Alphabetic Fanout Optimization

Hirendu Vaishnav and Massoud Pedram

CENG Technical Report 92-15

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213)740-4458

Alphabetic Fanout Optimization

Hirendu Vaishnav and Massoud Pedram
Department of Electrical Engineering – Systems
University of Southern California, Los Angeles CA 90089

23 October 1992

Abstract

We propose an efficient fanout optimization algorithm which improves circuit performance while honoring an order restriction on the sinks. This ordering is derived based on the network structure or the required time constraints at the primary outputs. We introduce the concept of apropos trees which allows us to significantly reduce the solution space for generating optimal alphabetic trees. Using this concept, we show that optimal alphabetic binary trees on n leaf nodes can be generated in $O(n^3)$ time complexity for monotone cost trees. This is a significant result as it gives the first successful solution to the alphabetic binary tree construction in such generality. For the alphabetic nonbinary tree construction, we propose a set of rules which reduce size of the solution space while maintaining the optimality. We apply these techniques to the problem of generating optimal alphabetic fanout trees where we introduce the notion of non-inferior set of fanout trees and use this to obtain further reduction in the solution space. We tried a number of ordering mechanisms and obtained an average of 8-10% improvement in performance and 4-7% reduction in chip area compared to the SIS fanout optimization tool.

Contents

1 Introduction and Motivation

1.1 Fanout Optimization

In past years, multi-level synthesis research has concentrated on techniques that maximize logic sharing resulting in circuits with high number of fanouts per net. Excessive loads due to high fanout count result in considerable performance degradation. Fanout optimization is thus necessary to improve the circuit performance.

Golumbic [5] and Klawe et. al. [6] addressed fanout optimization using the *unit* delay model with a restriction on maximum number of fanouts per buffer. In this sense, their algorithms were optimal *fanout bounding* algorithms. Golumbic's "combinatorial merging" algorithm [5] was an application of the *t-ary* Huffman tree generation procedure in logic synthesis. However, unit delay model is inaccurate for the fanout problem as the delay at the multiple fanout points is greatly influenced by the number of the sinks.

Berman et. al. [1], Singh et. al. [13], and Touati [15] used the more realistic *unit fanout* and *library* delay models[1]. It was shown in [1, 15] that even under very simplistic assumptions, fanout problem is *NP-hard*. These results led to different heuristics for fanout optimization. Berman et. al. proposed a heuristic which generates fanout trees by maintaining an order derived on the sinks using the required times. Their algorithm generates optimal fanout trees for a restricted set of trees which have depth at most two and have identical required times for all sinks. Singh et. al. proposed a set of heuristics which consisted of mainly three operations; *repowering, critical signal isolation*, and *load balancing*. A strong feature of this heuristic was that it considered sinks with both polarities concurrently.

Touati's work on fanout optimization is the most comprehensive to date. He combined buffer optimization with fanout optimization and extended Golumbic's algorithm to take into account varying loads and variable node degrees for internal nodes of the fanout tree. To integrate *critical signal isolation* with *load balancing*, Touati introduced the "LT-Trees" which balance the loads and isolate critical signals simultaneously by grouping signals with similar required times at the same levels of the fanout tree. A common characteristic of these heuristics is that they generate fanout trees such that depth of a sink is a non-decreasing function of the required timeat the sink. It has been shown in [15] that this need not be the case[2] for an optimal fanout tree.

[1] Under the *unit fanout* delay model, the delay of the buffer is given by $1 + \alpha\, number\_of\_fanout$, where $0 < \alpha \leq 1$. The *library* delay model uses accurate values of intrinsic delay and load for each fanout.

[2] Incidentally, our algorithm produces trees which do not necessarily satisfy this property.

1.2 Motivation

Routing plays an important role in determining the total circuit area and circuit performance and thus must be addressed as early as possible during the design process. Routing is a crucial factor during fanout optimization for two reasons: 1) Interconnect delays influence the criticality of signal paths. Fanout optimization must be applied to nodes along the critical paths in order to improve the circuit performance. Interconnect delays must be calculated based on the placement information. Fortunately, reliable placement solutions can be obtained at this late stage of the logic synthesis. 2) Circuit routability and wiring congestion profile are influenced by the type of fanout trees we construct. In particular, a fanout tree that preserves the wire crossings in the original network while maximizing the performance gain should be used. Hence, it is worthwhile to incorporate and utilize placement information during the fanout optimization.

Instead of using the placement information, we could use topological information to Alternatively, ordering based on the required times at the sinks can be used on the preimise that sinks with similar required time are likely to be on the same level of the fanout tree.

The main characteristic of alphabetic trees is that they are free of internal edge crossings. Most of the previous work in logic synthesis has ignored such edge crossing constraints which determine planarity of the underlying network. Using any of the fanout algorithms proposed hitherto, even if G is planar, the new graph G' might be far from being planar. If topological or placement information is used to order sinks, even if the circuit is nonplanar, it is desirable to have a fanout algorithm that does not increase the nonplanarity and hence does not create more routing difficulties than are present. However, we need to make sure that we achieve this goal at minimal cost. It has been shown that using the unit delay model, the increase in depth is at most 1 and in size is a constant multiplicative factor for each optimal alphabetic fanout tree as compared to optimal nonalphabetic trees[6, 9].

Most of the work on alphabetic trees has focused on binary trees with integer weights for the leaves. Depending on the *tree cost function* and the *combining function*[3], alphabetic trees can be classified as follows. The *minimax* alphabetic trees are those for which the combining function is the maximum of child weights while the tree cost is weight of the root which is being minimized. Optimal algorithms for generating such alphabetic trees have been proposed in numerous papers [8, 7, 4, 11, 9, 2]. The best known algorithm is due to Kirkpatrick et. al. [9] and solves the alphabetic t-ary *minimax* tree problem in linear time complexity for integer weights and in O(nlogn) for non integer weights. Coppersmith et. al. [2] extended the work of Kirkpatrick et. al. to allow variable

[3]Tree cost function is the cost of the tree defined on the weights of the leaf and internal nodes. Combining function is a function using which the weight of a parent node is calculated, given the weights of all its children.

node degrees. All these algorithms correspond to the *unit* delay model for alphabetic fanout trees[4]. Unfortunately, use of the *unit fanout* or *library* delay models make alphabetic fanout optimization problem *NP-hard*, rendering the above mentioned algorithms inapplicable.

In this report, we extend the alphabetic fanout optimization idea to the unit fanout and library delay models. We propose a tree generation mechanism to construct optimal alphabetic trees. It is shown that if the tree has monotone tree cost function, the solution space can be reduced substantially. This results in $O(n^3)$ complexity algorithm to generate optimal alphabetic binary trees. This is the first such algorithm to address the construction of optimal alphabetic binary trees in this generality. We apply the proposed tree generation mechanism to fanout optimization in order to incorporate placement/routing into the performance optimization. We also characterize an important property of fanout trees which further reduces the solution space, resulting in a very effective algorithm to generate optimal alphabetic fanout trees.

2 Alphabetic Trees

In the previous section, we gave definitions of the *combining* and *tree cost* functions. In this section, we will generalize these concepts. We follow the terminology given below throughout the paper.

- A graph $G(V, E)$ is *weighted* if there is a weight W_i associated with each node $v_i \in V$. It is *K-weighted*, if each weight W_i has K components (i.e. is K dimensional).
- A *forest* is an acyclic graph. *K-weighted* forests are defined similarly.
- A *tree* is an acyclic, connected graph. *K-weighted* trees are defined similarly.

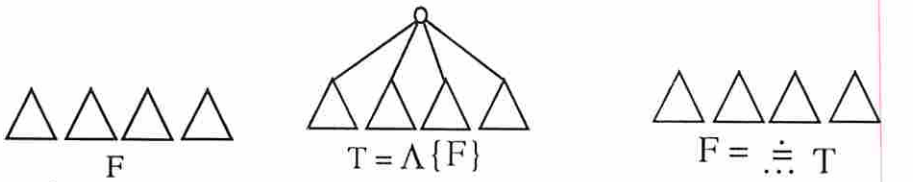
2.1 Non-alphabetic Trees

A general form of tree generation problem is as follows.

Problem 2.1 K-WEIGHTED_TREE_GENERATION

- **Instance:**

[4] A brief note here is that we will be concerned with *maximin* problem, i.e. the problem of maximizing the required time at the root of the fanout tree using the minimum of the required times for its fanins.



a) Original Forest b) The JOIN operation c) The SPLIT operation.

Figure 1: Illustration of operations JOIN and SPLIT

1. *A set of n leaf nodes* $(L_1, L_2, \ldots, L_n)$ *with K-dimensional weights* $W_{L_i} = (w^1_{L_i}, w^2_{L_i}, \ldots, w^K_{L_i})$.
2. *A K-dimensional combining function* $(f^1, f^2, \ldots, f^K)$ *which combines t leaf or internal nodes generating an internal node* I_p*, where the* j*th component of the new weight is given by*

$$w^j_{I_p} = f^j(W_{child_{I_{p,1}}}, W_{child_{I_{p,2}}}, \ldots, W_{child_{I_{p,t}}}) \text{ for } 1 \leq i \leq K$$

3. *A tree cost function* $C(F) = C(W_{L_1}, W_{L_2}, \ldots, W_{L_n}, W_{I_1}, W_{I_2}, \ldots, W_{I_m})$*, where* $(I_1, I_2, \ldots, I_m)$ *is the set of all internal nodes.*

- **Problem:** *Generate a minimum cost tree.*

Most of the trees considered by researchers in logic synthesis and information sciences are 1-weighted trees. Fanout trees with the unit fanout and library delay models for fanout optimization actually correspond to 2-weighted trees since each node has two weights associated with it, i.e., the signal required time and the load.

2.2 Alphabetic Trees

Given a set of n ordered leaf nodes, an *alphabetic tree* is a rooted tree which maintains the given order on the leaf nodes and has no internal edge crossing.

Definition 2.1 *Given a forest F with rooted trees, we define* JOIN *of F (denoted by* $\wedge F$*) as generating a rooted tree T having as immediate children roots of trees in F (Figure 1).*

Definition 2.2 *Given a rooted tree T with root x, we define* SPLIT *of T (denoted by* $\doteqdot T$*) as deleting the root making all its direct children rooted trees in a forest F (Figure 1).*

The argument of JOIN is a forest and the argument of SPLIT is a tree. These operations can be applied to a set of forests and a set of trees as well. If we denote by Ψ a set of r trees $\{T_1, T_2, \ldots, T_r\}$, then $.\dot{=}.\Psi$ generates a set of forests $\{F_1, F_2, \ldots, F_r\}$ by applying SPLIT on each element of Ψ. After applying JOIN operation to $\{F_1, F_2, \ldots, F_r\}$, we get back $\{T_1, T_2, \ldots, T_r\}$, i.e., $\wedge(.\dot{=}.\Psi) = \Psi$. Similarly, $.\dot{=}.(\wedge\Psi) = \Psi$.

Let Ψ_1^n denote the set of alphabetic trees on leaf nodes 1 through n and $\Phi_n = |\Psi_1^n|$. In general, Ψ_i^m denotes the set of alphabetic trees on leaf nodes i through m and $\Phi_{m-i+1} = |\Psi_i^m|$.

Lemma 2.1 *The following equation generates the set of all alphabetic trees* Ψ_i^m*:*

$$\Psi_i^m = \bigcup_{j=0}^{m-i-2} (\wedge(\Psi_i^{i+j} \times \Psi_{i+j+1}^m) \cup \wedge(\Psi_i^{i+j} \times (.\dot{=}.\Psi_{i+j+1}^m))) \bigcup \wedge(\Psi_i^{m-1} \times \Psi_m^m); \text{ for } i \leq m \tag{1}$$

where $\cup$ *denotes the set-union operation and* $\Psi' \times \Psi'' = \{F \mid F = \{T', T''\} \mid T' \in \Psi', \ T'' \in \Psi''\}$ *and* $\Psi' \times (.\dot{=}.\Psi'') = \{F \mid F = \{T', t''\}, \ \forall\, t'' \in (.\dot{=}.T''), \ T' \in \Psi', \ T'' \in \Psi''\}$.

Proof We prove the above by induction on d which we define to be $m - i + 1$.

- $d = 1$. Ψ_i^i consists of a tree with one leaf node.
- $d = 2$. $\Psi_i^{i+1} = \wedge(\Psi_i^i \times \Psi_{i+1}^{i+1})$ consists of a tree structure with two leaf nodes.
- We assume the equation is true for d and prove its correctness for $d + 1$.

Examining equation (1), increasing value of j in the equation corresponds to incrementally adding more leaf nodes to the leftmost child of the tree root. By induction Ψ_i^{i+j} and Ψ_{i+j+1}^m are sets of all alphabetic trees on leaf node i through $(i+j)$ and $(i+j+1)$ to m, respectively. Given any alphabetic forest F_i with at least two roots on leaf nodes $(i+j+1)$ through m, there is a corresponding rooted tree T_i in Ψ_{i+j+1}^m generated by adding a root R_i to the forest F_i and making all the original roots of F_i direct children of R_i. Thus, $(.\dot{=}.\Psi_{i+j+1}^m)$ represents the set of all alphabetic forests with two or more rooted trees on leaves $(i+j+1)$ through m. Hence, $\wedge(\Psi_i^{i+j} \times \Psi_{i+j+1}^m) \cup \wedge(\Psi_i^{i+j} \times (.\dot{=}.\Psi_{i+j+1}^m))$ generates all alphabetic trees where exactly $j+1$ leaf nodes, nodes i through $(i+j)$, are descendents of the leftmost child of the root. ∎

Equation (1) is illustrated for n = 1, 2, 3 and 4 in Figure 2.

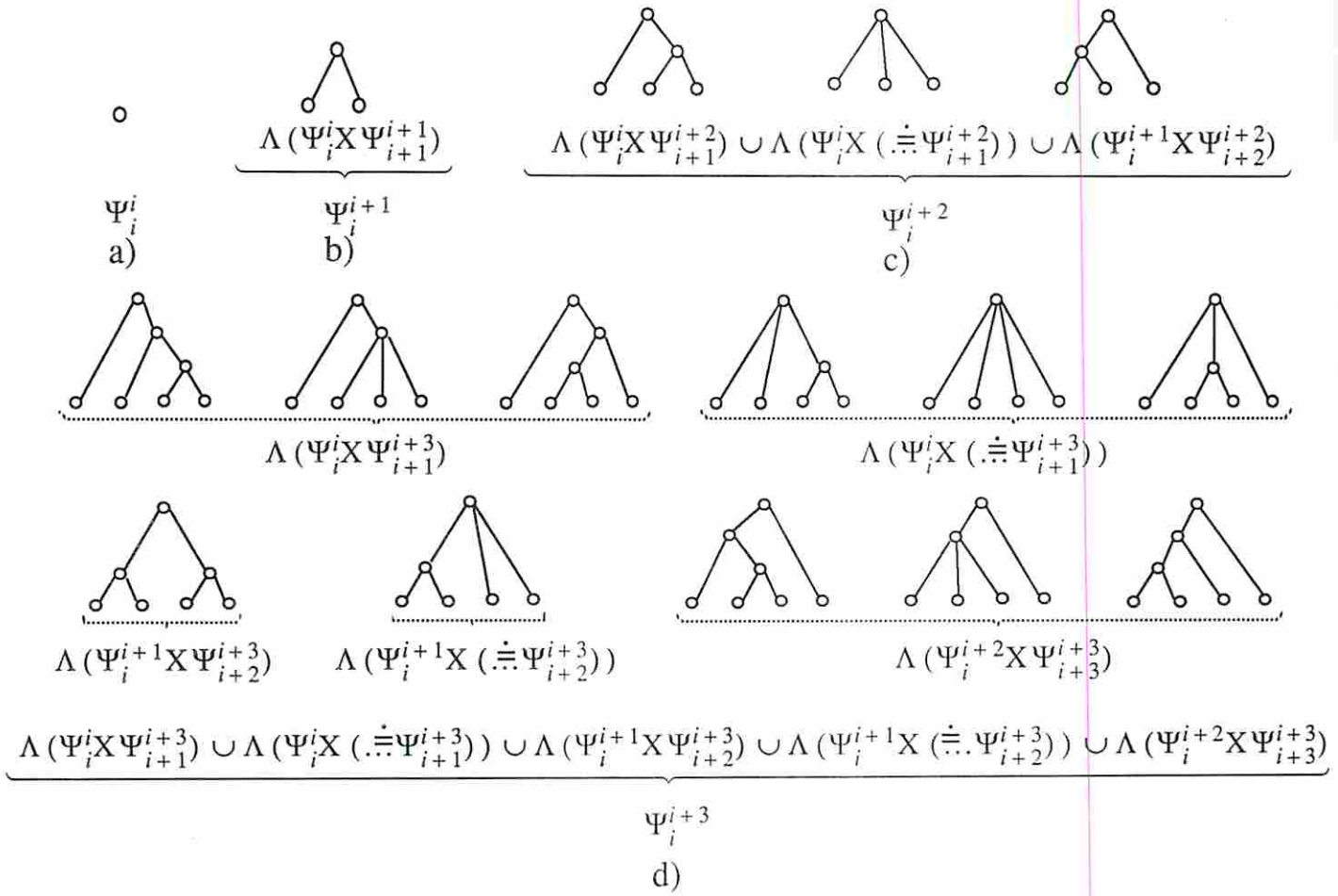


Figure 2: Generating alphabetic trees for a) 1 leaf node, b) 2 leaf nodes, c) 3 leaf nodes, and d) 4 leaf nodes

It is evident that each term in equation (1) generates a distinct set of trees. From (1), we can determine Φ_d using a recursive difference equation:

$$\Phi_d = \begin{cases} 2\sum_{j=2}^{d-1}\{\Phi_{d-j}\Phi_j\} + \Phi_{d-1}\Phi_1. & \text{if } 2 \le d \\ 1 & \text{if } d = 1 \end{cases} \tag{2}$$

The above equation is a convoluted recurrence similar to the equation for *Catalan numbers* [12, 3], the solution to which is given below.

Lemma 2.2 *Number of alphabetic tree structures on n leaf nodes is given by*

$$\Phi_n = -\frac{1}{4}\sum_{j=0}^{\lfloor n/2 \rfloor}\binom{n-j}{j}(-6)^{n-2j}\binom{1/2}{n-j}; \text{if } 2 \le n \tag{3}$$

Proof Equation (2) can be rewritten as,

$$\Phi_d = 2\sum_{j=1}^{d-1}\Phi_{d-j}\Phi_j - \Phi_{d-1} + [d=1]. \qquad \text{if } d > 0$$

Where $[d = 1]$ is 1 when $d = 1$ and 0 otherwise. The solution to the above can be obtained using generating function principles. The generating function $\Phi(Z)$ for any series with series element Φ_i is given by $\sum_i \Phi_i Z^i$.

Shifting our series left by one we get a new series Λ in which $\Lambda_i = \Phi_{i+1}$.

$$\Lambda_d = 2\sum_{j=0}^{d-1}\Lambda_{d-1-j}\Lambda_j - \Lambda_{d-1} + [d=0]. \qquad \text{if } d \ge 0$$

Multiplying the above equation with Z^d and summing over d, we get,

$$\Lambda(z) = \sum_d \Lambda_d Z^d = 2\sum_{d,j}\Lambda_{d-1-j}\Lambda_j Z^d - \sum_d \Lambda_{d-1}Z^d + \sum_{d=0} Z^d$$

$$\Lambda(z) = 2\sum_{d,j}\Lambda_{d-1-j}Z^{d-j}\Lambda_j Z^j - \sum_d \Lambda_{d-1}Z^d + 1$$

Using identity $\sum_i \Lambda_{i-1}Z^i = Z\Lambda(Z)$ we get,

$$\Lambda(z) = 2Z\Lambda(Z)\Lambda(Z) - Z\Lambda(Z) + 1 = 2Z\Lambda(Z)^2 - Z\Lambda(Z) + 1$$

$$2Z\Lambda(Z)^2 - (1+Z)\Lambda(Z) + 1 = 0$$

The coefficient of Z^i in the solution of the above equation for $\Lambda(Z)$ will give us the ith element Λ_i in the series Λ. Since we shifted the series Φ left by one place, Λ_i element will give us Φ_{i+1}. Solving the equation for $\Lambda(Z)$,

$$\Lambda(Z) = \frac{1 \pm \sqrt{(1+Z)^2 - 8Z}}{4Z} = \frac{1 \pm \sqrt{1 - 6Z + Z^2}}{4Z}$$

However, $\frac{1+\sqrt{1-6Z+Z^2}}{4Z}$ can not be the solution as it does not satisfy the initial condition $\Lambda_0 = 1$. Expanding $\frac{1-\sqrt{1-6Z+Z^2}}{4Z}$ as $\sum_i \Lambda_i Z^i$ will give us coefficients $\Lambda_i = \Phi_{i+1}$. Using binomial expansion $\sqrt{1 - 6Z + Z^2} = \sum_k \binom{1/2}{k} Z^k (Z-6)^k$. The coefficients of Z^i in this equation is given by $\sum_{j=0}^{\lfloor i/2 \rfloor} \{$ Coefficient of Z^j in $(Z-6)^{i-j}\} \binom{1/2}{i-j}$. (Substituting $i-j$ for k). Using binomial expansion, this is

$$\Lambda'_i = \sum_{j=0}^{\lfloor i/2 \rfloor} \binom{i-j}{j} (-6)^{i-2j} \binom{1/2}{i-j} \tag{4}$$

Hence, $\Lambda(Z) = Z^{-1}/4 - \sum_i \Lambda'_i Z^{i-1}/4$. The first term only affects the -1th element, and from the second term $-\Lambda'_i/4$ is Λ_{i-1}. However, since we had initially shifted the sequence left, $\Lambda_{i-1} = \Phi_i = -\Lambda'_i/4$. ∎

Thus, number of distinct alphabetic trees is thus $O(6^N)$. The number of distinct alphabetic trees on 1 to 12 leaf nodes is given below.

No. of Leaf nodes	No. of distinct Alphabetic trees	No. of Leaf nodes	No. of distinct Alphabetic trees
1	1	7	903
2	1	8	4279
3	3	9	20793
4	11	10	103049
5	45	11	518859
6	197	12	2646723

Table 1: Number of distinct alphabetic trees

We are often interested in finding the "best" alphabetic tree given the combining functions and the tree cost function.

Algorithm 2.1 Find_Best_Alphabetic_tree ($\mathcal{N}$)
$\mathcal{N}$ is given set of n leaf nodes with weights
begin
  for i = 1 to n **do**
    Ψ_1^i = Gen_Alp_Trees(1, i, $\mathcal{N}$)
  Result = Find_best(Ψ_1^n)
end

Gen_Alp_Trees uses the tree generating equation (1) to construct all alphabetic trees on i leaf nodes[5]. The function *Find_best(Ψ_1^n)* ranks exponential number of trees in set Ψ_1^n using the tree cost. Subtree structures may be saved, thus improving the efficiency of the tree generation procedure.

2.3 Apropos alphabetic trees

In general, depending on the combining and tree cost functions, complexity of determining the best tree can be reduced by considering only the *apropos*[6] trees, i.e. a subset of all tree structures which are non-inferior with respect to each other but superior to a large number of trees for optimizing the tree cost. In particular, if the tree cost is monotone, the optimal cost alphabetic binary tree can be found in polynomial time.

Definition 2.3 *A tree is said to be* **monotone** *if the tree* cost *is monotone in tree cost of each of its subtrees, that is, if we increase (decrease) the cost of some subtree, the tree cost will not decrease (increase).*

Theorem 2.3 *To obtain an optimal cost tree it is sufficient to consider only those trees whose subtrees have optimal tree costs.*

Proof Since the tree cost function is monotone, we could always substitute any non-optimal subtree by an optimal subtree without degrading the tree cost. ∎

Corollary 2.4 *To generate an optimal alphabetic tree using equation (1), it is sufficient to consider only the optimal trees as arguments of the* JOIN *operator.*

[5]The set Ψ_i^m is identical to set Ψ_1^{m-i+1}. Hence, as long as we are not computing the tree cost using the leaf nodes, we can use Ψ_1^{m-i+1} in place of Ψ_i^m

[6]Apropos: to the purpose, pertinent, appropriate (Oxford Dictionary).

Let θ_i^j denote the optimal tree on the set of leaves (i,...,j). Considering tree generation equation (1), we can replace $\wedge(\Psi_i^{i+j} \times \Psi_{i+j+1}^m)$ by $\wedge\{\theta_i^{i+j}, \theta_{i+j+1}^m\}$. Similarly we can substitute $\{\theta_i^{i+j}\}$ for Ψ_i^{i+j} in $\wedge(\Psi_i^{i+j} \times (\dot{=}.\Psi_{i+j+1}^m))$. Let $Ap\Psi_i^j$ denote the set of alphabetic trees on leaves $(L_i, \ldots, L_j)$ such that for each tree in the set, every child subtree is optimal. Then we can rewrite equations (1) and (2) as:

$$Ap\Psi_i^m = \bigcup_{j=0}^{m-i-2} (\wedge\{\theta_i^{i+j}, \theta_{i+j+1}^m\} \cup \wedge(\{\theta_i^{i+j}\} \times (\dot{=}.Ap\Psi_{i+j+1}^m))) \bigcup \wedge\{\theta_i^{m-1}, \theta_m^m\} \quad ; \text{for } i \leq m-1 \tag{5}$$

$$Ap\Phi_d = \begin{cases} \sum_{j=1}^{d-2} \{Ap\Phi_{d-j} + 1\} + 1 & \text{if } 2 \leq d \leq n \\ 1 & \text{if } d = 1 \end{cases} \tag{6}$$

Lemma 2.5 *The number of apropos alphabetic trees on n leaves ($n \geq 2$) is equal to $2^{n-1} - 1$ if the tree cost is monotone.*

Proof We prove this by induction.

- For two leaf nodes $Ap\Phi_2 = 2^{2-1} - 1 = 1$ which is true since there is only one tree structure on two leafs.
- Assuming the above equation is true for $i \geq 2$, we prove its correctness for $i+1$:

$$Ap\Phi_{i+1} = \sum_{j=1}^{i-1} \{Ap\Phi_{i+1-j} + 1\} = \sum_{j=1}^{i-1} \{2^{i-j} - 1 + 1\} + 1 = \sum_{j=0}^{i-1} \{2^j\} + 1 - 2^0 = 2^i - 1$$

Thus, $Ap\Phi_n = 2^{n-1} - 1$. ∎

For all practical purposes, this is the *apropos* set of trees any optimal alphabetic tree algorithm has to consider when the tree under consideration has monotone tree cost function and hence forms a loose upper bound on the complexity of any monotone alphabetic tree problem. A corresponding algorithm is shown below.

Algorithm 2.2 Find_Best_Alphabetic_tree_Apropos ($\mathcal{N}$)
$\mathcal{N}$ is given set of n leaf nodes with weights
begin
  for j = 0 to $n-1$ **do**
    for i = 1 to $n-j$ **do**
      $Ap\Psi_i^{i+j}$ =Gen_Apropos_Alp_trees(i,i+j,$\mathcal{N}$)
      θ_i^{i+j} = Best_of($Ap\Psi_i^{i+j}$)
  Result = θ_1^n
end

Gen_Apropos_Alp_trees uses equation (5) to generate all apropos trees.

2.3.1 Binary alphabetic trees with monotone tree cost

Lemma 2.6 *The number of apropos alphabetic binary trees on n leaves is equal to $n-1$ if the tree cost is monotone.*

Proof For monotone binary trees, $\wedge(\{\theta_i^{i+j}\} \times (\dot{=}.Ap\Psi_{i+j+1}^m))$ drops out of equation (5), thus

$$Bi\Psi_i^m = \bigcup_{j=0}^{m-i-1} (\wedge\{\theta_i^{i+j}, \theta_{i+j+1}^m\}) \qquad \text{for } i \leq m-1 \tag{7}$$

where $Bi\Psi_i^m$ denote the set of apropos alphabetic binary trees.

$$\Phi_d = \begin{cases} d-1 & \text{if } 2 \leq d \leq n \\ 1 & \text{if } d = 1 \end{cases} \tag{8}$$

Hence for n leaves $\Phi_n = n-1$ ∎

Lemma 2.6 can be exploited to generate optimal alphabetic binary trees in $O(n^3)$ time complexity as follows.

Algorithm 2.3 Find_Best_Alphabetic_tree_Binary ($\mathcal{N}$)
$\mathcal{N}$ is given set of n leaf nodes with weights
begin
  for j = 1 to $n-1$ **do**
    for i = 1 to $n-j$ **do**
      $Bi\Psi_i^{i+j}$ =Gen_Apropos_Binary_Alp_Trees(i, i+j, $\mathcal{N}$)
      θ_i^{i+j} = Best_of($Bi\Psi_i^{i+j}$)
  Result = θ_1^n
end

For each i we generate $n-i$ optimal binary trees on subset of i leaf nodes. This has to be done n times, thus the time complexity of algorithm 2.3 is

$$\sum_{i=1}^{i=n-1} i(n-i) = n^3/6 - n/6 = O(n^3).$$

The number of distinct alphabetic binary trees on n leaf nodes is given by the Catalan number $\begin{pmatrix} 2(n-1) \\ (n-1) \end{pmatrix}/n$. The monotonicity of tree cost has reduced the number of trees to be considered significantly, resulting in a polynomial time algorithm for finding an optimal alphabetic binary tree. This remains true irrespective of other characteristics of the weights, tree cost function and combining functions. Previous researchers [8, 7, 4, 11, 9, 2] have generated optimal alphabetic binary trees, restricting leaf weights and/or some parameters of the combining function to be integer. Our algorithm generates optimal alphabetic trees for monotone tree costs in $O(n^3)$ without any such restriction. Such trees include all *additive* and *minimax* trees described in Section 1.

2.3.2 Nonbinary alphabetic trees with monotone cost

For nonbinary trees, we need the splitting term in equation (5), i.e., we not only have the term $\wedge\{\theta_i^{i+j}, \theta_{i+j+1}^m\}$, but also the term $\wedge(\{\theta_i^{i+j}\} \times (\doteq .Ap\Psi_{i+j+1}^m))$. Because of this, the number of apropos trees increases from $n-1$ to $2^{n-1}-1$. The question here is: *For monotone nonbinary alphabetic trees, can we reduce the size of* Ψ_{i+j+1}^m *?* If we limit the number of apropos trees in Ψ_{i+j+1}^m to be polynomial in number $m-i-j$ of sinks, we can then compute apropos trees for each ordered subset of sinks before generating apropos trees for larger number of sinks. If the number of apropos alphabetic trees on n sinks was bounded by, say $O(n^k)$, using an algorithm similar to algorithm 2.3, we will be able to generate optimal alphabetic trees in $O(n^{k+2})$. Such size reductions may be gained with greater insight into the application.

In the next section, we will study the fanout optimization problem and show how such reduction in the number of apropos trees is obtained for the nonbinary case.

3 Alphabetic Fanout Optimization

Alphabetic fanout optimization may be formulated as a 2-WEIGHTED_TREE_GENERATION problem as follows.

Problem 3.1 ALPHABETIC_FANOUT_OPTIMIZATION (ALPFANOUT)

- **Instance:**
  1. *A set of n sinks* $(L_1, L_2, \ldots, L_n)$ *in a given, fixed order with 2-dimensional weights* $W_{L_i} = (r_{L_i}, \gamma_{L_i})$ *where* r_{L_i} = *required time at* L_i *and* γ_{L_i} = *input load of sink* L_i.
  2. *A 2-dimensional combining function* (f_{req}, f_{load}) *which combines l nodes generating an internal node* I_m *with weight vector* W_{I_m} *where,*

$$\begin{aligned} r_{I_m} &= f_{req}(W_{child^1_{I_m}}, W_{child^2_{I_m}}, \ldots, W_{child^l_{I_m}}) \\ &= min_i(r_{child^i_{I_m}}) - \beta_{buf} \sum_i \gamma_{child^i_{I_m}} - \alpha_{buf} \qquad (9) \\ \gamma_{I_m} &= f_{load}(W_{child^1_{I_m}}, W_{child^2_{I_m}}, \ldots, W_{child^l_{I_m}}) \\ &= \gamma_{buf} \qquad (10) \end{aligned}$$

     $\beta_{buf}, \alpha_{buf}$ *and* γ_{buf} *denote drive resistance, internal delay and input load of the buffer, respectively.*
  3. *A tree cost function* $C(T) = r_{I_m}$ *where* I_m *is the root of the fanout tree T.*
- **Problem:** *Generate a tree T such that the cost (required time at the root) is maximum and the tree has no internal wire crossings*[7].

The combining function in ALPFANOUT corresponds to the *library* delay model. The problem can be easily modified to consider other delay models by restricting the values of different parameters. For example, if unit fanout delay model is used, $\alpha_{buf} = 0$, $\beta_{buf} = 1$ and $\gamma_{buf} = 1$.

Lemma 3.1 *Alphabetic fanout trees are monotone*[8].

[7] Our method can handle more than one type of buffer and/or limit the number of fanouts per buffer. However, in this paper, for sake of simplicity, we will assume only one buffer with no fanout limit.

[8] As a matter of fact, this holds for nonalphabetic fanout trees as well

Proof For the ALPFANOUT problem, f_{load} is constant while f_{req} is a monotone nondecreasing function of the required time and a monotone decreasing function of the load of the child nodes. Increasing the required time of any node in the tree can only result in equal or higher required time at the root, while the load of the node remains constant. Since maximizing required time for any subtree is the same as maximizing the tree cost of the subtree, alphabetic fanout trees are monotone. ∎

As a result of the above lemma and theorem 2.3, we can use algorithm 2.2 with $O(2^n)$ complexity for solving the ALPFANOUT problem optimally. Let us denote the specialization of algorithm 2.2 for the ALPFANOUT problem as ALPFANOUT_ALG. However, before applying this algorithm, we analyze the ALPFANOUT problem in order to simplify the problem space and reduce the number of apropos trees which need to be considered in order to generate an optimal alphabetic fanout tree.

3.1 Simplifying the problem space

The delay through a buffer is given by $\alpha_{buf} + \beta_{buf} \sum_{j \in FO_{buf}} \gamma_j$ where FO_{buf} denotes fanouts of the buffer . The wiring load is estimated dynamically based on the number of fanouts. This load can then be included in γ_j. Using this mechanism, the required time at an intermediate buffer k is given by $r_k = min_{j \in FO_k}(r_j) - \alpha_{buf} - \beta_{buf} \sum_{j \in FO_k} \gamma_j$.

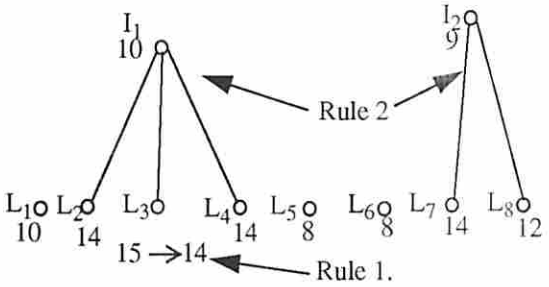
The fanout tree generating rules given below do not undermine the optimality of the algorithm but improve its efficiency. Given a set of original sinks, these rules generate a modified set of sinks. An additional requirement for these rules to be valid is that the $\gamma_j \geq \gamma_{buf}$ for each sink L_j. This requirement is satisfied in most cases as input capacitance of the inverter is less than most other gates. Similar rules for the unit delay model were proposed in [2].

Lemma 3.2 *Given n internal or sink nodes, application of the following rules does not undermine optimality of* ALPFANOUT_ALG.

Rule 1: *If* $n \geq 1$ *and* $r_i \geq max(r_{i-1}, r_{i+1}), 1 \leq i \leq n$, *make* $r_i = max(r_{i-1}, r_{i+1})$.

Rule 2: *If* $max(r_i, r_{i+s+1}) \leq min(r_{i+1}, \ldots, r_{i+s}) - \alpha_{buf} - \beta_{buf} \sum_{j=1}^{s} \gamma_{i+j}$, *replace the sequence* $L_{i+1}, ..L_{i+s}$ *of nodes by one node with required time* $min(r_{i+1}, ...r_{i+s}) - \alpha_{buf} - \beta_{buf} \sum_{j=1}^{s} \gamma_{i+j}$.

Proof We only give an outline of the proof. Each rule takes a set of sinks and produces a modified set of sinks which is smaller. For each rule, we consider an optimal tree on original set of sinks. The proof consists of indicating how this optimal tree can be altered



a) Original set of sinks and application of rule 1 and rule 2.

Figure 3: Illustration of Rule 1 and Rule 2

to obtain a tree with modified set of sinks such that the required time at the root is not decreased. ∎

Application of these rules is illustrated in Figure 3 a). To simplify the presentation, we assume the unit fanout delay model, i.e., $\alpha_{buf} = 1, \beta_{buf} = 1$ and $\gamma_j = 1$ for all j. As shown in the figure let the required time at sinks be given by vector $\mathbf{r}$ = (10, 14, 15, 14, 8, 8, 14, 12). Rule 1 is applied on L_3, while rule 2 is applied on (L_2, L_3, L_4) and on (L_7, L_8), generating internal nodes I_1 and I_2, respectively. At this stage, we reach an *impasse* as neither of the rules can be applied Instead, if we had used unit delay model, these rules would have generated optimal alphabetic fanout trees in O(n) time complexity without encountering any impasse. For the *unit fanout* or the *library* delay model, impasse is likely to occur, in which case we resolve the impasse by resorting to the tree generation part of ALPFANOUT_ALG. Application of these rules, however, has reduced the solution space by reducing the number of sinks from 8 to 5.

3.1.1 Rule specialization for simpler delay models

The above rules can be simplified (and strengthened) for the two simpler delay models as follows.

Let the ordered sink list be $(L_1, ..L_n)$.

Fanout tree generating rules for unit delay model:

1. If $n > 1$ and $r_i \geq max(r_{i-1}, r_{i+1})$, $1 \leq i \leq n$, make $r_i = max(r_{i-1}, r_{i+1})$.
2. If $max(r_i, r_{i+s+1}) \leq min(r_{i+1}, ...r_{i+s}) - 1$, replace $L_{i+1}, ..L_{i+s}$ nodes by node with required time $min(r_{i+1}, ...r_{i+s}) - 1$. Where $s \leq t$.
3. If $r_i = r_{i+1} = r_{i+2} = ... = r_{i+t-1} \geq r_{i+t} + 1$, replace $L_i, ..L_{i+t-1}$ nodes by node with required time $r_{i+t-1} - 1$

Fanout tree generating rules for unit fanout delay model:

1. If $n \geq 1$ and $r_i \geq max(r_{i-1}, r_{i+1})$, $1 \leq i \leq n$, make $r_i = max(r_{i-1}, r_{i+1})$.
2. If $max(r_i, r_{i+s+1}) \leq min(r_{i+1}, ...r_{i+s}) - s$, replace $L_i, ..L_{i+s}$ nodes by node with required time $min(r_{i+1}, ...r_{i+s}) - s$.
3. If $r_i = r_{i+1} = r_{i+2} = ... = r_{i+t-1} \geq r_{i+t} + t$, replace $L_i, ..L_{i+t-1}$ nodes by node with required time $r_{i+t-1} + t$

3.2 Reducing the number of apropos trees

As seen from equation (5), because of the tree splitting term, we must generate all subtrees in $\Psi_i^j; \forall i, j \leq n$. This number can be reduced significantly using the following theorem.

Let R_T, L_T, req_T and $load_T$ denote the minimum of the required times at the immediate children of the root of tree T, the cumulative load offered by immediate children of the root, the required time at the root of the tree, and the load at the root of the tree, respectively. Using this notation, the required time for a tree T generated by $\wedge(\{T_l\} \times (.\doteq .T_r))$ is thus given by

$$req_T = min(req_{T_l}, T_{T_r}) - \beta_{buf}(load_{T_l} + L_{T_r})$$

Definition 3.1 *Given two trees T' and T'' with $R_{T'} \geq R_{T''}$, we say*

- *T' and T'' are* non-inferior *with respect to each other if $(R_{T'} - R_{T''}) > \beta_{buf}(L_{T'} - L_{T''}) > 0$.*
- *T' is* superior *to T'' if $(R_{T'} - R_{T''}) > 0 \geq \beta_{buf}(L_{T'} - L_{T''})$.*
- *T' is* inferior *to T'' if $\beta_{buf}(L_{T'} - L_{T''}) \geq (R_{T'} - R_{T''}) > 0$.*

Theorem 3.3 *For fanout optimization, when generating the set of apropos trees on sinks j through m, it is sufficient to maintain a set of trees which are non-inferior with respect to each other, but superior to all other trees.*

Proof As per the notation proposed earlier, the set of apropos trees on sinks $j+1$ through m is denoted by $Ap\Psi_{j+1}^m$ and the set of all alphabetic trees on sinks $j+1$ though m is denoted by Ψ_{j+1}^m. When we split a tree $T_r \in \Psi_{j+1}^m$, it generates a forest F_r with minimum of the required time at the roots given by R_{T_r} and sum of loads at the roots given by L_{T_r}. According to tree generation procedure we have to join F with the best

tree $T_l = \theta_i^j$, while generating a tree $T \in \Psi_i^m$ [9]. The required time at the root of the tree T is given by

$$req_T = min(req_{T_l}, R_{T_r}) - \beta_{buf}(load_{T_l} + L_{T_r}) \tag{11}$$

Let us consider any two trees $T'_r, T''_r \in \Psi_{j+1}^m$. Without loss of generality, we assume $R_{T'_r} \geq T_{T''_r}$. From the definition 3.1, T'_r is either superior, inferior or non-inferior with respect to T''_r. We will now show that if T'_r is superior or inferior with respect to T''_r, we need to consider only one of the two.

T'_r is superior: Given $(R_{T'_r} - R_{T''_r}) > 0 \geq \beta_{buf}(L_{T'_r} - L_{T''_r})$, we need to prove the following:

$$min(req_{T_l}, R_{T'_r}) - \beta_{buf}(load_{T_l} + L_{T'_r}) > min(req_{T_l}, R_{T''_r}) - \beta_{buf}(load_{T_l} + L_{T''_r})$$

This is true from the definition of superior tree irrespective of the value of $req_{T_l}, load_{T_l}$ and β_{buf}, allowing us to ignore T''_r.

T'_r is inferior: Given $\beta_{buf}(L_{T'} - L_{T''}) \geq (R_{T'} - R_{T''}) > 0$, we need to prove the following:

$$min(req_{T_l}, R_{T'_r}) - \beta_{buf}(load_{T_l} + L_{T'_r}) < min(req_{T_l}, R_{T''_r}) - \beta_{buf}(load_{T_l} + L_{T''_r}) \tag{12}$$

From equation (12) we obtain the following.

$$min(req_{T_l}, R_{T'_r}) - min(req_{T_l}, R_{T''_r}) < \beta_{buf}(L_{T'_r} + L_{T''_r})$$

From the definition (3.1), this is true for each of the three cases, $T''_r \leq T'_r \leq T_l$, $T''_r \leq T_l \leq T'_r$, and $T_l \leq T''_r \leq T'_r$, allowing us to ignore T'_r.

Thus, from Ψ_j^m we need to consider only those trees which are non-inferior with respect to each other. ∎

Theorem 3.3 enables us to reduce the number of apropos trees for ALPFANOUT by only generating the set of non-inferior trees $Ni\Psi_i^m$. As we generate each tree structure, we compare it with the trees in the current $Ni\Psi_i^m$, deleting inferior trees from $Ni\Psi_i^m$ in the process. We introduce a rule which will exploit this fact.

Rule 3: During tree generation on sinks i through m, current tree T is added to $Ni\Psi_i^m$ only if T is non-inferior to all the trees in $Ni\Psi_i^m$. If T is superior to some trees currently in $Ni\Psi_i^m$, we remove those trees from $Ni\Psi_i^m$ before adding T.

The correctness of rule 3 follows from theorem 3.3.

[9] Actually, we might not always join F_r with a tree. while generating trees for Ψ_h^m, for example, we might split Ψ_i^m, resulting in the second argument of JOIN being a forest of two trees, one from Ψ_h^{i-1} and Ψ_i^{j-1} each. The theorem stills holds and can be proved with identical approach.

4 Experimental Results

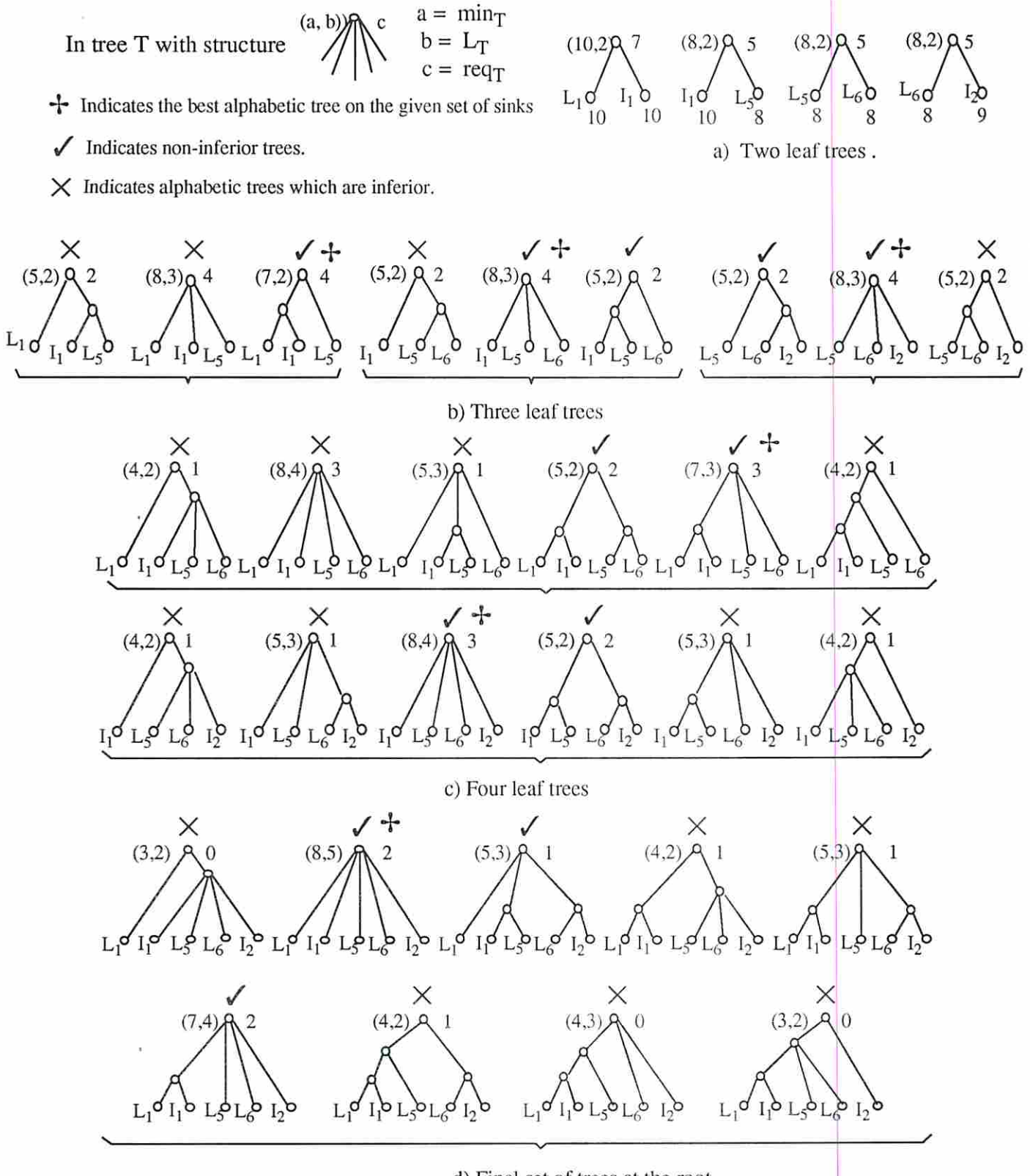
The algorithm ALPFANOUT_ALG was implemented in the *SIS* environment. Mapped networks were optimized with ALPFANOUT_ALG after deriving an order on the fanout of each node using the ordering mechanism specified.

Algorithm 4.1 AlpFanout_Alg (Γ, Θ, Ω)
Γ *is an optimized mapped Boolean network*
Θ *is a vector of required times*
Ω *is the ordering mechanism*
begin
  for each node $n \in \Gamma$ (in preorder) **do**
    L = Order_sinks (Γ, n, Ω)
    L' = Reduce_sinks (L, Θ)
    η = Generate_best_AlpFanout_tree (n, L')
    update_network (Γ, n, η)
  end
end

Reduce_trees reduces the number of sinks using *rule 1* and *rule 2*. Given a set of ordered sinks, Generate_best_AlpFanout_tree returns an optimal fanout tree on the ordered sinks L' using *rule 3* and the apropos tree generation equation (5). Application of *rule 3* during tree generation significantly improves efficiency of ALPFANOUT_ALG. This is illustrated in Figure 4. Continuing with the previous example, we want to generate the optimal fanout trees on the modified set of sinks $(L_1, I_1, L_5, L_7, I_2)$. From these 5 sinks, using the tree generation mechanism, we generate all alphabetic trees on every subset of ordered sinks of size 2 to 5. According to *rule 3*, every tree in the list of current apropos tree should be *non-inferior* to all others. Inferior trees are dropped from the current list of trees and are excluded from further consideration.

For this particular example, from equation (3), there are 903 alphabetic fanout tree structures. Due to the monotone tree cost function, the number of apropos trees (i.e., trees that must be considered to find the optimal solution) is 127. Use of rules 1 and 2 reduces the number of sinks to 5, hence lowering the number of apropos trees to 31. Rule 3 eventually reduces the total number of apropos trees on final set of sinks to 9. Note that rule 3 also reduces the number of apropos trees during the generation of subtrees.

Previous algorithms considered sinks with differing polarities separately. However, to maintain alphabetic order on the sinks irrespective of their polarities we used the following mechanism. For every set of the sinks, we generated two fanout trees: one with



21

Figure 4: Illustration of ALPFANOUT_ALG

positive polarity at the root and the other with negative polarity at the root. Apart from this, during every JOIN operation, we only joined the subtrees with identical polarities.

The ordering on the sinks were derived using different mechanisms.[10] The PLACE_ORDER imposes a sink ordering based on the sink positions derived from a global placement solution for the Boolean network. The idea is that since the placement solution captures the connectivity structure of the network and the addition of fanout tree does not modify the network structure to a great extent, we can rely on this placement solution for estimating the relative positions of the sinks after fanout optimization and placement. By using the placement information, due to the non-crossing property of the alphabetic trees, we will preserve the crossing number of the network during fanout optimization. Here we are trading performance for improved circuit routability.

The TOPOLOGICAL_ORDER constructs a sink ordering based on the amount of logic sharing among the transitive fanout cones of the sinks. That is, two sinks whose output cones overlap a lot, are placed near one another in the order. The goal is to put these two sinks near one another in the fanout tree.

The REQUIRED_ORDER generates a sink ordering based on the required times of the sinks. This option has been adopted by other researchers in the field. However, our fanout optimization algorithm is provably better than other algorithms as shown in the previous sections. The rationale behind this ordering is that, in general, it is desirable (from the performance point of view) to put sinks with similar required times at the same depth in the fanout tree.

Table 1 compares the ALPFANOUT using the three ordering options with SISFANOUT [15]. As can be seen, on average we do better then SISFANOUT in all cases. The best performance results are obtained with the REQUIRED_ORDER, and the best routing (smallest chip area) is obtained with the PLACE_ORDER or TOPOLOGICAL_ORDER.

SISFANOUT tries a number of algorithms (e.g., LT_trees, Two_Level, Balanced, etc) at each node and picks the best fanout solution. Therefore, we generated a new set of results where we added ALP_FANOUT as a fanout algorithm to the SISFANOUT tool. Table 2 shows the results which are better than those of ALPFANOUT or SISFANOUT alone (as is expected).

The stand-alone ALPFANOUT runtime are quite comparable with those of the stand-alone SISFANOUT. For example, on a Sun Sparc Station 2, for $C1355$, $C1908$, and $C6288$, ALPFANOUT took 52.9, 55.6, and 446.1 seconds versus the SISFANOUT time which were 47.7, 52.1, and 369.0 seconds.

[10] Depending on the secondary objective of fanout optimization (routing congestion, power efficiency, etc), other ordering mechanisms can be proposed and implemented. A strong point about our work is that it allows various optimization criteria to be considered during the fanout optimization with little or no degradation of the circuit performance.

circuit	SisFan		Stand-Alone version of AlpFanout					
			Req_Order		Place_Order		Top_Order	
	chip area	place delay	chip area	place delay	chip area	place delay	chip area	place delay
C1355	3241.2	55.56	2663.0	50.21	2592.0	55.80	2643.0	53.32
C1908	3270.4	73.31	3316.0	68.94	3064.3	73.25	3058.8	74.42
C2670	4845.5	60.12	4141.1	55.45	4143.2	58.05	4211.3	60.01
C3540	8855.5	128.05	8852.5	109.32	8873.7	125.42	8879.8	130.25
C432	1518.8	65.69	1391.1	66.01	1320.6	67.72	1326.7	71.11
b9	857.3	19.20	775.2	21.02	748.7	20.65	759.7	21.02
k2	8095.9	70.69	8035.6	73.97	7477.5	72.58	7483.0	76.67
rot	5296.9	54.60	4613.5	49.04	4202.8	56.54	4266.6	49.82
C7552	15100	150.59	14395	115.86	13753	131.16	13898	126.00
dalu	8852.4	133.39	9141.3	122.98	8329.6	129.07	8387.4	124.29
t481	5333.4	62.18	5489.0	58.90	5258.0	56.23	5355.3	61.06

Table 2: Comparison between fanout optimization in Sis and AlpFanout using different ordering mechanisms.

circuit	SisFan		AlpFanout Integrated with SisFan					
			Req_Order		Place_Order		Top_Order	
	chip area	place delay	chip area	place delay	chip area	place delay	chip area	place delay
C1355	3241.2	55.56	2867.3	49.26	2743.3	51.59	2717.8	50.97
C1908	3270.4	73.31	3213.9	71.88	3075.3	68.35	3108.1	69.35
C2670	4845.5	60.12	4392.2	52.63	4124.1	55.05	4151.7	60.27
C3540	8855.5	128.05	9083.5	104.51	8843.4	113.17	8767.4	111.97
C432	1518.8	65.69	1484.7	58.61	1473.8	60.62	1464.1	58.83
b9	857.3	19.20	802.6	18.94	782.5	18.40	777.0	18.31
k2	8095.9	70.69	8013.7	69.57	7797.6	70.29	7789.4	65.55
rot	5296.9	54.60	4753.9	48.43	4683.7	47.35	4713.5	46.99
C7552	15100	150.59	14526	106.37	14541	110.95	14446	104.13
dalu	8852.4	133.39	8645.8	124.10	8654.9	132.48	8633.6	153.76
t481	5333.4	62.18	5279.9	58.07	5172.9	56.82	5151.0	57.11

Table 3: Comparison between fanout optimization before and after integrating AlpFanout in Sis fanout optimization and effect of different ordering mechanisms.

5 Concluding Remarks and Future Work

We proposed an efficient fanout optimization algorithm which improves circuit performance while honoring an order restriction on the sinks. This ordering is derived based on an early global placement, analysis of the network structure, or required time constraints at the primary sinks. We introduced the concept of apropos trees which allows us to significantly reduce the solution space for generating optimal alphabetic trees. Using this concept we show that optimal alphabetic binary trees on n leaf nodes can be generated in $O(n^3)$ time complexity for monotone cost trees. This is the most general solution to the optimal alphabetic tree construction problem that we know of.

For the alphabetic nonbinary tree construction, we proposed a set of rules that reduce size of the solution space while maintaining the optimality. We applied these techniques to the problem of generating optimal alphabetic fanout trees where we introduce the notion of non-inferior set of fanout trees and use this to obtain further reduction in the solution space.

We tried a number of ordering mechanisms and obtained an average of 8-10% improvement in performance and 4-7% reduction in chip area compared to the SIS fanout optimization tool.

Our current results motivate us to apply the the alphabetic tree construction theory developed here to the logic decomposition and kernelization procedures. We hope that this theory will provide an effective way of incorporating routing issues (e.g., wire crossing, congestion) into logic synthesis.

Acknowledgements

This work was supported in part by the NSF's Research Initiation Award under contract No. MIP-9211668.

References

[1] C. L. Berman and J. L. Carter. The fanout problem: From theory to practice. In C. L. Seitz, editor, *Advanced Research in VLSI: Proceedings of the 1989 Decennial Caltech Conference*, pages 69–99. MIT Press, May 1989.

[2] D. Coppersmith, M. M. Klawe, and N. J. Pippenger. Alphabetic minimax trees of degree at most t. *SIAM Journal of Computing*, 15:189–192, 1986.

[3] Shimon Even. *Graph Algorithms.* Computer Science press, Rockville, Maryland, first edition, 1979.

[4] E. N. Gilbert and E. F. Moore. Variable-length binary encoding. In *Bell Systems Technical Journal*, volume 38, pages 933–968, 1959.

[5] M. C. Golumbic. Combinatorial merging. *IEEE Transactions on Computers*, 25(11):514–526, 1976.

[6] M. M. Klawe H. J. Hoover and N. J. Pippenger. Bounding fanout in logical networks. *Journal of the Association for Computing Machinery*, 31(1):13–18, January 1984.

[7] T. C. Hu, D. J. Kleitman, and J. K. Tamaki. Binary trees optimum under various criteria. *SIAM Journal of Applied Mathematics*, 37(2):246–256, October 1979.

[8] T. C. Hu and A. C. Tucker. Optimal computer search trees and varaible-length alphabetical codes. *SIAM Journal of Applied Mathematics*, 21(4):514–532, December 1971.

[9] D. G. Kirkpatrick and M. M. Klawe. Alphabetic minimax trees. *SIAM Journal of Computing*, 14(3):514–526, 1985.

[10] J. M. Kleinhans, G. Sigl, F. M. Johannes, and K. J. Antreich. GORDIAN: VLSI placement by quadratic programming and slicing optimization. *IEEE Transactions on Computer-Aided Design*, CAD-10:356–365, March 1991.

[11] D. Knuth. *The Art of Computer Programming.* Addison-Wesley, 1973.

[12] D. E. Knuth R. L. Graham and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science.* Addison-Wesley Publishing Company, Reading, Massachusetts, first edition, 1988.

[13] K. J. Singh and A. Sangiovanni-Vincentelli. A heuristic algorithm for the fanout problem. In *Proceedings of the 27th Design Automation Conference*, pages 357–360, June 1990.

[14] A. Srinivasan, K. Chaudhary, and E. S. Kuh. RITUAL: An algorithm for performance-driven placement of cell-based IC's. In *Proceedings of the Third Physical Design Workshop*, May 1991.

[15] Herve Touati. *Performance Oriented Technology Mapping.* PhD thesis, University of California, Berkeley, 1990.