beraudo, "An Approach to Placement-Coupled Logic Replication," *2004 ACM/IEEE Design Automation Conference*, San Diego, California, pp. 711-716, Jun 2004.

[6] S-W Hur and J. Lillis, "Mongrel: Hybrid Techniques for Standard Cell Placement," *IEEE/ACM International Conference on Computer-Aided Design*, pp 165-170, 2000.

[7] T. Kong, "A Novel Net Weighting Algorithm for Timing-driven Placement," *IEEE/ACM International Conference on Computer-Aided Design*, pp. 172-176, 2002.

[8] J. Lillis, C.-K. Cheng and T.-T. Y. Lin, "Algorithms for Optimal Introduction of Redundant Logic for Timing and Area Optimization," *Proc. IEEE International Symposium on Circuits and Systems*, pp. 196-201, 1996.

[9] A. Marquardt, V. Betz and J. Rose, "Using Cluster-Based Logic Blocks and Timing-Driven Packing to Improve FPGA Speed and Density," *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, CA, pp. 37-46, 1999.

[10] A. Marquardt, V. Betz and J. Rose, ``Timing-Driven Placement for FPGAs," *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, CA, pp. 203 – 213, February 2000.

[11] Neumann, D. Stoffel, H. Hartje, W. Kunz, "Cell Replication and Redundancy Elimination During Placement for Cycle Time Optimization," *IEEE/ACM International Conference on Computer-Aided Design*, pp. 25-30, 1999.

[12] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. Brayton and A. Sangiovanni-Vincentelli, "SIS: A System for Sequential Circuit Synthesis," *Electronics Research Laboratory*, Memorandum No. UCB/ERL M92/41, 1992.

[13] A. Srivastava, R. Kastner and M. Sarrafzadeh, "Timing Driven Gate Duplication: Complexity Issues and Algorithms," *IEEE/ACM International Conference on Computer-Aided Design*, pp. 447-450, 2000.

[14] http://www.eecg.toronton.edu/~vaughn/challenge/challenge.html, "The FPGA Place-and-Route Challenge".