Wireplanning in Logic Synthesis [*]

Wilsin Gosti Amit Narayan[+] Robert K. Brayton Alberto L. Sangiovanni-Vincentelli
Dept. of EECS, University of California, Berkeley, CA 94720
[+] Monterey Design Systems, Sunnyvale, CA 95139
Email: {wilsin, anarayan, brayton, alberto}@eecs.berkeley.edu

Abstract

In this paper, we propose a new logic synthesis methodology to deal with the increasing importance of the interconnect delay in deep-submicron technologies. We first show that conventional logic synthesis techniques can produce circuits which will have long paths even if placed optimally. Then, we characterize the conditions under which this can happen and propose logic synthesis techniques which produce circuits which are "better" for placement. Our proposed approach still separates logic synthesis from physical design.

1 Introduction

Conventional logic synthesis assumes that the delay of a circuit depends only on the delays of the gates in the circuit and mostly ignores the effect of interconnect delay. However, as we move towards smaller geometries, interconnect delay is becoming an increasingly larger fraction of the total delay. In fact, Semiconductor Industrial Alliance's National Technology Roadmap for Semiconductor for 1997 [1] predicts that interconnect delay will start dominating the total gate delay as we move down to 0.15μ technology and below. Another study by Keutzer, et al [5] shows that for 0.25μ technology and below, interconnect delay can contribute anywhere from 50% to 80% of the total delay. Therefore, logic synthesis can no longer afford to ignore the effect of interconnect delay during optimization.

In this paper, we adopt a diametrically opposite approach to that of conventional logic synthesis. We perform logic synthesis to optimize only for interconnect delay, ignoring the effect of gate delays. Our approach is based on the simple observation that if an output o depends on an input i, then the best way to connect i to o is through a path which is monotonic from i to o, that is, there are no diversions in the path from i to o. We first show, by means of an example, that conventional logic synthesis can produce a circuit for which it is impossible to find a placement with no diversions in the input-output paths. Therefore, no matter how good a place & route tool is, it may not be able to produce a circuit which is optimal in terms of interconnect delay.

We define the notion of *illegal* nodes. Intuitively speaking, a node is illegal if it introduces a diversion in the circuit no matter where it is placed. We characterize the condition under which a node is illegal and give a procedure to convert an arbitrary circuit into a circuit which has only legal nodes. We call such a circuit a *legal* circuit. We show that for a legal circuit, there always exists a point placement of the nodes such that every input-output path is monotonic. We also provide a set of logic synthesis transformations which are guaranteed to preserve the "legality" of a circuit.

The proposed approach has the advantage that it still maintains a distinction between the logic synthesis and place & route stages. It does not need to tightly couple synthesis and placement by frequently alternating between the two which can be inefficient and may not converge at all.

2 Previous Work

So far very little work has been done to model the effect of interconnect delay at the logic level. This is mainly due to the fact that at the logic level, very little information is available about the interconnect. Most of these approaches [9, 8, 14] use a rough companion placement to estimate the cost of various logic synthesis operations and make decisions based on this cost. In [13] an iterative approach to combine synthesis and placement is presented. Instead of using a companion placement to guide synthesis, they use actual placement which can be modified incrementally based on the netlist changes. In [15] a heuristic to minimize the layout cost is proposed which doesn't employ a companion placement solution. The method in [15] is based on minimizing the average fanout range and evenly distributing fanouts in the chip. It was shown that the chip delay could be reduced by this approach if all the input pins are located on one side of the chip and all the output pins on the opposite. Like [15], our approach also does not employ a companion placement. We analyze conditions under which a netlist is not "good" for placement given the locations of i/o pins and try to transform it into one which is.

3 Preliminaries

Definition 1 *A logic circuit $\mathcal{L}$ is a 3-tuple $(I, O, \mathcal{F})$. I is a set of primary input pins or simply primary inputs. O is a set of primary output pins or simply primary outputs. Each element of I and O is a binary variable. An element $f_j \in \mathcal{F}$ is a function $f_j : \mathbf{B}^{|I|} \mapsto \mathbf{B}$. Each f_j is called the global function of the primary output o_j.*

A logic circuit is represented by a Boolean network [3]. If n_k is an immediate fanout of n_j in the Boolean network, we write $n_j \rightarrow n_k$. A logic circuit is *pin-assigned* if each primary input i is labeled with a position (x_i, y_i) and each primary output o with position (x_o, y_o). A logic circuit $\mathcal{L}$ is *placed* if every node n of the Boolean network representing $\mathcal{L}$ has a position, i.e. every node n is labeled with (x_n, y_n), and the resulting placement is denoted by $\mathcal{P}_{\mathcal{L}}$. A point placement of $\mathcal{L}$ is a placement of $\mathcal{L}$ where each node is represented as a point. Given a point-placed circuit, a path, $p_{(i,o)}$, from a primary input i to a primary output o is a sequence of connected nodes from i to o, and the length of the path, $d_{(i,o)}$, is the length of all the wires along the path from i to o. The path $p_{(i,o)}$ is called *monotonic* if its length is equal to the

---

[*] This work was supported in part by SRC-98-DC-324.

ICCAD98, San Jose, CA, USA

Manhattan distance from i to o. The placement $\mathcal{P}_\mathcal{L}$ of $\mathcal{L}$ is optimal if there is no other placement of $\mathcal{L}$ whose length of the longest path is shorter than that of $\mathcal{P}_L$.

The coordinate system that we use in the paper assumes that the x-axis goes from left to right and the y-axis goes from top to bottom.

4 Problem Description

Given a logic circuit $\mathcal{L}$, the goal is to find a placed circuit $\mathcal{N}_\mathcal{L}$ such that the interconnect delay of the circuit is minimized. Due to efficiency reasons, we want to maintain the decoupling of the problem into a separate synthesis phase followed by a place & route phase as in the conventional approach. Given a logic circuit $\mathcal{L} = (I, O, \mathcal{F})$, we address the problem of finding a Boolean network $\mathcal{N}_\mathcal{L}$, which when placed optimally, leads to a circuit with minimum interconnect delay. It is up to the placement tool to find the optimal placement for such a network. Intuitively speaking, we are trying to create a circuit for which a "good" placement exists.

We assume that the die is represented by a rectangle R with width w_R and height h_R and the given logic circuit is pin-assigned. We assume that the delay of a path is a linear function of its length. In general, the interconnect delay depends quadratically on the length of the interconnect. However, it can be made linear by buffer insertion and wire sizing, as shown in recent studies by Otten and Brayton [7] and Cong and Pan [4]. A circuit is said to be optimal in terms of interconnect delay if the length of a path from any primary input i to any primary output o is its Manhattan distance (monotonic), i.e.

$$d_{(i,o)} = |x_i - x_o| + |y_i - y_o|$$

This definition is motivated by the pin-to-pin delay model of Kukimoto and Brayton [6]. Under this model, a delay number is assigned for every input-output pair. This model is particularly suited for intellectual property (IP) blocks where the arrival time of the pins are not known in advance. Consequently, any input-output path can end up being a critical path. Therefore, to minimize the delay, we have to minimize the delay for all input-output paths. We call this problem the *IP-based synthesis* problem.

We will also be addressing a slightly different problem called the *slack-based synthesis* problem, where the only difference from the IP-based problem is the objective function. Instead of minimizing the length of the path from any primary input to any primary output, we minimize the longest path of the circuit, i.e.

$$\min\{\max_{(i \in I, o \in O)} d_{(i,o)}\}$$

In this paper, we will mainly focus on the *IP-based synthesis* problem. However, the approach can be modified to address the *slack-based* problem as well. We will very briefly discuss this in Section 7.

5 Approach

To understand the problem better, let us first look at an example where the conventional logic synthesis which considers only gates during optimization may not be able to find a circuit with minimum interconnect delay.

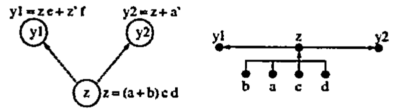

Figure 1: Network $\mathcal{N}_{\min}$ and its optimal placement.

5.1 Logic Synthesis and Interconnect Delay: An Example

Let us consider a minimum literal boolean network $\mathcal{N}_{\min}$ with 10 literals as shown in Figure 1 on the left. Assuming that the pin positions are given, the optimal placement of $\mathcal{N}_{\min}$ is shown in Figure 1 on the right. Pins e and f are not shown and are assumed to be close to y_1. In this solution, there are two longest paths of equal length, i.e. one path from b to y_1 and the other from b to y_2. This circuit is not optimal in terms of both the IP-based and the slack-based synthesis problems because there is a better decomposition of the circuit that produces shorter longest paths. The better decomposed network with 11 literals is shown in Figure 2 together with its optimal placement. Although network $\mathcal{N}_{\min}$ has fewer literals than $\mathcal{N}'$, it has an extra path from b to y_2. Consequently, the placement tool places node z to minimize the longest paths from b to y_1 and y_2. However, as we see in Figure 2, y_2 is independent of b and therefore, b can be removed from the support of y_2. This leads to the network in Figure 2 whose optimal placement has shorter longest path as compared to $\mathcal{N}_{\min}$.

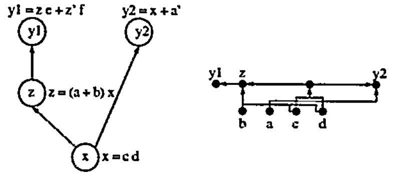

Figure 2: Network $\mathcal{N}'$ and its optimal placement.

Although network $\mathcal{N}'$ is better than $\mathcal{N}_{\min}$ in terms of both IP-based and slack-based synthesis problems, there is yet a better decomposition for the IP-based synthesis problem. In $\mathcal{N}'$, the path from c to y_1 is greater than its Manhattan distance. The same is true for the path from d to y_2. A better decomposed circuit $\mathcal{N}''$ with 11 literals and its optimal placement are shown in Figure 3.

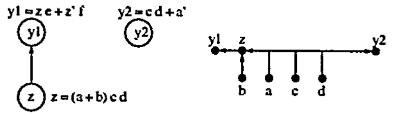

Figure 3: Network $\mathcal{N}''$ and its optimal placement.

From the example above, we see that sometimes the output of a logic synthesis is not "good" for placement, i.e. no matter how we place the nodes, there is at least one path which is longer than its Manhattan distance. In our approach, the aim is to guide logic synthesis such that it produces a circuit which is good for placement. It is up to the placement tool to find the optimal placement for the decomposed circuit in the placement phase.

In this section we define what we mean by a circuit which is "good" for placement and then give a set of transformation rules which can find such a circuit. Our approach can be divided into two broad stages: constraint generation and constraint driven synthesis. In the constraint generation step, we partition the die into regions and identify the types of functions that are allowed to fill them. We define the notion of *illegal* nodes. Intuitively speaking, a node is illegal if it can not be placed somewhere on the die without causing a diversion in the circuit. We show that if a circuit consists of only legal nodes then there exists a point placement of the nodes such that every input-output path is monotonic. We call such a circuit a *legal* circuit. We characterize the condition under which a node is illegal and give a procedure to convert an arbitrary circuit into a legal circuit.

Since nodes have areas, in the constraint driven synthesis step, we synthesize the legal circuit to find another legal circuit with minimum area. We extend the algebraic transformations and don't care minimization such that they operate on legal nodes and produce legal nodes. As in the conventional logic synthesis case, we use the number of factored-form literals as our area estimates since it has been proven to be a good indication of the size of a Boolean network.

5.2 Constraint Generation

Since the length of every path from a primary input to a primary output is restricted to its Manhattan distance (monotonic), there is a well defined region where a Boolean node can be placed. Let us define region formally.

Definition 2 *A region $r = \{x_l, y_t, x_r, y_b\}$, where $x_l \leq x_r$ and $y_t \leq y_b$, is the set of all points in the rectangle bounded at opposite corners by the points (x_l, y_t) and (x_r, y_b). Mathematically, $r = \{(x,y) \mid x_l \leq x \leq x_r \text{ and } y_t \leq y \leq y_b\}$.*

Definition 3 *Given two points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$, the region defined by p_1 and p_2 is region $r_{(p_1,p_2)} = \{\min(x_{p_1}, x_{p_2}), \min(y_{p_1}, y_{p_2}), \max(x_{p_1}, x_{p_2}), \max(y_{p_1}, y_{p_2})\}$.*

With these definitions, we go back to analyze why node z of the Boolean network $\mathcal{N}'$ in Figure 2 is "good" but not x. Because node z fans out to y_1 and its support set is $\{a,b,c,d\}$, z should be placed in the region $r_{(y_1,b)}$, which is r_z in Figure 4, so that the path from any primary input in the support set, i.e. a, b, c, or d, to y_1 is monotonic. For the same example, there is no good region to place node x because there are two conflicting requirements. One requirement says that node x should be placed in region $r_{(y_1,c)}$, which is r_1 in Figure 5, for the path from c to y_1 to be monotonic; while the other says that node x should be placed in region $r_{(y_2,d)}$, which is r_2 in Figure 5, for the path from d to y_2 to be monotonic. As shown in the figure, x can not be placed in both r_1 and r_2. Hence, x is not a desirable factor.

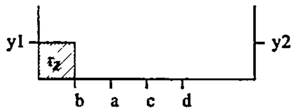

Figure 4: Legal region of node z.

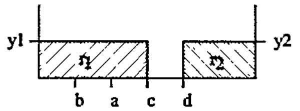

Figure 5: Conflicting legal region requirements for x.

5.2.1 Region Placement Constraints

The example above illustrates that if there is a path from a primary input i to a primary output o, then for the path to be monotonic, all the logic gates along the path should be placed in the region $r_{(i,o)}$. This leads us to first partition the die into rectangles along the pin positions and label each region with functions that can be placed in it. Continuing with our example, the die area associated with $y_1, y_2, a, b, c,$ and d is partitioned into regions $\mathcal{R} = \{r_1, r_2, r_3, r_4, r_5\}$ as shown in Figure 6. Region r_1 is labeled with $\{a,b,c,d\}_{y_1}$ to mean that factors whose supports are a subset of $\{a,b,c,d\}$ and transitively fan out only to y_1 are allowed to be placed in r_1. Region r_3 is labeled with $\{c,d\}_{y_1}$ and $\{a,b\}_{y_2}$ to mean that factors whose supports are a subset of $\{c,d\}$ and transitively fan out only to y_1 or factors whose supports are a subset of $\{a,b\}$ and transitively fans out only to y_2 are allowed to be placed in r_3. Other regions are labeled in a similar fashion. Refering back to Boolean network $\mathcal{N}'$, we see that node z is a "good" node and can be placed in r_1 because its support set is $\{a,b,c,d\}$ and it transitively fans out only to y_1. This matches the label of r_1. Node x is not a "good" node because there is no region whose label contains its support set $\{c,d\}$ and both of its transitive fanouts are y_1 and y_2.

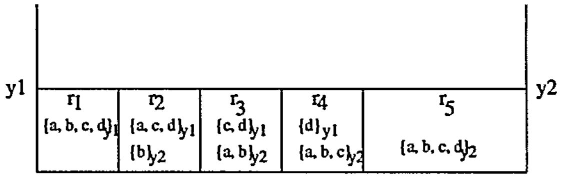

Figure 6: Regions and labels of regions.

Definition 4 *A placement constraint d is a 2-tuple (O^d, σ^d), where $O^d \subseteq O$, and $\sigma^d \subseteq I$. O^d is called the output set and σ^d the support set of d. We also write d as $\{i_1, i_2, \ldots\}_{o_1, o_2, \ldots}$, where $\sigma^d = \{i_1, i_2, \ldots\}$ and $O^d = \{o_1, o_2, \ldots\}$.*

Each region is labeled with a set of placement constraints, e.g. r_1 is labeled with $\{a,b,c,d\}_{y_1}$ and r_3 is labeled with $\{c,d\}_{y_1}$ and $\{a,b\}_{y_2}$ as shown in Figure 6. A placement constraint on a region r is called its *region placement constraint.*

Hence, each region placement constraint $d_r = (O^r, \sigma^r)$ in a region r denotes that Boolean nodes that fan out only to a subset of the primary outputs in O^r and have at most σ^r in their support can be placed in r.

5.2.2 Node Placement Constraints

We see that given a region r, only certain types of nodes can be placed in r and this is captured in its *region placement constraint.* We now define the dual for nodes. Given a node n, it can only be placed in certain regions. For example, node z of Boolean network $\mathcal{N}'$ in Figure 2 can only be placed in region r_1 as shown in Figure 6. Hence, we label each node with a placement constraint and it is called its *node placement constraint.* The node placement constraint of node n denotes the support of n and its transitive primary outputs. For example, the node placement constraint of z of Boolean network $\mathcal{N}'$ is $\{a,b,c,d\}_{y_1}$.

The node placement constraints of nodes of a Boolean network can be easily computed by traversing the Boolean network in a breadth-first manner from the primary inputs to compute the support sets and from the primary outputs to compute the output sets.

5.2.3 Properties of Placement Constraints on Boolean Networks

In this section, we show what "good" nodes mean and having a Boolean Network with only "good" nodes can lead to a monotonic point placement of the network.

Intuitively, a "good" node is one that can be placed in a region. We define such "good" nodes as legal. However, before we can formally define the legality of a node, we need the definition of containment of placement constraints.

Definition 5 *Placement constraint* $d_a = (O^a, \sigma^a)$ *is contained in placement constraint* $d_b = (O^b, \sigma^b)$*, denoted as* $d_a \subseteq d_b$*, if* $O^a \subseteq O^b$ *and* $\sigma^a \subseteq \sigma^b$.

Definition 6 *Boolean node n with node placement constraint* d_n *is* **legal** *with respect to region r with region placement constraints* $\{d_{r_1}, d_{r_2}, \ldots\}$*, denoted as* $n \downarrow r$*, if there exists a j such that* $d_n \subseteq d_{r_j}$.

Definition 6 says that node n is legal with respect to region r if n can be placed in r.

Definition 7 *A Boolean node n is* **legal** *if there is a region r such that* $n \downarrow r$.

Definition 7 says that node n is legal if there is a region r where n can be placed. This definition and Definition 6 are about the legality of a Boolean node. Now given a node, the next definition defines the region in which the node is legal.

Definition 8 *The legal regions of a node n, denoted as* $R(n)$*, is the set of regions* $\mathcal{R} = \{r_1, r_2, \ldots, r_l\}$ *such that for any region* $r_j \in \mathcal{R}$, $n \downarrow r_j$.

For clarity purposes, we denote the legal region of a node n with node placement constraint d_n as $R(d_n)$. We will then assume that given a node placement constraint, the node is implicitly defined.

It can be easily seen that $R(\{i_k\}_{o_l})$ is the region $r_{(i_k, o_l)}$. If we define $R(d_1) \cap R(d_2)$ to be the overlapping region between $R(d_1)$ and $R(d_2)$, then it is easy to see that $R(\{i_1, i_2, \ldots, i_m\}_{o_1, o_2, \ldots, o_n})$ is equal to:

$$\begin{aligned} &R(\{i_1\}_{o_1}) \cap R(\{i_2\}_{o_1}) \cap \cdots \cap R(\{i_m\}_{o_1}) \cap \\ &R(\{i_1\}_{o_2}) \cap R(\{i_2\}_{o_2}) \cap \cdots \cap R(\{i_m\}_{o_2}) \cap \\ &\qquad \cdots \cap \\ &R(\{i_1\}_{o_n}) \cap R(\{i_2\}_{o_n}) \cap \cdots \cap R(\{i_m\}_{o_n}). \end{aligned}$$

This is called the *intersection rule*. For example, as shown in Figure 7, for node z of Boolean network $\mathcal{N}'$,

$$\begin{aligned} R(z) &= R(\{a, b, c, d\}_{y_1}) \\ &= R(\{a\}_{y_1}) \cap R(\{b\}_{y_1}) \cap R(\{c\}_{y_1}) \cap R(\{d\}_{y_1}) \\ &= r_{z_1} \cap r_{z_2} \cap r_{z_3} \cap r_{z_4} \\ &= r_z \end{aligned}$$

Based on Definition 7, the legality of a node n with node placement constraint $d_n = (O^n, \sigma^n)$ can be checked by traversing all regions and check if n is legal for each region. Assuming $|I| > |O|$, the complexity for this algorithm is $O(|I|^2\,|O|)$ because the number of regions is $O(|I|\,|O|)$ and the number of region placement constraints in a region is $O(|I| + |O|)$. A better algorithm would be to check if the legal region of n is empty or not. This can be done by using the intersection rule defined above. The complexity is then $O(|O^n|\,|\sigma^n|)$, which can be much smaller. However, there is a linear algorithm with complexity $O(|O^n| + |\sigma^n|)$ according to the next three lemmas.

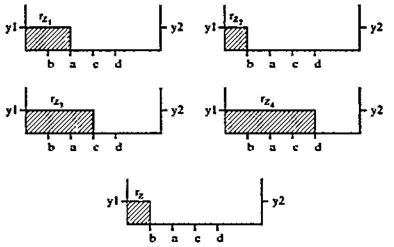

Figure 7: Region intersection for node z of $\mathcal{N}'$.

Lemma 1 below says that nodes that transitively fan out to only one output are always legal.

Lemma 1 *For a node placement constraint* $\{i_1, i_2, \ldots, i_m\}_{o_1, o_2, \ldots, o_n}$ *with* $n = 1$, $R(\{i_1, i_2, \ldots, i_m\}_{o_1, o_2, \ldots, o_n}) \neq 0$.

Proof: For $\{i_1, i_2, \ldots, i_m\}_{o_l}$, the point (x_{o_l}, y_{o_l}) is in $R(\{i_1, i_2, \ldots, i_m\}_{o_l})$. ■

Lemma 2 below enumerates the cases when nodes that transitively fan out to two outputs are legal.

Lemma 2 *For a node placement constraint* $\{i_1, i_2, \ldots, i_m\}_{o_1, o_2, \ldots, o_n}$ *with* $m \geq 2$ *and* $n = 2$, $R(\{i_1, i_2, \ldots, i_m\}_{o_1, o_2, \ldots, o_n}) \neq 0$ *iff*

1. $(\forall i \forall o\ x_i \geq x_o \wedge y_i \geq y_o) \vee (\forall i \forall o\ x_i \geq x_o \wedge y_i \leq y_o) \vee (\forall i \forall o\ x_i \leq x_o \wedge y_i \geq y_o) \vee (\forall i \forall o\ x_i \leq x_o \wedge y_i \leq y_o)$*, or*
2. $R(\{i_1, i_2, \ldots, i_m\}_{o_l})$ *is a point, i.e.* $x_{o_1} = x_{o_2} \wedge \forall i\ y_i = C$*, or* $y_{o_1} = y_{o_2} \wedge \forall i\ x_i = C$*, for some* $C \in \mathcal{N}$.

Proof: If part:

1. Let us assume without loss of generality that $(\forall i \forall o\ x_i \geq x_o \wedge y_i \geq y_o)$, and let $i_{\min} = (\min\{x_i\}, \min\{y_i\})$ and $o_{\max} = (\max\{x_o\}, \max\{y_o\})$, then the legal region is $r_{(i_{\min}, o_{\max})}$ and it is not empty.
2. If the legal region is a point, then it is not empty.

Only if part: Without loss of generality assume that the legal region is not empty and it is not a point, but $x_{i_1} < x_{o_1} < x_{i_2}$, i.e. o_1 is on the top side of the die, then $R(\{i_1, i_2\}_{o_1})$ is a point if both i_1 and i_2 are on the top side as well (Figure 8a); it is a line otherwise (Figure 8b). Since $R(\{i_1, i_2\}_{o_1, o_2}) = R(\{i_1, i_2\}_{o_1}) \wedge R(\{i_1, i_2\}_{o_2})$, it is not empty iff $y_{i_1} = y_{i_2}$ and $x_{o_1} = x_{o_2}$, i.e. o_1 and o_2 are at opposite side (Figure 8c). If we have more than two inputs, then they all have to be either on the top or the bottom side of the chip for the legal region to be non-empty and the legal region has to be a point (Figure 8d). Hence, it is a contradiction. ■

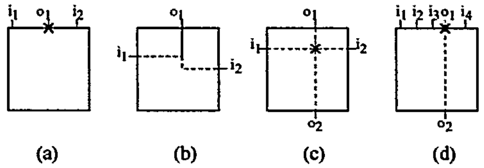

Figure 8: Figure for proof of Lemma 2.

The following lemma says if a node transitively fans out to more than two outputs, then there can only be one case where it is legal.

Lemma 3 *For a node placement constraint* $\{i_1, i_2, \ldots, i_m\}_{o_1, o_2, \ldots, o_n}$ *with* $m \geq 2$*, and* $n > 2$*,* $R(\{i_1, i_2, \ldots, i_m\}_{o_1, o_2, \ldots, o_n}) \neq 0$ *iff* $(\forall i \forall o\, x_i \geq x_o \wedge y_i \geq y_o) \vee (\forall i \forall o\, x_i \geq x_o \wedge y_i \leq y_o) \vee (\forall i \forall o\, x_i \leq x_o \wedge y_i \geq y_o) \vee (\forall i \forall o\, x_i \leq x_o \wedge y_i \leq y_o)$.

Proof: The proof is similar to the proof of Lemma 2.

If part: This is the same as the first case of the if part of Lemma 2 proof.

Only if part: Without loss of generality assume that the legal region is not empty but $x_{i_1} < x_{o_1} < x_{i_2}$, i.e. o_1 is on the top side of the die, then $R(\{i_1, i_2\}_{o_1}$ is a point if both i_1 and i_2 are on the top side as well (Figure 8a); it is a line otherwise (Figure 8b). Since $R(\{i_1, i_2\}_{o_1, o_2}) = R(\{i_1, i_2\}_{o_1}) \wedge R(\{i_1, i_2\}_{o_2})$, it is not empty iff $y_{i_1} = y_{i_2}$ and $x_{o_1} = x_{o_2}$, i.e. o_1 and o_2 are at opposite side (Figure 8c). There is no way to add a third output to $\{i_1, i_2\}_{o_1, o_2}$ with a non-empty legal region. Hence, it is a contradiction. ■

By the input-output symmetric nature of legal regions, the above three lemmas apply with the role of *m* and *n* interchanged.

Let the condition $(\forall i \forall o\, x_i \geq x_o \wedge y_i \geq y_o) \vee (\forall i \forall o\, x_i \geq x_o \wedge y_i \leq y_o) \vee (\forall i \forall o\, x_i \leq x_o \wedge y_i \geq y_o) \vee (\forall i \forall o\, x_i \leq x_o \wedge y_i \leq y_o)$ be called the *non-overlapping* condition. Then, with these three lemmas, the legality of a node with node placement constraint $\{i_1, i_2, \ldots, i_m\}_{o_1, o_2, \ldots, o_n}$ can be checked with the following algorithm:

1. If *n* is 1, then the node is legal.
2. If the non-overlapping condition is true, then the node is legal. This can be checked in $O(m+n)$ by first finding the largest and smallest *x* and *y* coordinates of both inputs and outputs and then check for the overlapping condition using these values.
3. If the node placement constraint satisfies Condition 2 of Lemma 2, then it is legal.
4. If none of the above are satisfied, then the node is illegal.

It is obvious that this legality checking algorithm is $O(m+n)$. Hence, it is very efficient.

Corollary 5.1 *There exists a corner point* p_c *of* $R(\{i_1, i_2, \ldots, i_m\}_{o_1, o_2, \ldots, o_n})$ *that is closest in distance to all outputs, and a corner point* p_f *furthest from all outputs. The point* p_c *is called the closest point of the region and* p_f *the furthest point.*

Lemma 4

1. *If* $R(\{i_1, i_2, \ldots, i_m\}_{o_1, o_2, \ldots, o_n}) \cap R(\{i_k\}_{o_1, o_2, \ldots, o_n})$*, where* $i_k \notin \{i_1, i_2, \ldots, i_m\}$*, is not empty, then it contains the closest point of* $R(\{i_1, i_2, \ldots, i_m\}_{o_1, o_2, \ldots, o_n})$*.*
2. *If* $R(\{i_1, i_2, \ldots, i_m\}_{o_1, o_2, \ldots, o_n}) \cap R(\{i_1, i_2, \ldots, i_m\}_{o_k}$*, where* $o_k \notin \{o_1, o_2, \ldots, o_n\}$*, is not empty, then it contains the furthest point of* $R(\{i_1, i_2, \ldots, i_m\}_{o_1, o_2, \ldots, o_n})$*.*

Proof: Assume that $m \geq 2$ and $n > 2$. The proof is similar for other cases.

1. Assume $(\forall i \forall o\, x_i > x_o \wedge y_i > y_o)$ (the proofs of the other cases are the same), then $x_k > x_o \wedge y_k > y_o$. If x_k is greater than the *x*-coordinates of any other input, then $R(\{i_1, i_2, \ldots, i_m\}_{o_1, o_2, \ldots, o_n}) \cap R(\{i_k\}_{o_1, o_2, \ldots, o_n}) = R(\{i_1, i_2, \ldots, i_m\}_{o_1, o_2, \ldots, o_n})$. If x_k is less than the *x*-coordinate of all other inputs, then the vertical line going through i_k partitions $R(\{i_1, i_2, \ldots, i_m\}_{o_1, o_2, \ldots, o_n})$ into two regions and $R(\{i_1, i_2, \ldots, i_m\}_{o_1, o_2, \ldots, o_n}) \cap R(\{i_k\}_{o_1, o_2, \ldots, o_n})$ is the partition that includes the closest point.
2. The proof is similar to case 1.

■

Lemma 4 says that:

1. Adding inputs to a node placement constraint will not change the closest point of its legal region.
2. Adding outputs to a node placement constraint will not change the farthest point of its legal region.

At this point, we have defined what legal nodes are and how to check for legality of nodes. We now put the legal context into Boolean networks and discuss the implication of legality of Boolean network on placement.

Definition 9 *A Boolean network is* **legal** *is every node in the network is* **legal**.

There is a nice property of a legal Boolean network as described by the following theorem.

Theorem 5.1 *Given a legal boolean network, there exists a monotonic point placement for the network.*

Proof:

This is an induction proof. We traverse the Boolean network in a reverse topological order, i.e. a node is visited only after all its fanouts have been visited.

The base case is where we have all primary outputs. Let *o* be an arbitrary primary output, then place *o* at its pin location. For *o*, its pin location is its closest point. The induction hypothesis is that fanouts of a node *n* are placed at their closest points and still maintaining monotonicity, i.e. the distances from their closest points to their primary outputs are their Manhattan distances. we show that *n* can also be placed at its closest point while still maintaining monotonicity.

Let n_f be an arbitrary fanout of *n*. Let c' be the node placement constraint of n_f with all fanins except *n* removed. Also let the node placement constraints of *n* and n_f be *c* and c_f. Then c_f is derived from c' by adding the primary inputs of fanins of n_f other than *n* and *c* is derived from c' by adding the primary outputs of fanouts of *n* other than n_f. We know that $R(c') \neq 0$ because $c' \subseteq c$ and $R(c) \neq 0$ by the assumption that *n* is legal. By applying Lemma 4 for each primary input added to c' to form c_f, $R(c_f)$ includes the closest point of $R(c')$. Since $R(c) \subseteq R(c')$, the distance from the closest point of $R(c)$ to a primary output *o* is the same as the sum of the distance from the closest point of $R(c)$ to the closest point of $R(c_f)$ and the distance from the closest point of $R(c_f)$ and *o*. Hence, the monotonic property is maintained and *n* can be placed at the closest point of $R(c)$.

■

Theorem 5.1 reduces our problem of finding a monotonic point placement of a circuit into the problem of finding a legal Boolean network. The logic synthesis transformations we use to convert an illegal Boolean network into a legal one is called *make_legal*, and it is explained below.

5.2.4 Make_Legal

The *make_legal* operation takes a Boolean network as its input and produces a legal Boolean network. In the effort of producing a legal Boolean network, it attempts to minimize the number of new Boolean nodes created.

The following lemma and corollary guarantee that a Boolean network can always be made legal.

Lemma 5 *If $n \dashrightarrow n_f$, and n is illegal but n_f is legal, then collapsing n into n_f will not make n_f illegal.*

Proof: Collapsing n to n_f does not change the support of n_f, nor does it add any primary output to the transitive fanout of n_f. Therefore, the node placement constraint of n_f does not change and hence n_f stays legal. ■

By the proof of Theorem 5.1, we know that every primary output is legal. Then it is easy to see the following corollary.

Corollary 5.2 *An illegal Boolean network can always be made legal by collapsing all nodes into the primary output nodes.*

Beside collapsing, node duplication can also legalize a node.

Lemma 6 *If $n \dashrightarrow n_f$, $n \dashrightarrow n_g$, and n is illegal but both n_f and n_g are legal, then duplicating n into $n_1 \dashrightarrow n_f$ and $n_2 \dashrightarrow n_g$ makes n_1 and n_2 legal.*

Proof: The support of n is a subset of both the supports of n_f and n_g, but the output set of the node placement constraint of n is a superset of the node placement constraints of both n_f and n_g. By duplicating n into $n_1 \dashrightarrow n_f$ and $n_2 \dashrightarrow n_g$, node placement constraint of n_1 is contained in that of n_f and thus n_1 is legal. Similarly for n_2. ■

Make_legal traverses the Boolean network in a reverse topological order, i.e. a node is visited after all its fanouts have been visited. During the traversal, if it sees an illegal node, it collapses the node into its fanouts until the node becomes legal. Hence, there is a frontier moving from each primary output to primary inputs in its support where every node is legal on the side of the frontier toward the primary output. If the sum-of-product expression of the fanout, as a result of collapsing a node into one of its fanouts, exceeds a user-defined parameter, t, number of literals, the node is replicated for each fanout until it becomes legal. The intuition behind this parameter is that large nodes tend to have more common subfunctions with other nodes and thus allow for sharing. However, the parameter should not be too large since it can result in explosion in memory usage.

As shown above, legality of a node can be checked efficiently, that is, it is linear in the size of the node placement constraint. Hence, the *make_legal* operation is efficient.

5.3 Constraint-Driven Synthesis

The constraint generation step takes a possibly illegal Boolean network and makes it legal. Theorem 5.1 guarantees that there exists a point placement for this network. However, by definition of the point placement of a circuit, nodes are assumed to be a point; hence, they have no area. In reality, nodes have area and the length of a longest path depends strongly on the size of a Boolean network. The constraint-driven synthesis step is responsible for minimizing the area of an already legal Boolean network while preserving its legality. As mentioned in Section 5, we use the number of factored-form literals of a Boolean network as a measure of the area of the circuit represented by the Boolean network. So this step is to optimize the network such that we get a minimum literal legal Boolean network.

We leverage the well developed algebraic transformations in the conventional logic synthesis by extending them to deal with and produce legal Boolean nodes. Each of these operations is explained below.

5.3.1 Fast_Extract

The *fast_extract* algorithm is explained in [16]. It basically looks for a two-cube divisor or a two-literal cube that reduces the most number of literals in every iteration.

When dealing with legal Boolean network, this algorithm may result in illegal divisors. For example, assume that node n is the best divisor found and it divides nodes x, y, and z. Then the output set of the node placement constraint of n is the union of the output sets of the node placement constraints of x, y, and z. From Section 5.2, we know that the legal region of n may be empty and n may therefore be illegal. However, it may be the case that n remains legal if it only divides x and y, or x and z, etc. Hence, the fast_extract algorithm is modified such that the best legal divisor is chosen in every iteration.

If node n divides a set of nodes N, then complexity of finding a subset N_l of N which preserves the legality of n and has the largest reduction in the number of literals is exponential in the size of N. Hence, a heuristic is used to select an optimal subset. First the nodes in N are ordered in decreasing sizes of the legal regions to form a list N_{sorted}. Then N_{sorted} is linearly traversed. Each node is added to the subset N_l if the legality of n is preserved. Node n is used as a divisor if it reduces the number of literals in the network.

In this paper, the fast_extract implemented in SIS is used.

5.3.2 Resubstitution

In the conventional logic synthesis, a node n is resubstituted into another node x if n divides x. This may affect the legality of both n and x. The following observation states when n and x can become illegal.

Observation 1 *If n divides x and both n and x are legal before resubstitution, then after resubstitution*

1. *x can become illegal if its support is not the superset of that of n.*
2. *n can become illegal if its output set is not the superset of that of x.*

In this paper, n can only be resubstituted into x if the legality of n is preserved. Hence, a check is made before every resubstitution.

5.3.3 Full_Simpfily

There are two types of *don't cares*, i.e. the observability don't cares (ODCs) and the satisfiability don't cares (SDCs). Computing the exact ODCs of a node is computationally expensive. In practice, a subset of the ODCs called the compatible ODCs (CODCs) are computed. These CODCs are expressed in terms of the primary inputs. Then together with the external don't cares (XDCs) of the primary outputs, a don't care set in terms of the immediate fanins is computed using an image computation. In computing the SDCs, a support filter is used. A node is included in the SDCs if its support set intersects the support set of the node being considered. Employing SDCs in the minimization procedure can result in boolean resubstitutions. The support filter procedure can also be used in the image computation of the CODCs and XDCs. Once the SDCs are computed and the XDCs and CODCs are expressed in terms of immediate fanins, a two-level minimization algorithm is invoked to find an optimized expression. This is simply a brief description of the *full_simplify*. For a more detail explanation, we refer the readers to [10].

Lemma 7 *Throughout full_simplify computation, the only steps that can introduce illegality into the network are the image computation and the SDC computation.*

Proof: Let node n be the node we are computing don't cares for. Legality of the Boolean network can only change if an edge is added to the network. During the whole full_simplify process, only the fanin edges of n can be added. Edges of fanins of other nodes can not change. Adding a fanin edge to n means that a resubstitution happens and Observation 1 applies. Potential new fanin edges of n are added only during the image computation and SDC computation through the support filter, which basically says that a node x is a potential divisor of n if the support of x intersect the support of n. ■

We therefore constrain this operation by allowing a node x to be in the support filter when computing full_simplify for node n if the inclusion of node x preserves the legality of the network according to Observation 1.

5.3.4 Synthesis Flow

With all the above basic operations, a synthesis flow is then a script similar to the *script.rugged* in SIS. An empirical study needs to be conducted to derive an optimal script.

6 Experimental Results

To see the effect of the proposed approach, we have implemented the basic operations described in Section 5.3. An optimization script has been created and we call it *script.wire*, which consists of:

```
make_legal
eliminate 5
sweep; eliminate -1
simplify -m nocomp
eliminate -1
sweep; eliminate 5
simplify -m nocomp
resub -a
fx
resub -a; sweep
eliminate -1; sweep
full_simplify -m nocomp
```

Our experiment uses SIS and Ritual version 3.4, a timing-driven standard cell placer [12]. The input blif file and a randomly generated pad assignment file is read into SIS. The *script.wire* optimization script is run in SIS to generate an optimized logic netlist. The optimized netlist is mapped to the standard cell technology library *stdcell2_2.genlib* of SIS. The mapped netlist is then placed by Ritual with a fixed pad assignment. We measure the length of the longest path and the delay of the Ritual output. The distance of two cells is measured as the Manhattan distance from the center of both cells. The length of a path is the sum of all distances between consecutive cells along the path.

Table 1 shows the results for four circuits. The circuit *bbaraComb* is obtained from the sequential circuit *bbara* by removing all latches and treating the outputs of the latches as primary inputs and the inputs to the latches as primary outputs of the network. Column 2, 3, and 4 show the number of literals in factored forms of the scripts script.rugged, script.delay, and script.wire respectively. Columns 5, 6, and 7 list the length of the longest path for each script. Columns 8, 9, and 10 show the CPU time. The experiments were run on a DEC AlphaServer 8400 with 2GB of memory. The runtime is for the technology independent step.

As shown in this table, although the number of literals in *script.wire* approach is more than that of *script.rugged*; the length of its longest path is the same for *rd53* and better in other circuits. The longest paths are much shorter than *script.delay* results. As seen from this table, the runtime is comparable. This is expected since the legality checking is linear in the size of the node placement constraints and hence its runtime is a minor part of the total runtime.

Table 2 shows the delay computed by Ritual for the four circuits. Columns 2, 3, and 4 show the cell delay for each script. The wire delay is shown in columns 5, 6, and 7. The total delay is listed in columns 8, 9, and 10. Except for the total delay of *z4ml* running *script.delay*, the total delay of all circuits is the best using *script.wire*.

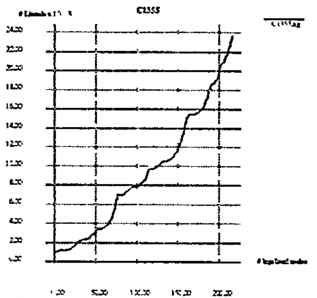

Figure 9: Number of literals vs number of nodes legalized for C1355.

7 Open Issues and Future Work

Though the results in the previous section shows that the approach performs satisfactorily, these circuits are fairly small. For bigger circuits, the number of nodes in a legal network can be large and optimizing such large networks using operations like *fast_extract* and *full_simplify* can be very expensive.

To illustrate this, we plot the number of literals versus the number of nodes in the constraint generation step for C1355 as shown in Figure 9. On the x-axis is the number of illegal nodes that are legalized. On the y-axis is the number of literals in the Boolean network. The network increases from 1032 literals to 23709 literals after 216 nodes have been legalized out of a total of 514 nodes in the network.

There are three various directions that can be pursued to address this problem. The first one is to improve the area optimization algorithm presented in this paper. Rewiring and redundancy removal is a technique that falls into this direction. SPFDs [2] can be used to minimize, rewire circuits, and potentially legalizing nodes. In this paper, we are assuming that we are given a circuit represented as a Boolean network. We then apply make_legal and several algebraic transformations followed by don't care minimization. The final circuit depends on the quality of the initial Boolean network. Alternatively, the Boolean network can first be collapsed as much as possible into a two-level circuit where all primary outputs are expressed in terms primary inputs. Then functional decomposition, like [11], can be used to decompose the network into a minimum literal legal Boolean network.

The second direction which we believe is more promising is to relax the constraint that every path must be monotonic. In other words, this is about solving the slack-based synthesis problem instead of the more restrictive IP-based synthesis problem. This can be done by applying

Table 1: Path length comparison of *script.rugged, script.delay,* and *script.wire* for IP-based synthesis.

	Number of Literals			Length of Longest Path			CPU Time		
Name	sc.rugged	sc.delay	sc.wire	sc.rugged	sc.delay	sc.wire	sc.rugged	sc.delay	sc.wire
z4ml	41	84	49	1324	1342	1025	0.2	0.3	0.3
rd53	42	62	50	1122	1624	1122	0.1	0.3	0.2
rd73	74	178	87	1689	2457	1680	0.8	1.8	1.2
bbaraComb	69	79	109	2021	1573	1464	0.5	0.5	0.3

Table 2: Delay comparison of *script.rugged, script.delay,* and *script.wire* for IP-based synthesis.

	Cell Delay			Wire Delay			Total Delay		
Name	sc.rugged	sc.delay	sc.wire	sc.rugged	sc.delay	sc.wire	sc.rugged	sc.delay	sc.wire
z4ml	5.66	5.28	4.78	0.93	1.03	0.97	6.59	6.31	5.75
rd53	9.73	7.37	5.94	1.67	2.13	1.42	11.40	9.50	7.36
rd73	7.01	5.09	5.59	1.37	0.88	0.86	8.38	5.97	6.45
bbaraComb	8.18	6.22	4.90	2.19	1.72	1.08	10.37	7.94	5.98

the IP-based synthesis algorithm only to a subset of the paths. Intuitively, we can wireplan only the critical paths so that no diversions are allowed in them; other paths can have diversions. One approach would be to modify the definition of legality so that legality is checked based on the primary inputs and outputs that are relevant only to the critical paths. Only the nodes on the critical paths are legalized. We have done some preliminary experiment and our results show that if you select top few longest paths and legalize all the nodes on those paths, then the area penalty is not very high. However, at present there is no easy way to perform a meaningful comparison of this approach (i.e. modified IP-based algorithm to solve the slack-based synthesis problem) with the conventional approach. For that, we need a placement tool that uses the same delay model as ours and we have not been successful at making Ritual use our model.

One other issue that needs further attention is that of pin assignment The approach in this paper assumes that the pin assignment is given. In the design process, usually only partial pin assignment is given. However, the quality of the final solution strongly depends on the pin locations. Therefore, we need to look into algorithms to find good pin assignment during synthesis. Such an algorithm can also be used to extend this approach to handle sequential circuits by finding good placement for the latches present.

The optimizations that we have shown are technology independent. We have not yet addressed the issue of technology mapping. Also, we have completed ignored gate delays. We are presently looking into both of these issues, i.e. technology mapping and how to best incorporate gate delays in our approach.

Finally we are also looking into extending the proposed approach to handle other interconnect issues, like crosstalk and reliability.

8 Conclusions

We have proposed a new approach to deal with the increasingly importance of wire delays in deep submicron technologies. It is based on the fact that the shortest path between any two points in a circuit is the Manhattan distance between them. We showed an example of why conventional logic synthesis may produce circuits where the minimum distance can not be achieved.

The proposed approach still decouples logic synthesis phase and place & route phase. It consists of a constraint generation step which produces a legal Boolean network, which can be placed such that every path is monotonic, and a constraint-driven synthesis step which minimizes the legal Boolean network while preserving legality. We show an example of how this approach can be extended to solve the slack-based synthesis problem. Finally, we describe directions for future work which includes an investigation into a new placement tool that works together with the proposed approach.

References

[1] Semiconductor Industrial Alliance. *National Technology Roadmap for Semiconductors.* 1997.

[2] R.K. Brayton. Understanding SPFDs: A new method for specifying flexibility. *IWLS,* May 1997.

[3] R.K. Brayton, A.L. Sangiovanni-Vincentelli, and G. Hachtel. Multi-level logic synthesis. *Proceedings of the IEEE,* vol. 78(no. 2):264–300, February 1990.

[4] J. Cong and Z. Pan. Interconnect Performance Estimation Models for Synthesis and Design Planning. In *IWLS 98.*

[5] K. Keutzer, A.R. Newton, and N. Shenoy. The future of logic synthesis and physical design in deep-submicron process geometries. In *ISPD,* pages 218–224, 1997.

[6] Y. Kukimoto and R.K. Brayton. Hierarchical functional timing analysis. In *DAC,* 1998.

[7] R. H. J. M. Otten and R. K. Brayton. Planning for Performance. In *DAC,* June 1998.

[8] M. Pedram and N. Bhat. Layout Driven Logic Restructuring/Decomposition. In *ICCAD,* pages 134–137, November 1991.

[9] M. Pedram and N. Bhat. Layout Driven technology Mapping. In *DAC,* pages 99–105, June 1991.

[10] H. Savoj. *Don't cares in multi-level network optimization.* PhD thesis, University of California, Berkeley, May 1992.

[11] C. Scholl and P. Molitor. Communication based FPGA synthesis for multi-output boolean functions. In *ASP-DAC,* pages 279–288, August 1995.

[12] A. Srinivasan, K. Chaudhary, and E. S. Kuh. Ritual: a performance driven placement algorithm. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing,* 39(11):825–840, November 1992.

[13] G. Stenz, B. M. Riess, B. Rohfleisch, and F. M. Johannes. Timing Driven Placement in Interaction with Netlist Transformations. In *ISPD 97,* Napa Valley, CA, 1997.

[14] H. Vaishnav and M. Pedram. Routability-Driven Fanout Optimization. In *DAC,* pages 230–235, June 1993.

[15] H. Vaishnav and M. Pedram. Minimizing the Routing Cost During Logic Extraction. In *DAC,* pages 70–75, June 1995.

[16] J. Vasudevamurthy and J. Rajski. A method for concurrent decomposition and factorization of Boolean expressions. In *ICCAD,* pages 510–513, November 1990.