Hierarchical Buffered Routing Tree Generation

Amir H. Salek, *Member, IEEE*, Jinan Lou, *Member, IEEE*, and Massoud Pedram, *Fellow, IEEE*

Abstract--**This paper presents a solution to the problem of performance-driven buffered routing tree generation for VLSI circuits. Using a novel bottom-up construction algorithm and a local neighborhood search strategy, our polynomial time algorithm finds the optimum solution in an exponential-size solution subspace. The final output is a buffered rectilinear Steiner routing tree that connects the driver of a net to its sink nodes. The two variants of the problem, i.e., maximizing the required time at the driver subject to a maximum total area constraint and minimizing the total area subject to a minimum required time at the driver constraint, are handled by propagating three-dimensional solution curves during the construction phase. Experimental results demonstrate the effectiveness of our algorithm compared to other techniques.**

I. Introduction

The consideration of the effects of interconnect delay and area has become a crucial factor in the design of ultra-dense, high speed integrated circuits. In an industry where higher performance design brings substantial advantage over the competition, more and more time and resources are being spent on making faster chips through careful optimization of many design aspects, especially interconnect planning and optimization. In particular, the problem of constructing a buffered routing tree has emerged as a critical design problem.

The first part of this paper presents a new algorithm *FANROUT* that simultaneously solves the fanout optimization and routing tree construction problems. Both of these design tasks are difficult optimization problems and have considerable impact on the circuit delay and area. Fanout optimization is effective because it boosts the transmitted signal via the insertion of sized buffers whereas performance-driven routing generation is effective because it generates interconnect structures that deliver the signal to critical sinks faster. In conventional design flows, these two tasks are often performed in a sequential manner, i.e., a solution made by one optimization step becomes the input to the other. By solving the unified problem, i.e., generating a buffered routing tree for a set of sinks and a driver, the intrinsic interactions between the design steps are captured and higher quality results are produced by a systematic search in a much larger solution space. This type of solution technique is referred to as a *unification-based approach* in [Pe98].

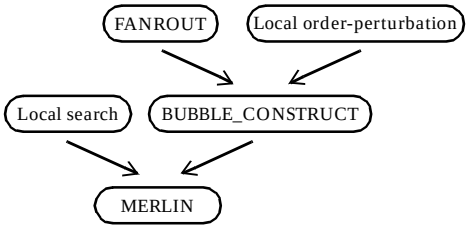

Fig. 1. Structure of MERLIN.

Similar to many other dynamic-programming based algorithms, FANROUT is only optimal with respect to a given order on its input objects (in this case the net sinks). This shortcoming is addressed in this paper by introducing a new technique called *local order-perturbation* which is used to enhance FANROUT. The resulting algorithm, *MERLIN*, is less sensitive to the input sink order with the cost of having a reasonably more complex computation.

The core optimization engine of MERLIN, called *BUBBLE_CONSTRUCT*, optimally solves the simultaneous routing and buffer insertion problem for a local neighborhood around an initial sink order. It recognizes the similar subsolutions among the members of the neighborhood in order to maintain the polynomial complexity of the algorithm. Although a complete buffered routing structure is not generated for every member of the neighborhood, the sink order that results in the best buffered routing structure is automatically chosen from among the members of the neighborhood. The outer optimization part of MERLIN (see Fig. 1) is an iterative technique based on a *local neighborhood search* strategy [Ya92].

Both FANROUT and BUBBLE_CONSTRUCT generate and propagate three-dimensional required time, load, and total area solution curves in a bottom-up fashion. In the three-dimensional solution curves, the load and the required time dimensions ensure the validity of the dynamic-programming principle [Be57] for solving the problem whereas the total area allows the user to solve the problem for either one of the following two variants: I) minimizing the required time subject to an area constraint or II) minimizing the area subject to a required time constraint.

The technique presented in this paper[1] offers the following advantages compared to prior methods:

- full integration of fanout optimization and routing tree generation using a dynamic-programming method,
- employment of a novel *local order-perturbation technique*

This research was sponsored in part by the NSF PECASE award number MIP-9628999.

A. H. Salek is with PMC-Sierra, Inc., Santa Clara, CA 95054 USA.

J. Lou is with Synopsys, Inc., Mountain View, CA 94043 USA.

M. Pedram is with the Department of Electrical Engineering-Systems, University of Southern California, Los Angeles, CA 90089 USA.

[1] The initial versions of the presented techniques have appeared in [SLP98] and [SLP99].

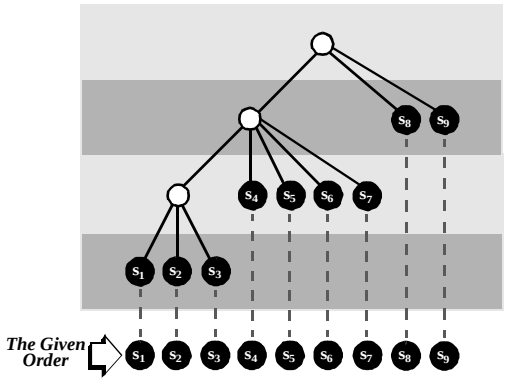

Fig. 2. An LT-Tree Type-I for a net with 9 sinks.

that enables the optimization engine to find (in polynomial time) the best buffered routing tree structure in an exponential-size sink order neighborhood of the initial order,

- propagation of *three-dimensional solution curves* that allows the algorithm to trade-off required time with total area and vice versa,
- definition and use of *Cα-Tree and *P-Tree structures* that expand the power of the optimization algorithms, resulting in highly optimized solutions,
- employment of the *local neighborhood search strategy* along with the core optimization algorithm to find the best solution in the neighborhood of the input sink. The resulting method is less sensitive to the initial order.

The remainder of the paper is organized as follows. In section II, prior work is given. Section III presents the problem formulation. Sections IV and V introduce FANROUT and the local order-perturbation technique. In section VI, MERLIN and its constituting elements are presented and discussed. Finally, sections VII and VIII give the experimental results and the concluding remarks, respectively.

II. Prior Work

A. Fanout Optimization

Fanout optimization, an operation performed in the logic domain, addresses the problem of distributing an electrical signal to a set of sinks with known loads and required times so as to maximize the required time at the signal driver (root of the net). Interconnect delays are either ignored or loosely modeled in this operation because the sink locations are not known at this stage. The general form of this problem is NP-hard [To90]; however, its restriction to some special families of topologies is known to have polynomial complexity.

Among many fanout optimization techniques - e.g., [Go76], [BCD89], [SS90], and [VP93] - the one proposed by [To90] has been proven to be very effective. That algorithm introduces a special class of tree topologies, called *LT-Tree*, for which the fanout problem is solved optimally with respect to a given order of sinks using dynamic programming. An *LT-Tree of type-I* is a tree that permits at most one internal node among the immediate children of its internal nodes and also does not allow any left sibling for the internal nodes (see Fig. 2).

Touati proposed a dynamic-programming based algorithm

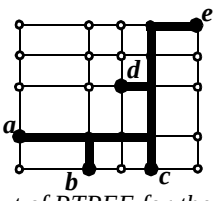

Fig. 3. An output of PTREE for the "dcba" order.

for the fanout optimization problem where the buffer structure is restricted to the LT-Tree topology and sinks with larger required times are placed farther from the root of the tree. The algorithm first sorts the sinks in their non-decreasing required time order and then, starting from the least critical sink, it enumerates all the left-most grouping of the sinks to be driven by a buffer. Finally for each grouping, it enumerates all possible ways of adding either zero or one buffer to drive the leftmost subset of the sinks. Touati gives sufficient conditions for the LT-Tree construction algorithm *LTTREE* to be optimal. For more details, see [To90].

Lemma 1: LT-Tree construction algorithm shows $O(n^2)$ complexity where n is the number of sink nodes [To90].

B. Routing Tree Construction

Performance-driven interconnect design, an operation performed in the physical domain, addresses the problem of connecting a signal driver to a set of sinks with known loads, required times and locations so as to maximize the required time at the driver. [CHKM96] gives a comprehensive review of the algorithms for solving this problem.

The inherent complexity of the problem has forced the researchers to focus on heuristic solutions and/or impose constraints on the structure of resulting interconnect. Among the recent works in this area, the algorithm presented by Lillis et al. in [LCLH96] has been shown to be quite effective. The authors proposed the Permutation-Constrained Routing Tree or *P-Tree* structure and solved the above problem with respect to the P-Tree structure; see Fig. 3 for an example. This approach consists of two major phases: I) heuristically finding a proper order for the sinks and II) generating the routing structure based on the order. The second phase of the algorithm is referred to as *PTREE* throughout this paper. Given an order for the sink nodes, PTREE finds the optimal embedding of the net into the *Hanan grid*[2] using a dynamic-programming approach. In PTREE, the intermediate routing solutions are stored in the form of two dimensional, non-dominated solution curves of total area versus required time for every *Hanan point*[3].

Lemma 2: For a given order on the sinks and with the restriction that the Steiner points lie on the Hanan points, PTREE computes the set of all rectilinear Steiner trees with non-dominated required time and total capacitance [LCLH96].

Lemma 3: If the individual capacitive values of wires and gate inputs are polynomially bounded integers or can be mapped to such with sufficient precision, then PTREE has $O(n^5q)$ *pseudo-polynomial complexity* (see [GJ79]), where n is

[2] The Hanan grid of a net is defined as the grid formed by the intersection of horizontal and vertical lines running through the terminals of the net [Ha66].

[3] Hanan points of a net are the vertices of the Hanan grid of the net.

the number of sink nodes and q is the maximum number of distinct load values [LCLH96].

Corollory 1: If the PTREE function is called with α sinks and uses k candidate locations instead of Hanan points, its complexity is $O(k\alpha^3 q)$.

C. Other Related Works

Lukas van Ginneken in [Gi90] proposed an algorithm to insert buffers on appropriate internal nodes of a given routing tree in order to maximize the required time at the driver. The application of van Ginneken's method after constructing the routing tree is usually more effective than applying fanout optimization followed by routing tree generation [SLP98].

The first attempts to combine fanout optimization and routing generation were presented in [OC96a] and [LCL96]. In [OC96a], the authors proposed a combination of A-Tree routing generation [CLZ93] and van Ginneken's buffer insertion [Gi90] methods. They later extended the work in [OC96b] to include wire sizing as well. Their algorithm takes the placement information of the source and the sinks in addition to the signal required times and heuristically generates a buffered routing structure that maximizes the required time at the source of the net. In these works, the subtrees are combined using a weighted addition function with a user-specified parameter to heuristically decide which two subtrees are to be merged. The algorithms in [OC96a] and [OC96b] have no guarantee of optimality. In [LCL96], Lillis et al. introduced a new version of PTREE which systematically solves the integrated problem of buffering and routing. That algorithm, called *B_PTREE* in the rest of this paper, uses a dynamic-programming formulation and generates three dimensional solution curves. Similar to PTREE, B_PTREE is optimal only with respect to a given sink order.

III. Problem Formulation

Given a net with $n+1$ pins, the problem is to drive the set of sink pins, $S=\{s_1, s_2, \ldots, s_n\}$, by the driver of the net s via a buffered routing structure that satisfies a combination of the maximum required time at the root and the minimum total area constraints. The area constraint can be stated in the form of total buffer area or total capacitance; the total capacitance is considered as a metric indicating the total buffer and interconnect area. More specifically, the problem may be stated in two ways: I) maximize the required time subject to an area constraint and II) minimize the area subject to a required time constraint.

The following information is provided as input:

1. The position of the source $s=(s^x,s^y)$ where s^x and s^y are the horizontal and vertical coordinates of s.
2. The properties of each sink node $s_i=(s_i^x,s_i^y,s_i^l,s_i^r)$ for $1\le i\le n$ where s_i^x and s_i^y are the horizontal and vertical coordinates, s_i^l is the capacitive load, and s_i^r is the signal required time of node s_i.
3. A library of buffers $B=\{b_1, b_2, \ldots, b_m\}$ containing m buffers with different strengths.

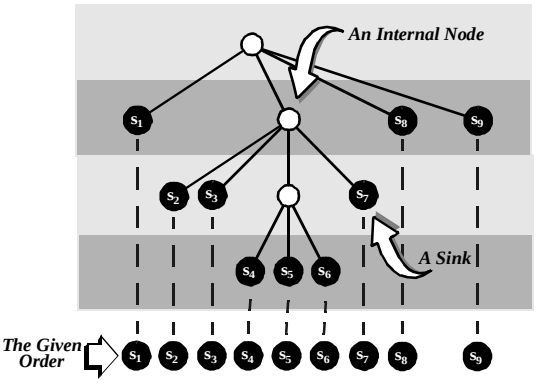

Fig. 4. A valid C4-Tree for $(s_1,s_2,\ldots,s_9)$.

4. A set of k candidate locations for placing the buffers $P=\{p_1, p_2, \ldots, p_k\}$.
5. A linear ordering of the sinks $\Pi=(s_1, s_2, \ldots, s_n)$.

There are many candidates for P; it can be the set of Hanan points [Ha66] (similar to what [LCLH96] has proposed) or a set of reserved buffer locations (identified after performing the initial placement step). Our experiments, in agreement with a conclusion made in [LCLH96], demonstrate that neither one of the above choices would significantly alter the final results as long the following two conditions are satisfied: I) k is large enough with respect to n and II) the candidate locations are distributed within the bounding box of the net with higher concentration in regions with a high density of sink pins.

IV. Order-Dependent Hierarchical Buffered Routing Tree Construction

This section presents FANROUT, an algorithm for solving the problem of simultaneous fanout optimization and routing generation. The resulting buffered routing tree contains a logical hierarchy that captures the hierarchical sink groups used during the construction. The hierarchy tree has a certain structure, which is formally defined below.

A. Cα-Trees

A desired property for a hierarchical algorithm is independence from any specific class of hierarchy graph structures. However, in many cases the complexity is so high that there is no choice but to restrict the solution space to a family of hierarchies. In this case, the problem is to identify and construct a set of structures that are consistent with the nature of the problem, both of which require a reasonable effort.

In this sub-section, a new class of structures, referred to as *Cα-Trees* (read as *si-alpha trees*), used to capture the hierarchy in the buffered routing construction algorithm is introduced.

Definition 1: A tree is a *degree-restricted alphabetic buffer chain tree* (*Cα-Tree*) for a given order of sinks $\Pi=(s_1,s_2,\ldots,s_n)$ if and only if:

- every internal node has at most one internal node among its immediate children,
- there is a depth-first traversal that visits the sinks in the $(s_1,s_2,\ldots,s_n)$ order,
- the maximum branching factor is α.

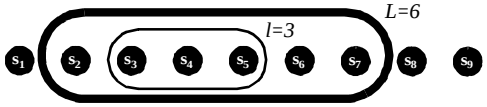

Fig. 5. Optimal Cα-Tree construction.

Fig. 4 illustrates an example for C4-Trees. In this figure the maximum branching factor is four and every internal node (shown by white circles) is connected to at most one other internal node while preserving the given order.

Lemma 4: In a Cα-Tree, the internal nodes construct a unique path (chain).

Proof: This is an immediate conclusion from Definition 1. ■

In this application, every internal node is a buffer, and in the resulting buffer chain, a more critical sink (considering both timing and physical information) tends to be connected closer (in terms of the number of intermediate stages) to the root.

Parameter α represents the maximum number of fanouts for every buffer or branching point. Our experience shows that even when no restriction is imposed on the maximum number of fanouts for each buffer, the maximum fanout count in the optimal buffer tree solution is usually bounded by a small number. That value is generally dependent on the characteristics of the cells (sink nodes) and the buffer library and not the problem size (number of sinks). Note that eliminating the parameter α from the definition does not cause the main structure and properties of Cα-Trees to breakdown. The only disadvantage is the longer (still polynomial) runtime needed for optimally constructing such a structure.

Although there are a large number of Cα-Trees for every sink order, the optimal Cα-Tree can be found in polynomial time using dynamic programming. Briefly, the optimal Cα-Tree for an ordered set of sinks is generated by starting from small *L*'s and combining every *L* neighboring sinks, until *L=n*. At every step, the best solutions for the sub-groups with length *l* (<*L*) are available - due to the bottom-up flow of the method - and are used to generate the solution for the length *L* sub-problem, see Fig. 5. Note that the final Cα-Tree structure satisfies the given sink order. This algorithm will be referred to as *CαTREE* in the rest of this paper.

Lemma 5: LT-Tree Type-I [To90] is a special case of Cα-Tree where $\alpha = +\infty$ and no internal node has a left sibling.

Proof: The proof directly follows the definitions of LT-Tree Type-I and Cα-Tree. ■

Note Cα-Trees can be relaxed with respect to the first property given in Definition 1, i.e., each internal node may have more than one internal node (but bounded by a certain parameter) among its immediate children. Although the optimal structure can still be achieved using dynamic programming, the complexity of the corresponding optimal construction algorithm is significantly higher.

B. FANROUT

FANROUT incorporates the Cα-Tree and P-Tree construction techniques into a unified framework such that the resulting routing structure is both Cα-Tree, in terms of the overall topology, and a P-Tree, in terms of the detailed physical

```
algorithm FANROUT( s, P, B, Π=(s_1 , s_2, ... , s_n) )
    INITIALIZATION
1.  for r = n downto 1
2.    foreach p∈P
3.      set Γ( 1, r, p ) = {The set of all non-inferior paths extended
        from p to s_r and driven with or without a buffer}
    CONSTRUCTION
4.  for L = 2 to n
5.    for R = n downto L
6.      for l = max(1, L-α+1) to L
7.        for r = R downto R-l+1
8.          foreach p∈P
9.            foreach γ ∈ Γ( l , r , p )
10.             set Δ = PTREE(P, {s_{R-L+1} , ... , s_{r-l} , γ , s_{r+1} , ... , s_R})
11.             foreach δ ∈ Δ
12.               set p' = Location of the root of δ
13.               foreach b ∈ B
14.                 set δ' = A buffered routing structure created by
                    driving δ by b located at p'
15.                 set < c, t, a > to the input capacitance, the input
                    required time and the area of δ', respectively
16.                 if < c, t, a > is a non-inferior solution in Γ(L , R , p')
17.                   insert < c, t, a > in Γ(L , R , p')
    EXTRACTION
18. find the solution ρ in Γ( n , n , s ) which best satisfies the constraints
19. retrieve the buffered routing tree structure ℜ of ρ by following the
    pointers stored during the generation of the solution curves
20. return ℜ
```

Fig. 6: Pseudo-code for FANROUT.

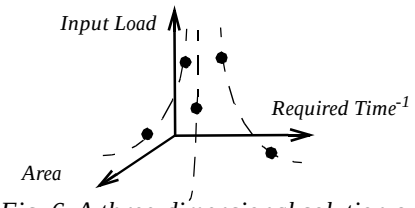

Fig. 6. A three-dimensional solution curve.

structure. FANROUT requires an ordering of the sinks and guarantees the optimality of the solution with respect to that ordering only. In the following paragraphs, the details of FANROUT are given.

FANROUT (see Fig. 6) is called with a set of parameters: *s*, *P*, *B*, and Π as defined in section III. It operates on three dimensional solution curves Γ(L, R, p) (see Fig. 6), each associated with a candidate buffer location *p* and a sub-group of sinks identified by variables *L* and *R*. *L* is the length of the sub-group and *R* indicates the position of the rightmost sink of the sub-group in Π . For example, if Π=(s_1, s_2, ..., s_9), Γ(*3, 7, p_i*) stores all the buffered routing structures that connect p_i to {s_5, s_6, s_7}.

In FANROUT, only non-inferior solutions - as defined below - are stored in the solution curves.

Definition 2: Suppose σ_1 and σ_2 are two different buffered routing structures that connect a candidate location to set of sinks. σ_2 is said to be inferior to σ_1, iff $load(\sigma_1) \le load(\sigma_2)$, $reqTime(\sigma_2) \le reqTime(\sigma_1)$, and $area(\sigma_1) \le area(\sigma_2)$.

As shown in Fig. 6, FANROUT consists of three main sections: *Initialization*, *Construction*, and *Extraction.* The Initialization section deals with creating and initializing solution curves corresponding to sub-problems consisting of only one sink, i.e., *L=1*. FANROUT is a dynamic-programming based technique and at each step it generates new curves by

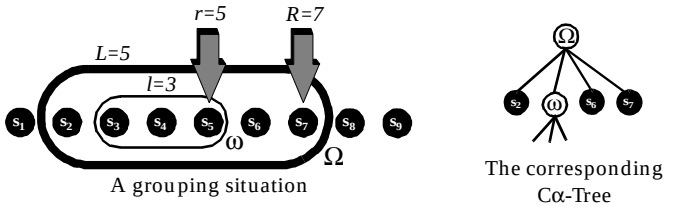

Fig. 7. An illustration for the grouping steps.

combining and manipulating already available curves for smaller sub-problems. In the Construction section, this bottom-up step is repeated until the solution curve for the main problem is found. Finally in the Extraction section, from among the solutions of the final Γ, the solution with the best trade-off between required-time and total area is chosen. At the end, the corresponding structure is generated by tracing back the pointers of the constituting sub-problems. The following is a detailed description of the algorithm.

1) Initialization: Before performing any operation, a set of solution curves are initialized in lines 1 through 3. In this part of FANROUT, sub-groups of length 1 are considered and the corresponding solution curves for every candidate buffer location and sink sub-group are initialized. These initial solutions consist of the minimum Manhattan distance paths from the candidate location p to the sink s_r . At the root of these paths, both options of inserting and not inserting a buffer are examined.

2) Construction: FANROUT starts by working on the groups consisting of only one sink, i.e., $L=1$, and proceeds until $L=n$. At each step, it constructs buffered routing structures that connect L neighboring sinks within Π. Line 4 enumerates all the possible values for L, and line 5 detects every legal sub-group Ω of Π that contains L sinks.

Every sub-group of sinks can potentially constitute an internal node in the final Cα-Tree structure; therefore, according to Definition 1, it can contain at most one other internal node (smaller sub-group) as its immediate child. During the process of grouping a set of L sinks, the algorithm considers cases in which a subset of sinks (call it ω) are already grouped; see Fig. 7 and lines 6 and 7 in the pseudo-code. That way, the Cα-Tree structure which captures the hierarchy of design is generated and maintained. In this context, the hierarchy implies that during the generation of a buffered routing structure, all the sinks are not processed at once; instead, a subset of sinks are combined together at any time in agreement with the Cα-Tree structure. Later, each combination is treated as one node in the next level of the hierarchy.

In line 6, the term *max(1,L-α-1)* ensures that Ω does not drive more than α other internal and sink nodes, following the third property of a Cα-Tree in Definition 1. In line 7, the term R to *R-l'+1* ensures that ω remains within Ω .

After line 7, it is known which sub-group ω is to be combined with which sink node(s) to generate the new sub-group Ω. However, there are many solutions associated with each ω. In fact, for every buffer candidate location p and sub-group ω there is a solution curve that is used by the merge operation; see Fig. 8. Line 8 enumerates all the candidate points, and line 9 retrieves the non-inferior solutions γ from a

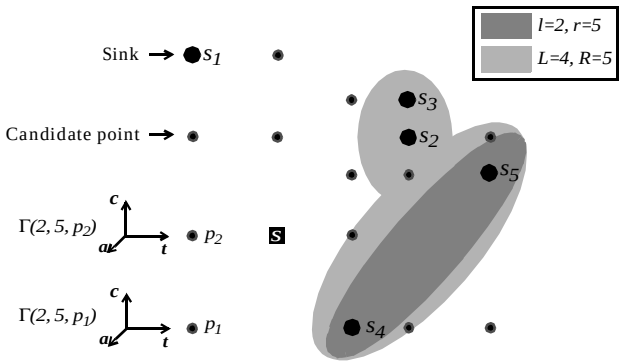

Fig. 8. Employing existing sub-solutions to generate larger sub-solutions.

solution curve of p which corresponds to ω.

In line 10, PTREE is called on the root of γ [4] and the remaining sinks in Ω in order to combine them by routing structures whose roots can be located at any candidate location. PTREE returns a collection of solution curves and stores them in Δ. Then, for every buffered routing structure in Δ, all the buffers in the library are used to drive its root, and the non-inferior combinations are stored in the corresponding solution curves; see lines 11 through 17. Along with every solution, a set of pointers is stored to be used during the extraction phase. The operations performed in lines 11 through 17 can be performed internally by a modified PTREE with no change in its worst case complexity. Hence, the complexity of those steps is not considered during the complexity analysis of FANROUT.

3) Extraction: The above bottom-up construction process continues until the solution curve for the whole problem, i.e., $L=n$, is generated. At that point, the solutions are stored in Γ(n, n, s) because they are rooted at s and are connected to all sinks. From among all the non-inferior solutions of Γ(n, n, s), the one which best satisfies the input constraints is chosen. The buffered routing structure corresponding to that solution is retrieved in lines 18 and 19 by following the stored pointers. Finally, in line 20 the best buffered routing structure is returned.

C. Quality and Complexity Analysis

FANROUT is an optimal polynomial algorithm based on a set of assumptions as explained previously and in the following lemmas and theorems.

Theorem 1: The solution space of FANROUT is the product of those of PTREE and CαTREE.

Proof: Analysis of the pseudo-code shows any P-Tree structure whose buffers directly drive at most one other buffer is considered by FANROUT. Also, any Cα-Tree whose buffers' output nets are implemented using PTREE is considered by FANROUT. ■

Lemma 6: The following statements are true for any routing structure ℜ that connects a source to a set of sinks:

I) By decreasing the load of any sink, the capacitance observed at the root of ℜ does not increase.

II) By increasing the required time of any sink, the required time at the root of ℜ does not decrease.

[4] PTREE sees the root of γ as a pseudo-sink whose required time and load are the ones of γ.

Proof: The above two statements are proven using simple circuit and graph theory rules. ■

Lemma 7: PTREE is monotone with respect to the load and the required time of the sinks.

Proof: This lemma is proven by induction and Lemma 6. ■

Lemma 8: The use of the prune operation by FANROUT does not result in the loss of any non-inferior solution.

Proof: Assume that σ_2 is inferior with respect to σ_1. By induction, if σ_2 is the whole net and its input is directly connected to the net driver, the required time does not decrease and the load does not increase by replacing σ_2 with σ_1. If σ_2 is a solution to a sub-problem, its input is driven by another internal node, called *g*. Due to the monotonic behavior of PTREE (c.f. Lemma 7), at *g* the required time and the input load of the implementation, including σ_2, is guaranteed to be no better than those of the implementation containing σ_1. A similar argument is then valid for *g* and the rest of the internal nodes down to the leaf nodes. ■

Theorem 2: FANROUT is an optimal algorithm.

Proof: An examination of the dynamic-programming structure of FANROUT shows that if no pruning is performed on the solution curves, all the possible solutions will be considered. Therefore, to prove the optimality of the algorithm it is enough to prove that for an optimal solution, replacing a non-inferior solution with an inferior solution cannot improve the whole implementation; this, however, was proven in Lemma 8. ■

Lemma 9: Depending on what metric is used for measuring the area, the number of solutions in a solution curve is bounded either polynomially or pseudo-polynomially.

Proof: The load of any solution is the input capacitance of the driving buffer. However, the number of distinct input capacitances of the buffers is bounded by the total number of available buffers in the library *m*.

In every solution the maximum number of inserted buffers is bounded by *O(n)*. Therefore, the number of distinct buffer areas is also bounded by *O(n)* since the area of every buffer is smaller than a constant number, and the smallest non-zero difference between the area of every two solutions is always greater than a constant number. Both these limits are determined by the library and do not depend on the size of the problem.

If the total buffer area is the metric used for measuring the area, the number of non-inferior solutions in a solution curve is bounded by *O(mn)*. The reason is that the prune operation keeps at most one solution per each distinct area and input load values.

The bound becomes pseudo-polynomial when the total capacitance is the metric used for measuring the area. In that case we assume that the number of distinct capacitive loads (called *q*) is polynomially bounded and is larger than the number of inserted buffers. As a result the number of non-inferior solutions is pseudo-polynomially bounded by *O(mq)*. ■

For the sake of simplicity, in the following theorem it is assumed that the total buffer area is used as the metric to measure the area cost of a solution.

Theorem 3: FANROUT has $O(kmn^3)$ memory complexity, where *k*, *m*, and *n* are numbers of candidate locations, buffers, and sinks, respectively.

Proof: There are *k* candidate locations, and for each combination of *L* and *R* (a total of *n(n+1)/2* combinations), there is a solution curve. Each solution curve stores *O(mn)* solutions, and as a result, the claim is proven. ■

Theorem 4: FANROUT has $O(mqk^2\alpha^5n^3)$ runtime complexity, where *k*, *m*, and *n* are the numbers of candidate locations, buffers, and sinks, respectively. Also, α is the maximum branching factor in Cα-Tree, and *q* is a polynomially bounded number of distinct capacitive loads.

Proof: In Fig. 6, the number of iterations performed in lines 4 through 7 is $O(\alpha^2n^2)$. Lines 8 and 9 introduce *O(k)* and *O(mn)* complexity, respectively. Calling PTREE in line 10 costs $O(k\alpha^3q)$, because the number of sinks provided to PTREE is always less than α ; see Corollory 1. The complexity of FANROUT is determined by considering all of the above factors. ■

V. Local Order-Perturbation

This section presents a new technique that can enhance any order-dependent dynamic-programming based algorithm - such as PTREE, CαTREE, and FANROUT - to generate optimal solutions with respect to a neighborhood of solutions.

Definition 3: An *order* Π on *n* sinks is a one-to-one function defined as $\Pi:\{1, 2, \ldots, n\}\rightarrow\{1, 2, \ldots, n\}$, and $j=\Pi(i)$ is called the *position of* s_i *in* Π. Also, Π^{-1} is the inverse function of Π, and $i=\Pi^{-1}(j)$ gives the sink's index of the *j*th element in Π.

Example 1: $\Pi=\{(1\rightarrow4), (2\rightarrow6), (3\rightarrow1), (4\rightarrow5), (5\rightarrow3), (6\rightarrow2), (7\rightarrow8), (8\rightarrow7), (9\rightarrow9)\}$, or equivalently, $(s_3,s_6,s_5,s_1,s_4,s_2,s_8,s_7,s_9)$ is an order on $\{s_1,s_2,\ldots,s_9\}$. Also, $\Pi(3)=1$ means s_3 is the first element in Π, and $\Pi^{-1}(2)=6$ means that s_6 is the second element in Π .

Although an algorithm that constructs an optimal structure for any given order is a useful tool, determining a "good" sink order remains as the main challenge. In the problem of buffered routing generation, required times, input loads, and physical locations of sink nodes should all be considered in generating a suitable order. Incorporating those independent and sometimes opposing parameters into the construction of an order is not an easy task. Due to the exponentially large number of possible orders, designers are forced to use heuristic approaches to combine the effects of those parameters in an ad-hoc manner. In general, the limitation imposed by working with one order at a time is very restrictive and undesirable.

The local order-perturbation method is a technique that works in a neighborhood of sink orders. No matter how one has determined an order, the semi-order-independent dynamic-programming formulation performs a systematic search in the neighborhood of that order. If the initial order is not a locally optimal order but close to it, this method chooses the optimal order automatically. The main advantage of such a technique is that it maintains an efficiency that is exponentially better than that of an exhaustive search method while preserving the optimality. Its superiority originates primarily from its

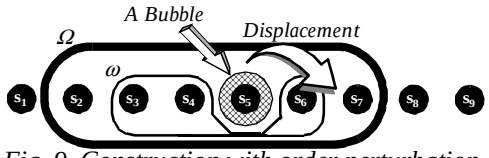

Fig. 9. Construction with order perturbation.

enhanced dynamic programming nature that enables the method to take advantage of all similar sub-problems among all the neighboring orders and thus avoid recomputing any sub-solution.

By allowing the bottom-up semi-order-independent algorithm to apply order perturbation operations, the sink order in the resulting solution can deviate from the initial order. A simple case is shown in Fig. 9 where the right-side border of a sub-group ω has been perturbed. Consequently, the order in the resulting group Ω is $(s_2,s_3,s_4,s_6,s_5,s_7)$ as opposed to the initial $(s_2,s_3,s_4,s_5,s_6,s_7)$ order; that is, in the new order s_5 has been swapped with s_6. In Fig. 9, s_5 has been left out from ω; it is called a *bubble*. When ω is used in a larger sub-group, it can be assumed that the bubble has been moved to the other side of the border of ω in the final structure. That operation causes the swapping of the position of two neighboring sinks.

Definition 4: For a set of sinks $\{s_1, s_2, \ldots, s_n\}$, the *neighborhood* of Π is defined as:

$$N(\Pi)=\{\Pi' \mid \forall s_i, |\Pi(i) - \Pi'(i)| \le 1\}.$$

In other words, the difference between the position of every s_i in Π and Π' is at most one.

Example 2: $\Pi'=(s_1,s_3,s_2,s_4,s_5,s_6,s_8,s_7,s_9)$ is in the neighborhood of $\Pi=(s_1,s_2,s_3,s_4,s_5,s_6,s_7,s_8,s_9)$, but $\Pi''=(s_3,s_2,s_1,s_4,s_5,s_6,s_7,s_8,s_9)$ is not in the neighborhood of Π.

Definition 5: For $n>1$, *displacing* the element i $(1\le i\le n-1)$ of Π (also referred to as the *displacement operation*) is defined as swapping the location of $s_{\Pi^{-1}(i)}$ with the location of $s_{\Pi^{-1}(i+1)}$ in Π. The displacement operation on element i always involves two neighboring elements in Π, i.e., $s_{\Pi^{-1}(i)}$ and $s_{\Pi^{-1}(i+1)}$. The two elements are called the *elements* of the displacement operation.

Definition 6: Two displacement operations are *non-overlapping* if and only if they have no element in common. A set of displacement operations are non-overlapping if and only if every two operations in the set are non-overlapping.

Example 3: Displacing the *4th* element of $\Pi'=(s_1,s_3,s_2,s_4,s_5,s_6,s_8,s_7,s_9)$ results in $\Pi''=(s_1,s_3,s_2,s_5,s_4,s_6,s_8,s_7,s_9)$

Lemma 10: Every $\Pi' \in N(\Pi)$ can be built from Π using a series of non-overlapping displacement operations.

Proof: This is a proof by induction. Let us represent a sub-string of Π that consists of the i left-most elements of Π by $sub\_string(\Pi, i)$. For $i=0$, it is trivial that $sub\_string(\Pi', 0)=sub\_string(\Pi, 0)$. Suppose for $i=\kappa-1$, $sub\_string(\Pi', \kappa-1)$ can be obtained from $sub\_string(\Pi, \kappa-1)$, using a series of non-overlapping displacement operations. Let $j=\Pi^{-1}(\kappa)$. Since $\Pi' \in N(\Pi)$, there are three neighborhood cases according to Definition 4.

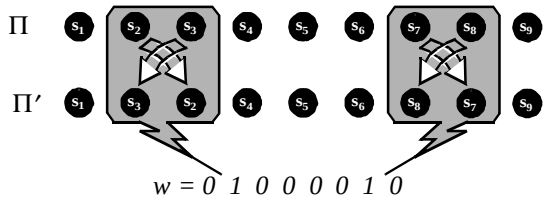

Fig. 10. Equivalence of W and N(Π).

- $\Pi(j)=\Pi'(j)$: This means that the κth elements in Π and Π' are the same. Therefore, the statement given in the above lemma holds for $i=\kappa$ as well.
- $\Pi(j)=\Pi'(j)-1$: This means that the $\kappa+1$th element in Π' is the same as the κth element in Π. Let $j'=\Pi'^{-1}(\kappa)$. In this case, we will have $\Pi(j')=\Pi'(j')+1$, otherwise $\Pi(j')-\Pi'(j')>1$ (in violation of Definition 4) because the $\Pi'(j')-1$ and $\Pi'(j')$ slots have already been taken by sinks other than $s_{j'}$. Therefore, we have a displacement at the κth element of Π, and the statement given in the above lemma holds for $i=\kappa+1$ as well.
- $\Pi(j)=\Pi'(j)+1$: This case cannot happen because it implies s_j is the $\kappa-1$th element of Π' which is in conflict with the above assumption. ■

Definition 7: Any arbitrary (n-1)-bit binary number w is called a *non-overlapping displacement code*, if and only if, it contains no two adjacent bits of 1. Also, W is defined as the set of all non-overlapping displacement codes.

Definition 8: The position of a bit b in a binary number w is defined as the number of bits on the left-side of b plus one.

Lemma 11: There exists a one-to-one relationship between the members of W and $N(\Pi)$.

Proof: First, we prove that for $\forall w \in W$ there exists a corresponding $\Pi' \in N(\Pi)$. For every 1 bit in w, set i to the position of that bit in w and displace the ith element of Π; call the resulting order Π'. Before the first displacement operation the inequality given in Definition 4 holds. Also, if the inequality between the initial order and the resulting order after j displacement operations is valid, it still holds after the $j+1$th displacement operation. That is because every displacement operation changes the location of the two swapped elements by ± 1 and keeps the location of the other elements unchanged. In addition, since w is non-overlapping code, no element is displaced more than once. Consequently, using induction we conclude that $\Pi' \in N(\Pi)$.

Now, we prove that for $\forall \Pi' \in N(\Pi)$ there exists a corresponding $w \in W$. According to Lemma 10, Π' can be generated from Π using a unique set of non-overlapping displacements. Those displacement operations can be coded in a non-overlapping displacement code which belongs to W. ■

Example 4: Fig. 10 illustrates the equivalence of W and $N(\Pi)$ for a simple example.

Theorem 5: For $n>1$, the number of distinct orders in the neighborhood of a given order Π is equal to:

$$\frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{n+2} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+2}\right)$$

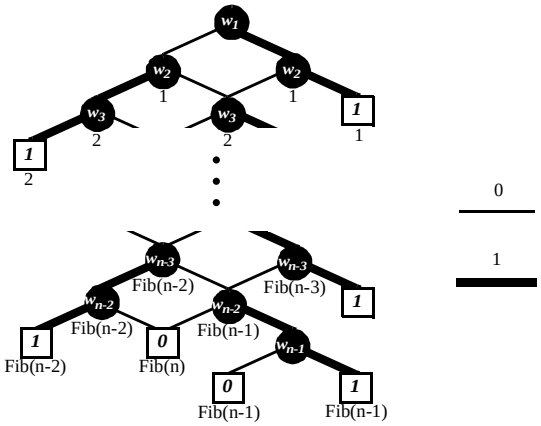

Fig. 11. BDD of f.

Proof: According to Lemma 11 there is a one-to-one relationship between $N(\Pi)$ and W. Therefore, these two sets have equal cardinality, and we can equivalently prove the above equation for W. So, we have to find out how many binary numbers in the form of $w=w_1w_2\ldots w_{n-1}$ exist that have no two adjacent 1s. The population of such numbers is equal to the off-set size of the following Boolean equation:

$$f=w_1w_2+w_2w_3+\ldots+w_{n-3}w_{n-2}+w_{n-2}w_{n-1}$$

Fig. 11 is the binary decision diagram (BDD) representation of the above equation. By induction, it can be verified that this structure is valid for $n>0$. In this figure, the number under each BDD node gives the number of distinct paths that exist from that node to the root.

Due to the symmetric structure of the equation and the corresponding BDD, the number of paths from a node to the root follows the Fibonacci number series. In the Fibonacci number series, the $k+1$th number is the sum of the k-1th and kth numbers in the series. As shown in Fig. 11, the number of distinct paths from the leaf node zero to the root is $2\times Fib(n)+Fib(n-1)$. Note that the factor of 2 appearing in the equation represents the fact that during the decomposition a zero sub-space has been reached while the decomposition is not yet over with respect to the last variable. By a few simple manipulations we get the final result:

$$2\times Fib(n)+Fib(n-1)=(Fib(n)+Fib(n-1))+Fib(n)$$
$$=Fib(n+1)+Fib(n)=Fib(n+2)$$

There is a direct closed-form for calculating the $n+2$th number in a Fibonacci series. According to the formula given in [MCS], we derive the equation given in the theorem. ■

Note the formula that returns the nth Fibonacci number involves square root of 5 (an irrational number), yet it always returns an integer for all (integer) values of n [MCS].

Theorem 5 proves that the size of $N(\Pi)$ is an exponential function of the number of sinks. Consequently, finding the best order in that sub-space of orders is a task of exponential complexity if a simple enumeration-based technique is used. However, all the common sub-solutions of different orders can be shared in a dynamic-programming based algorithm that utilizes the aforementioned idea of local order-perturbation. This in turn allows us to investigate the whole neighborhood in polynomial time.

Fig. 12. Grouping structures.

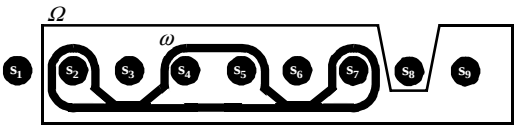

Fig. 13. Construction with perturbation.

Fig. 12 presents a set of *abstract grouping structures* $\{\chi_0, \chi_1, \chi_2, \chi_3\}$ by which one can cover a whole neighborhood of orders. χ_0 has no bubble on its sides, and χ_1, χ_2, and χ_3 have bubbles on the right-side, left-side, and both sides, respectively. For instance, the grouping ω of Fig. 9 is a χ_1-type structure. A full neighborhood is covered, if at each level of dynamic programming and for each sub-group of sinks, all the grouping structures are generated from all the grouping structures of their internal sub-groups; Fig. 13 shows an example.

Example 5: The example in Fig. 13 illustrates the use of χ_3 structure to generate a χ_1-type solution for Ω In this case, the resulting order is $(s_3,s_2,s_4,s_5,s_7,s_6,s_9)$. This new sub-solution will be used to generate larger sub-solutions that contain it.

The local order-perturbation technique can be extended to structures with more than one bubble on each side. Those structures in turn result in covering larger neighborhoods. However in that case, the number of grouping structures grows exponentially and, consequently, results in a significant slow-down of the corresponding construction algorithm.

VI. Semi Order-Independent Hierarchical Buffered Routing Tree Construction

In this section, the local order-perturbation theory is applied to FANROUT and B_PTREE (sub-sections A. and B.), and the resulting algorithm, MERLIN, is presented in sub-section C.

A. BUBBLE_CONSTRUCT

The technique presented in the following generates hierarchical buffered routing trees in a neighborhood of orders. The resulting hierarchies are consistent with the Cα-Tree structure, and in addition, the routing inside each layer of the Cα-Tree hierarchy is a P-Tree. Some parts of the BUBBLE_CONSTRUCT code (Fig. 14) are similar to the code of FANROUT and are not discussed again. It is assumed that the reader is familiar with the algorithm presented in section IV.

BUBBLE_CONSTRUCT operates on three dimensional solution curves Γ, each associated with a distinct set of values for l, r, p, and e. The first three variables are the same as the ones defined for FANROUT. The variable e encodes the grouping structure used to generate the solution curve.

1) Initialization: In this section, a set of solution curves are initialized. Here, sub-groups of length 1 are considered, and the corresponding solution curves for every candidate buffer location, sink, and grouping structure are initialized. Note that

algorithm *BUBBLE_CONSTRUCT(s, P, B,* $\Pi=(s_1, s_2, \ldots, s_n)$ *)*

INITIALIZATION

1. **for** $e = 0$ **to** 3
2. // similar to the lines 1 through 3 in Fig. 6 except that
   // $\Gamma(1, r, p)$ should be replaced with $\Gamma(1, r, p, e)$

CONSTRUCTION

3. **for** $L = 1$ **to** n
4. **for** $E = 0$ **to** 3
5. **set** $L' = L + STRETCH(L, E)$ // see Fig. 15
6. **for** $R = n$ **downto** L'
7. **set** $G = SINK\_SUBSET(\Pi, R, L, E)$ // see Fig. 16
8. **for** $l = \max(1, L\text{-}\alpha+1)$ **to** $L\text{-}1$
9. **for** $e = 0$ **to** 3
10. **set** $l' = l + STRETCH(l, e)$ // see Fig. 15
11. **for** $r = R$ **downto** $R\text{-}l'+1$
12. **set** $g = SINK\_SUBSET(\Pi, r, l, e)$ // see Fig. 16
13. **if** $g\text{-}G \neq \phi$ **continue**
14. **foreach** $p \in P$
15. **foreach** $\gamma \in \Gamma(l, r, p, e)$
16. **set** $G' = REORDER(G, g, \gamma)$
17. **set** Δ = *PTREE(P, B, G')
18. // similar to the lines 11 through 17 in Fig. 6 except that
    // $\Gamma(L, R, p')$ should be replaced with $\Gamma(L, R, p', E)$

EXTRACTION

19. // similar to the lines 18 through 20 in Fig. 6 except that
    // $\Gamma(n, n, s)$ should be replaced with $\Gamma(n, n, s, 0)$

Fig. 14. The pseudo-code for BUBBLE_CONSTRUCT.

algorithm *STRETCH(L, E)*

1. **set** $g=0$
2. **if** $L=2$ **and** $E>0$ **set** $g=1$
3. **else if** $L>2$ **and** $E=1$ **set** $g=1$
4. **else if** $L>2$ **and** $E=2$ **set** $g=1$
5. **else if** $L>2$ **and** $E=3$ **set** $g=2$
6. **return** g

Fig. 15. The pseudo-code for STRETCH.

algorithm *SINK_SUBSET(* $\Pi=(s_1, s_2, \ldots, s_n)$*, R, L, E)*

1. **if** $L=1$ **set** $G = \{ s_R \}$
2. **else if** $L=2$
3. **switch** E
4. **case** 0 : **set** $G = \{ s_{R-1}, s_R \}$
5. **case** $1, 2, 3$: **set** $G = \{ s_{R-2}, s_R \}$
6. **else**
7. $L'=STRETCH(L, E)$
8. **switch** E
9. **case** 0 : **set** $G = \{ s_{R-L'+1}, s_{R-L'+2}, s_{R-L'+3}, \ldots, s_{R-2}, s_{R-1}, s_R \}$
10. **case** 1 : **set** $G = \{ s_{R-L'+1}, s_{R-L'+2}, s_{R-L'+3}, \ldots, s_{R-2}, s_R \}$
11. **case** 2 : **set** $G = \{ s_{R-L'+1}, s_{R-L'+3}, \ldots, s_{R-2}, s_{R-1}, s_R \}$
12. **case** 3 : **set** $G = \{ s_{R-L'+1}, s_{R-L'+3}, \ldots, s_{R-2}, s_R \}$
13. **return** G

Fig. 16. The pseudo-code for SINK_SUBSET.

for sub-groups with length 1, all four grouping structures (χ_0 , χ_1 , χ_2 , and χ_3) are the same; however, for the sake of simplicity in the rest of the pseudo-code, separate (although similar) solution curves are generated for each case. A similar situation occurs for χ_1 and χ_2 where $L=2$.

2) Construction: The main difference between this algorithm and FANROUT is in the grouping phase, i.e., lines 3 through 13 in Fig. 14. BUBBLE_CONSTRUCT starts from $L=1$ and goes up to $L=n$. For each new sub-group of sinks, all possible grouping structures (coded by numbers *0* to *3*) are enumerated in line 4. For the case of χ_0 ($E=0$), the length of the sub-group is equal to L, but for the other cases the actual length of the sub-group is larger by one or two units in order to capture the effect of inserting one or two bubbles on the sides. This new length is

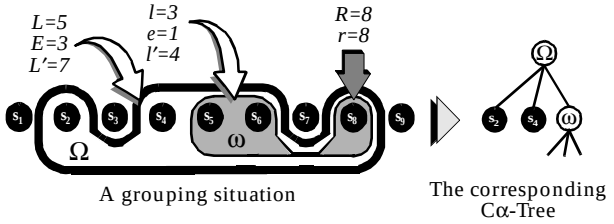

Fig. 17. A legal grouping scenario for BUBBLE_CONSTRUCT.

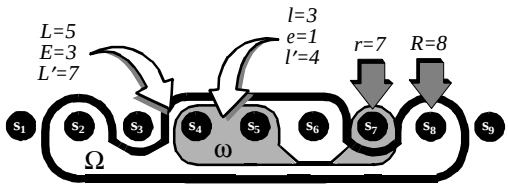

Fig. 18. An illegal grouping case for

calculated and stored in L' (refer to line 5 and Fig. 15). In line 6, all the possible sub-strings of length L' are considered from the right to the left of Π. In fact, the variable R points to the right-most element of the sub-strings of L' elements.

Lines 8 through 11, similar to lines 3 through 6, investigate all possible sub-group lengths with different grouping structures and positions which fit inside the sub-group being constructed. Fig. 17 illustrates an example where a sub-group of 5 sinks, Ω, is being generated using a combination of an already generated sub-group of 3 sinks, ω, and two other sinks, i.e., s_2 and s_4.

It can be seen that in some cases Ω and ω are not compatible. As an example, consider the situation shown in Fig. 18 where the difference between the values of r and R causes the grouping structure of ω to not fit in the grouping structure of Ω. Those cases are detected and skipped in line 13 of the pseudo-code. Note that sets G and g - calculated in lines 7 and 12 - represent the sets of sinks included in Ω and ω, respectively (also refer to Fig. 16).

In line 16, REORDER updates G by replacing all the sinks that belong to g with a pseudo-sink that represents the root of γ (a solution to ω). The resulting order is called G′.

In line 17, an enhanced version of B_PTREE (called *PTREE) is called to generate a new set of solutions for all candidate locations. Every solution created by *PTREE shows the combination of ω with the rest of sink nodes of Ω. The details of *PTREE are presented in the following sub-section.

3) Extraction: In this section, a solution from $\Gamma(n, n, s, 0)$ that best satisfies the input constraints is selected and reconstructed by tracing back the stored pointers.

The quality and complexity of BUBBLE_CONSTRUCT is further discussed in sub-section D.

*B. *PTREE*

*PTREE is a solution to the problem of non-hierarchical buffered routing tree construction. As mentioned earlier, *PTREE is called by BUBBLE_CONSTRUCT within each level of the Cα-Tree hierarchy. Consequently, *PTREE is responsible for conducting the order-perturbation task within each level of the hierarchy, otherwise BUBBLE_CONSTRUCT would not be optimal with respect to

```
algorithm *PTREE(P, B, Π=(s1 , s2, ... , sn))
    INITIALIZATION
1.  // the same as the one in BUBBLE_CONSTRUCT, see Fig. 14
    CONSTRUCTION
2.  for L = 2 to n
3.    for E = 0 to 3
4.      set L′ = L + STRETCH(L, E); // see Fig. 15
5.      for R = n downto L′
6.        set G = SINK_SUBSET( Π, R, L, E ); // see Fig. 16
7.        for l1 = 1 to L-1
8.          for e1 = 0 to 3
9.            set l1′ = l1 + STRETCH(l1, e1); // see Fig. 15
10.           set r1 = R
11.           set g1 = SINK_SUBSET( Π, r1, l1, e1 ); // see Fig. 16
12.           if g1-G ≠ ϕ continue;
13.           set l2 = L - l1
14.           switch e1
15.             case 0, 1 : set r2 = r1-l1′
16.               switch E
17.                 case 0, 1 : set e2 = 0
18.                 case 2, 3 : set e2 = 2
19.             case 2, 3 : set r2 = r1-l1′+2
20.               switch E
21.                 case 0, 1 : set e2 = 1
22.                 case 2, 3 : set e2 = 3
23.           set g2 = SINK_SUBSET( Π, r2, l2, e2 ); // see Fig. 16
24.           if g2-G ≠ ϕ or g2-g2 ≠ ϕ continue;
25.           // combine Γ( l1, r1, p, e1 ) and Γ( l2, r2, p, e2 ) in
              // the same way PTREE combines the solution curves
```

*Fig. 19. The pseudo-code for *PTREE.*

the complete neighborhood of orders.

The algorithm is an enhanced version of B_PTREE [LCL96] with two main differences: I) it uses the local order-perturbation technique, and as a result, it is optimal with respect to a neighborhood of orders and II) as an input it takes a set of candidate buffer locations, and therefore, the locations of buffers and Steiner points are not restricted to Hanan points.

At every step of dynamic programming in *PTREE, a sub-group of sinks Ω is solved using the existing solutions to two smaller sub-groups (ω_1 and ω_2) which partition Ω into two segments. Fig. 20 illustrates a legal grouping situation in which ω_1 and ω_2 properly partition Ω

For the design of *PTREE, three issues have been considered:

- ω_1 and ω_2 do not share any sinks,
- ω_1 and ω_2 cover all (and only) the sinks of Ω,
- All the possible combinations of grouping structures and sizes must be considered for ω_1 and ω_2.

The pseudo-code of *PTREE is given in Fig. 19. In lines 2 through 5, all combinations of size *L*, grouping structure *E*, and position *R* are constructed for Ω . Then, ω_1 is constructed in lines 7 through 10. As shown in the code, the rightmost element of ω_1 is always the rightmost element of Ω . Some combinations of Ω and ω_1 may not be legal, which means Ω does not contain at least one element of ω_1 . Although those cases could be avoided during the construction of ω_1, for the sake of presentation, they are explicitly pruned by the condition in line 12. In that line, any ω_1 incompatible with Ω is detected and disregarded.

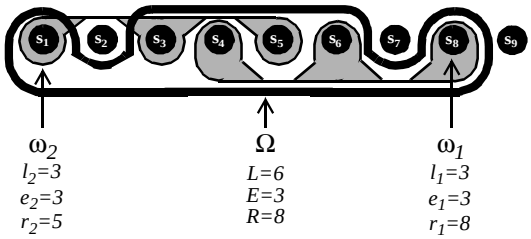

*Fig. 20. A legal grouping scenario for *PTREE.*

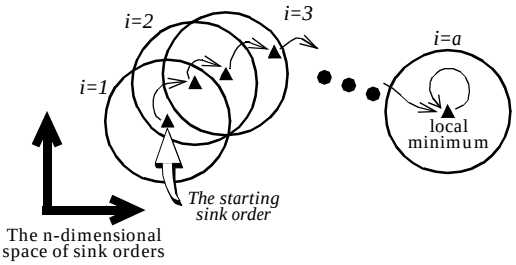

Fig. 21. Local neighborhood search in MERLIN.

.In lines 13 through 22, ω_2 is generated by considering Ω and ω_1. The size, grouping structure, and position of ω_2 are determined so that ω_2 contains all the sink nodes of Ω that are not included in ω_1 . If ω_1 has a bubble on its left side, i.e., grouping structures χ_2, χ_3, that bubble is the rightmost element of ω_2; see line 19. Similarly, if Ω has a bubble on its left-side, ω_2 should have a grouping structure with a bubble on its left, and so on. For more details refer to the pseudo-code in Fig. 19. Again, some illegal cases may occur but are pruned in line 24.

After line 24, ω_1 and ω_2 are known and their solution curves are combined the same way that B_PTREE combines the solution curves. Interested readers may refer to [LCL96] for the details of B_PTREE.

C. MERLIN

This sub-section presents MERLIN, a local neighborhood search algorithm, which employs BUBBLE_CONSTRUCT to find a local optimum sink order and the optimum buffered routing tree corresponding to that order.

Generally, an optimization problem has a set of solutions and a cost function that assigns a value to every solution. The goal is to find an optimal solution, i.e., one that has the minimum (or maximum) cost. The local neighborhood search, as a member of iterative solution methods, is a widely-used, general approach for solving optimization problems.

To obtain a local search *(LS)* algorithm for solving an optimization problem, one superimposes a neighborhood structure on the solutions, i.e., for each solution a set of neighboring solutions is specified. This LS algorithm starts from some initial solution, which may be constructed by some other algorithm or generated randomly, and from then on keeps moving to a better neighboring solution, until finally it terminates at a locally optimal solution. This method has been applied both in the context of continuous and discrete optimizations [Ya92]. In general, *simulated annealing* is a special case of local neighborhood search that allows uphill moves. Fig. 21 illustrates the behavior of a local neighborhood

```
algorithm MERLIN( s , P, B, Π=(s_1 , s_2 , … , s_n) )
1.  set Π′ = Π
2.  do {
3.    set Π = Π′
4.    set ℜ = BUBBLE_CONSTRUCT( s , P , B , Π )
5.    set Π′ = SINK_ORDER( ℜ )
6.  } while ( Π != Π′ )
7.  return ℜ
```

Fig. 22. The pseudo-code for MERLIN.

search.

Definition 9: A function $N:F \rightarrow 2^F$, which associates a subset $N(x)$ with each $x \in F$, is a *neighborhood function* over F iff $\forall x \in F$, $x \in N(x)$ and $\forall x \in F$, $x \in N(y) \Rightarrow y \in N(x)$.

BUBBLE_CONSTRUCT induces a well-defined neighborhood function in which it finds the best solution. The same definition is also used by MERLIN.

Lemma 12: The properties required by Definition 9 are consistent with those of neighborhood introduced in Definition 4.

Proof: In Theorem 5, we proved that the size of the neighborhood, $N(\Pi)$, is always greater than *1*, independent of the choice of Π. Also, for every $\Pi' \in N(\Pi)$ there is a unique non-overlapping displacement code, *w*, that transforms Π to Π'. To prove that $\Pi \in N(\Pi')$also, we need to prove that there is a non-overlapping displacement code, *w′*, that transforms Π' to Π. It can be shown that $w'=w$ is, in fact, the solution. ■

There exist at least two sink orders , i.e., Π and Π', in common between the neighborhood of two consecutive iterations of MERLIN's local search (see Fig. 22). In fact, this overlap, $OVERLAP(N(\Pi),N(\Pi'))$, is often relatively large. Intuitively, when the corresponding non-overlapping displacement code has more 1s, $OVERLAP(N(\Pi),N(\Pi'))$ is smaller. Obviously, it is a waste to consider the overlapping sub-space twice. This can be prevented by keeping solution curves of the very last iteration. For similar sub-problems simply copy the corresponding solution curve between the two iterations. However, this speed-up is achieved at the cost of doubling memory usage.

D. Quality and Complexity Analysis

Theorem 6 : *PTREE executes in $O(k\alpha^3 q)$ where *k* is the total number of buffer candidate points, α is the number of sinks, and *q* is a polynomially bounded number of distinct capacitive loads.

Proof: In Fig. 19, lines 2, 5, and 7 each introduce $O(\alpha)$ complexity. Note that in the pseudo-code, *n* is the number of sinks that is referred to as α in this theorem. The merge operation (line 25), which is the same as in B_PTREE, has a $O(kq)$ complexity [LCL96]. ■

Lemma 13: Orders generated by BUBBLE_CONSTRUCT are in the neighborhood of the initial order.

Proof: This is a proof by induction. The pseudo-code directly forces the grouping structures to cover each other like nested shells. Starting from the innermost shell, we analyze the effect of grouping structures. Case *i=1*: after the bubble-out step (see Fig. 9), for the innermost grouping structure, the order of all the sinks remains unchanged except for the two which are on the border of the bubbled sub-group. Consequently, the

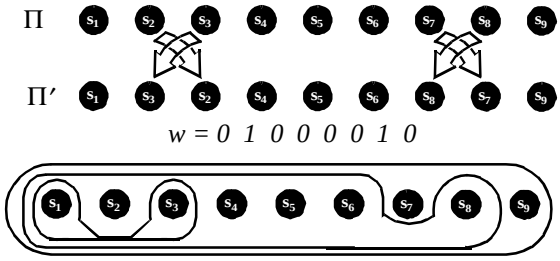

Fig. 23. An illustration for the proof of Lemma 14.

inequality relation in Definition 4 remains valid, and the resulting order is within the neighborhood of the initial order. Case *i=n:* suppose that after the bubble-out step for the *n-1* innermost grouping structures, the inequality of Definition 4 still holds. The order for the sinks on the border of the *n*th grouping structure must still be unchanged because no overlap is allowed between the borders of two grouping structures. Therefore, even after the bubble-out step for the *n*th grouping structure, the resulting order is within the neighborhood of the initial order. ■

Lemma 14: Any $\Pi' \in N(\Pi)$, is considered by BUBBLE_CONSTRUCT.

Proof: BUBBLE_CONSTRUCT implicitly tries all the possible valid combinations of grouping structures on Π. Therefore, it is enough to prove that $\forall \Pi' \in N(\Pi)$ there exists a combination of grouping structures that result in that order. Suppose that *w* is the non-overlapping displacement code of $\Pi' \in N(\Pi)$, as given in Definition 7. Starting from the left-most 1-bit in *w* (*j* is the position of that bit in *w*) extend a χ_1-type sub-group from the left-most sink to the *j+1*th sink. After the bubble-out step the resulting order is similar to Π' for the *j* left-most sinks and similar to Π for the rest of the sinks. Repeat this operation for the next bit 1 in *w* in order from left to right. There are no two neighboring 1s in *w*; therefore at each step the left portion of the resulting order resembles Π' and the other portion is like Π . At the last step when there is no 1 left in *w*, we cover the initial order from left to right with a χ_0-type sub-group. The resulting order, after the bubble-out step for all the sub-groups, is Π', and since it has a valid grouping structure it is considered by the pseudo-code of Fig. 14. ■

The example in Fig. 23 illustrates the proof of Lemma 14.

Lemma 15: Any identical sub-problem among the members of $N(\Pi)$ is shared and processed only once.

Proof: Any sub-problem is uniquely identified by *l*, *e*, and *r* values. $\forall p \in P$, $\Gamma(l,e,r,p)$ is generated only once, no matter in which compatible and larger grouping structure it will be used later. Note that according to Lemma 13 and Lemma 14, BUBBLE_CONSTRUCT covers the whole space of $N(\Pi)$. ■

Theorem 7: The solution space of BUBBLE_CONSTRUCT is the product of the spaces of *P-Tree and Cα-Tree for the neighborhood of the initial given order.

Proof: $\forall \Pi' \in N(\Pi)$, all the corresponding Cα-Trees with the boundary of their sub-groups on the displaced sinks' locations are visited and for every one of them all the *P-Tree structures are considered by *PTREE. However, there are some Cα-Trees that correspond to Π' whose displacements are not at

the boundary of the sub-groups. *PTREE considers all the necessary displacements inside one layer of those Cα-Tree. ■

Lemma 16: BUBBLE_CONSTRUCT is monotone with respect to required time, load, and area.

Proof: By considering that *PTREE is monotone with respect to the required time, load, and area, we can conclude that in a Cα-Tree, decreasing the load of either an internal or a sink node results in the decrease of load in its immediate parent. A similar argument is valid for required time and total area. ■

Lemma 17: In BUBBLE_CONSTRUCT, the pruning operation does not eliminate any non-inferior solution.

Proof: The proof follows Lemma 16 and Definition 2. ■

Theorem 8: Subject to restrictions imposed by the *P-Tree and Cα-Tree structures, BUBBLE_CONSTRUCT finds all the non-inferior solutions with respect to required time and total area in the neighborhood of a given order.

Proof: If no pruning is performed all the space is explicitly constructed (see Theorem 7). Lemma 17 states that the prune operation drops the sub-solutions that are only used in inferior solutions. Therefore, all the non-inferior solutions remain in the final curves of BUBBLE_CONSTRUCT. ■

For the sake of simplicity, in the following it is assumed the total buffer area is the metric used to measure solution area.

Theorem 9: BUBBLE_CONSTRUCT has $O(kmn^3)$ memory complexity where k, m, and n are numbers of candidate locations, buffers, and sinks, respectively.

Proof: The proof is the same as for Theorem 3. The only difference is that the number of solution curves is four times higher in BUBBLE_CONSTRUCT than in FANROUT, since for every grouping structure a solution curve is stored. ■

Theorem 10: BUBBLE_CONSTRUCT has $O(mqk^2\alpha^5n^3)$ runtime complexity where k, m, and n are the number of candidate locations, buffers, and sinks, respectively. Also, α is the maximum branching factor in Cα-Trees, and q is polynomially bounded number of distinct capacitive loads.

Proof: The proof is similar to that of Theorem 4. ■

Corollary 2: Assuming that m, q, and α are parameters independent from the size of the problem n and are determined by the library and technology, the effectual worst-case complexity of BUBBLE_CONSTRUCT is $O(k^2n^3)$.

Theorem 11: The cost associated with orders produced by iterations of MERLIN (but the last one) is strictly decreasing.

Proof: BUBBLE-CONSTRUCT always returns the best order in the neighborhood; thus if a different order is returned, it must correspond to a lower cost. In the last iteration, the cost of the given order is the best in the neighborhood, and that is how the iteration is terminated. ■

VII. Experimental Results

In this section, three experimental setups have been tested and compared on a set of benchmark circuits.

- *Setup-I:* For every net, fanout optimization using LTTREE is followed by a routing tree construction phase using PTREE. In LTTREE, the net sinks are sorted with respect to their required times. However, in PTREE the net sinks are sorted by a solution to the TSP (Traveling Salesman Problem) using the method suggested in [LCLH96].
- *Setup-II:* Routing tree generation using PTREE is followed by buffer insertion using the van Ginneken's method [Gi90]. The sink order for PTREE is again the TSP order.
- *Setup-III:* Finally, hierarchical buffered routing generation is performed using MERLIN and an initial TSP order.

All the experiments have been implemented and executed in SIS [SSLM92] and on a dual-processor Ultra-2 Sun Sparc workstation with 256MB memory. In these experiments, an industrial standard cell library (0.35μm CMOS process) consisting of 34 buffers has been used. Gate and wire delays are calculated using a 4-parameter delay equation and the Elmore delay model [El48], respectively.

A. Comparison on Individual Nets

Table 1 reports the results of running the above three experimental setups on 18 individual nets randomly selected from a set of benchmark circuits. For every extracted net, the sink locations are determined randomly in a bounding box. The size of the box has been determined such that the delay of a wire segment whose length is half the perimeter of the box is approximately equal to the delay of an average gate driving that wire. In addition, the load and required time sink data have been selected randomly from a nominal range.

In Table 1, the reported area and delay values are the total buffer area and the maximum delay at the root of the net in the resulting buffered routing structures. Also, the runtimes have been reported in seconds for every net and setup. Note, the data of Setup-I has been reported in absolute values; however, for the other two setups the results have been scaled with respect to their corresponding data in Setup-I.

For each net, the last column in Table 1 reports the number of iterations performed during the execution of MERLIN. For about 28% of the cases reported in the table, MERLIN converges in one iteration. That indicates that the initial sink order is a local minimum in its neighborhood. This effect can be used as a metric to measure the effectiveness of the heuristics used to generate the initial order. The experiments indicate that the TSP heuristic tends to perform better with respect to this metric compared to a few other heuristics. Hence, the TSP order has been used in all experimental setups.

B. Comparison on Circuits

Table 2 reports the post-layout total area and delay values for a set of benchmark circuits. In these experiments the above three setups have been plugged into a full design flow that extends from the logic synthesis all the way down to the detailed routing. The resulting design flows have been named *Flow-I*, *Flow-II*, and *Flow-III*, respectively. Again, the data for Flow-I is the absolute to which the rest are scaled.

VIII. Conclusions

In this paper, the problem of distributing a signal among a set of sinks with different placement, load, and required time values has been addressed. The proposed technique generates a

set of non-inferior buffered routing structures that provide different trade-offs between the required-time at the root and the total buffer area. The introduced solution consists of an iterative optimization block that uses a local neighborhood search strategy and an optimization engine based on dynamic programming that generates all the non-inferior structures in the neighborhood of a given sink order. This optimization engine generates and propagates 3-dimensional solution curves and employs a novel local order-perturbation method to cover an exponentially sized solution space in polynomial time. The experimental results show significant delay improvement with little area penalty compared to the conventional buffer and routing tree generation techniques.

ACKNOWLEDGEMENTS

The authors wish to thank Dr. John Lillis of the University of Illinois at Chicago for helpful discussions and comments.

REFERENCES

[Be57] R. Bellman, *Dynamic Programming*. Princeton University Press, 1957.

[BCD89] C. L. Berman, J. L. Carter, and K. F. Day, "The fanout problem: From theory to practice," In *Proceedings of Advanced Research on VLSI*, pp. 69-99, 1989.

[CHKM96] J. Cong, L. He, C. Koh, and P. Madden, "Performance optimization of VLSI interconnect layout," In *Integration, the VLSI Journal 21*, pp. 1-94, 1996.

[CLZ93] J. Cong, K. Leung, and D. Zhou, "Performance-driven interconnect design based on distributed RC delay model," In *Proceedings of Design Automation Conference*, pp. 606-611, 1993.

[El48] W. C. Elmore, "The transient response of damped linear network with particular regard to wideband amplifiers," In *Journal of Applied Physics* 19, pp. 55-63, 1948.

[Go76] M. C. Golumbic, "Combinatorial merging," *IEEE Transactions on Computers*, vol. 25, pp. 1164-1167, Nov. 1976.

[Gr92] L. K. Grover, "Local search and the local structure of NP-complete problems," In *Operations Research Letters 12*, pp. 235-243, Oct. 1992.

[Gi90] L.P.P.P. van Ginneken, "Buffer placement in distributed RC-tree networks for minimal Elmore delay," In *Proceedings of International Symposium on Circuits and Systems*, pp. 865-868, 1990.

[GJ79] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco: W. H. Freeman, 1979.

[Ha66] M. Hanan, "On Steiner's problem with rectilinear distance," *SIAM Journal of Applied Mathematics*, No. 14, pp. 255-265, 1966.

[LCL96] J. Lillis, C. K. Cheng, and T. Y. Lin, "Simultaneous routing and buffer insertion for high performance interconnect," In *Proceedings of the Sixth IEEE Great Lakes Symposium on VLSI*, pp. 148-153, 1996 and *Proceedings of ACM/SIGDA Physical Design Workshop*, pp. 7-12, 1996.

[LCLH96] J. Lillis, C. K. Cheng, T. Y. Lin, and C. Ho, "New performance driven routing techniques with explicit area/delay tradeoff and simultaneous wire sizing," In *Proceedings of the 33rd Design Automation Conference*, pp. 395-400, 1996.

[LSP97] J. Lou, A. H. Salek, and M. Pedram, "An exact solution to simultaneous technology mapping and linear placement problem," In *Proceedings of International Conference on Computer-Aided Design*, pp. 671-675, 1997.

[MCS] http://www.mcs.surrey.ac.uk/Personal/R.Knott/Fibonacci/fibFormula.html .

[OC96a] T. Okamoto and J. Cong, "Buffered Steiner tree construction with wire sizing for interconnect layout optimization," In *Proceedings of International Conference on Computer-Aided Design*, pp. 44-49, 1996.

[OC96b] T. Okamoto and J. Cong, "Interconnect layout optimization by simultaneous Steiner tree construction and buffer insertion," In *Proceedings of ACM/SIGDA Physical Design Workshop*, pp. 1-6, 1996.

[Pe98] M. Pedram, "Logical-physical co-design for deep submicron circuits: challenges and solutions," In *Proceedings of Asia and South Pacific Design Automation Conference*, pp. 137-142, Feb. 1998.

[SLP98] A. H. Salek, J. Lou, and M. Pedram, "A simultaneous routing tree construction and fanout optimization algorithm," In *Proceedings of International Conference on Computer-Aided Design*, 1998.

[SLP99] A. H. Salek, J. Lou, and M. Pedram, "MERLIN: Semi-order-independent hierarchical buffered routing tree generation using local neighborhood search," In *Proceedings of Design Automation Conference*, pp. 472-478, 1999.

[SS90] K. J. Singh and A. Sangiovanni-Vincentelli, "A heuristic algorithm for the fanout problem," In *Proceedings of Design Automation Conference*, pp. 357-360, 1990.

[SSLM92] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli, "SIS: A system for sequential circuit synthesis," *Memorandum No. UCB/ERL M92/41*, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, May 1992.

[To90] H. Touati, "Performance-oriented technology mapping," Ph.D. thesis, *University of California, Berkeley, Technical Report UCB/ERL M90/109*, Nov. 1990.

[VP93] H. Vaishnav and M. Pedram, "Routability-driven fanout optimization," In *Proceedings of Design Automations Conference*, pp. 230-235, 1993.

[WM89] W.S. Wong and R.J.T. Morris, "A new approach to choosing initial points in local search," In *Information Processing Letters 30*, pp. 67-72, Jan. 1989.

[Ya92] M. Yannakakis, "The analysis of local search problems and their heuristics," In *Proceedings of the 7'th Annual Symposium on Theoretical Aspects of Computer Science*, pp. 298-311, 1990.

						Ratios normalized w.r.t. Setup I						
Taken from circuit	Net name	Num of sinks	Setup-I: LTTREE + PTREE			Setup-II: PTREE + Buffer Insertion			Setup-III: MERLIN			
			Area *1000 λ^2	Delay (ns)	Runtime (s)	Area	Delay	Runtime	Area	Delay	Runtime	Loops
C432	net1	16	58	38.54	22	0.33	0.87	0.36	0.28	0.39	25.09	2
	net2	16	83	35.49	41	0.27	0.71	1.66	0.69	0.48	5.24	1
	net3	10	51	32.19	44	1.31	0.88	4.27	0.56	0.70	15.27	7
C1355	net4	9	35	26.69	16	0.64	0.88	1.88	0.82	0.57	3.00	4
	net5	9	16	23.42	15	0.80	0.95	0.86	3.80	0.47	2.33	5
	net6	13	29	25.42	14	0.33	0.95	3.43	0.56	0.30	78.00	6
C3540	net7	12	58	41.03	29	0.50	0.88	1.79	1.44	0.55	23.59	12
	net8	35	93	47.05	99	0.17	0.83	4.42	0.17	0.49	7.92	1
	net9	73	214	60.73	229	1.55	0.69	1.83	0.12	0.42	1.98	1
C5315	net10	49	70	40.29	302	0.64	0.78	2.34	0.36	0.33	6.09	2
	net11	21	80	38.20	111	1.12	0.66	1.02	0.40	0.26	4.32	4
	net12	50	128	58.79	829	0.65	0.53	0.64	0.20	0.27	13.20	9
C6288	net13	16	58	44.65	52	0.83	0.73	1.12	2.11	0.49	9.33	5
	net14	20	58	45.67	28	0.67	0.91	1.71	1.00	0.73	3.54	1
	net15	60	90	90.29	197	0.25	0.74	1.42	0.29	0.55	16.20	4
C7552	net16	12	54	32.20	26	1.35	0.90	3.00	1.18	0.54	12.38	2
	net17	16	58	31.35	54	0.94	0.86	1.11	1.56	0.39	9.72	5
	net18	23	54	38.38	43	0.35	0.91	2.16	0.29	0.39	5.70	1
					Average:	**0.71**	**0.81**	**1.95**	**0.88**	**0.46**	**13.49**	

Table 1: Total buffer area, delay, and runtime for a number of individual nets.

Circuits	Flow-I: LTTREE + PTREE			Ratios normalized w.r.t. Flow I					
				Flow-II: PTREE + Buffer Insertion			Flow-III: MERLIN		
	Area*1000 λ^2	Delay (ns)	Runtime (s)	Area	Delay	Runtime	Area	Delay	Runtime
C1355	3630	8.18	1276	0.97	0.97	0.99	0.93	0.72	2.23
C1908	7768	14.47	2560	1.03	1.10	0.95	1.02	0.80	2.55
C2670	9428	12.40	1699	0.99	0.99	1.09	1.06	0.96	2.05
C3540	15762	22.17	5436	1.21	1.57	0.79	1.27	0.88	0.98
C432	3574	10.13	1382	1.16	1.06	0.79	1.57	1.00	1.17
C6288	28497	52.94	13547	0.96	1.03	0.88	1.00	0.90	1.00
C7552	35189	19.80	9250	0.78	1.06	0.95	0.85	0.74	1.36
Alu4	8191	15.69	2842	1.22	0.99	0.86	1.02	0.96	1.62
B9	1210	2.81	271	0.98	1.25	0.82	1.36	0.99	4.18
Dalu	10344	18.59	3465	0.73	0.88	0.66	0.88	0.67	1.74
Desa	32388	27.00	19427	1.12	1.12	0.75	1.19	0.82	0.83
Duke2	5499	9.00	2554	1.15	0.91	0.74	1.04	0.83	0.80
K2	22823	26.66	5831	0.85	0.75	1.73	0.93	0.63	2.56
Rot	8315	7.80	1572	0.91	1.02	0.83	1.00	0.81	3.40
T481	8917	10.12	5239	1.22	1.01	0.78	0.92	1.08	1.26
			Average:	**1.02**	**1.05**	**0.91**	**1.07**	**0.85**	**1.85**

Table 2: Post-layout area, delay, and runtime for a set of benchmark circuits.

Amir H. Salek (S'95-M'01) received the B.S. degree in Electrical Engineering from Sharif University of Technology, Tehran, Iran in 1995. He received the M.S. degree in Electrical Engineering and the Ph.D. degree in Computer Engineering from the University of Southern California, Los Angeles, California, in 1998 and 2000, respectively.

Since 2000, he has been with PMC-Sierra, Inc., working on the design and verification of Gigabit Ethernet, SONET, ATM and network processing integrated circuits. He has held short term appointments at Cadence Design Systems, Inc., and Magma Design Automation, Inc., in summers of 1996, 1997 and 1998. Dr. Salek is a recipient of the 2000 IEEE Circuits and Systems Society Outstanding Young Author Award.

Jinan Lou (S'99-M'00) received the B.S. degree in Computer Engineering and Computer Science, the M.S. and Ph.D. degrees in Computer Engineering, from the University of Southern California, Los Angeles, in 1993, 1995 and 1999, respectively.

He is currently a Sr. R&D Engineer in the Physical Compiler Product Group at Synopsys. His research interests include physical optimization, layout driven logic synthesis and post-layout optimization for deep-submicron technologies. Dr. Lou is also a member of the technical program committee for Intentional Symposium on Physical Design in 2002.

Massoud Pedram (S'88-M'90-SM'98-F'01) received a B.S. degree in Electrical Engineering from the California Institute of Technology in 1986 and M.S. and Ph.D. degrees in Electrical Engineering and Computer Sciences from the University of California, Berkeley in 1989 and 1991, respectively. He then joined the department of Electrical Engineering - Systems at the University of Southern California where he is currently a professor.

Dr. Pedram has served on the executive and technical program committee of a number of design conferences. He has published three books and more than 190 journal and conference papers. His research has received a number of awards including two Best Paper Awards from International Conference on Computer Design, a Distinguished Paper Citation from International Conference on Computer Aided Design, a Best Paper Award from the Design Automation Conference, and an IEEE Transactions on VLSI Systems Best Paper Award. He is a recipient of the NSF's Young Investigator Award (1994) and the Presidential Faculty Fellows Award (a.k.a. PECASE Award) (1996).

Dr. Pedram is an IEEE Fellow, a member of the Board of Governors for theIEEE Circuits and systems Society, an IEEE Solid State Circuits Society Distinguished Lecturer, a board member of the ACM Interest Group on Design Automation, and an associate editor of the IEEE Transactions on Computer Aided Design and the ACM Transactions on Design Automation of Electronic Systems.

His current work focuses on developing computer aided design methodologies and techniques for low power design, system-level dynamic power management, smart battery design and management, and integrated RT-level synthesis and physical design.

Hierarchical Buffered Routing Tree Generation

Amir H. Salek, *Member, IEEE*, Jinan Lou, *Member, IEEE*, and Massoud Pedram, *Fellow, IEEE*

Abstract--**This paper presents a solution to the problem of performance-driven buffered routing tree generation for VLSI circuits. Using a novel bottom-up construction algorithm and a local neighborhood search strategy, our polynomial time algorithm finds the optimum solution in an exponential-size solution subspace. The final output is a buffered rectilinear Steiner routing tree that connects the driver of a net to its sink nodes. The two variants of the problem, i.e., maximizing the required time at the driver subject to a maximum total area constraint and minimizing the total area subject to a minimum required time at the driver constraint, are handled by propagating three-dimensional solution curves during the construction phase. Experimental results demonstrate the effectiveness of our algorithm compared to other techniques.**

I. Introduction

The consideration of the effects of interconnect delay and area has become a crucial factor in the design of ultra-dense, high speed integrated circuits. In an industry where higher performance design brings substantial advantage over the competition, more and more time and resources are being spent on making faster chips through careful optimization of many design aspects, especially interconnect planning and optimization. In particular, the problem of constructing a buffered routing tree has emerged as a critical design problem.

The first part of this paper presents a new algorithm *FANROUT* that simultaneously solves the fanout optimization and routing tree construction problems. Both of these design tasks are difficult optimization problems and have considerable impact on the circuit delay and area. Fanout optimization is effective because it boosts the transmitted signal via the insertion of sized buffers whereas performance-driven routing generation is effective because it generates interconnect structures that deliver the signal to critical sinks faster. In conventional design flows, these two tasks are often performed in a sequential manner, i.e., a solution made by one optimization step becomes the input to the other. By solving the unified problem, i.e., generating a buffered routing tree for a set of sinks and a driver, the intrinsic interactions between the design steps are captured and higher quality results are produced by a systematic search in a much larger solution space. This type of solution technique is referred to as a *unification-based approach* in [Pe98].

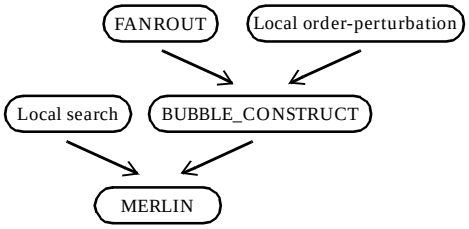

Fig. 1. Structure of MERLIN.

Similar to many other dynamic-programming based algorithms, FANROUT is only optimal with respect to a given order on its input objects (in this case the net sinks). This shortcoming is addressed in this paper by introducing a new technique called *local order-perturbation* which is used to enhance FANROUT. The resulting algorithm, *MERLIN*, is less sensitive to the input sink order with the cost of having a reasonably more complex computation.

The core optimization engine of MERLIN, called *BUBBLE_CONSTRUCT*, optimally solves the simultaneous routing and buffer insertion problem for a local neighborhood around an initial sink order. It recognizes the similar subsolutions among the members of the neighborhood in order to maintain the polynomial complexity of the algorithm. Although a complete buffered routing structure is not generated for every member of the neighborhood, the sink order that results in the best buffered routing structure is automatically chosen from among the members of the neighborhood. The outer optimization part of MERLIN (see Fig. 1) is an iterative technique based on a *local neighborhood search* strategy [Ya92].

Both FANROUT and BUBBLE_CONSTRUCT generate and propagate three-dimensional required time, load, and total area solution curves in a bottom-up fashion. In the three-dimensional solution curves, the load and the required time dimensions ensure the validity of the dynamic-programming principle [Be57] for solving the problem whereas the total area allows the user to solve the problem for either one of the following two variants: I) minimizing the required time subject to an area constraint or II) minimizing the area subject to a required time constraint.

The technique presented in this paper[1] offers the following advantages compared to prior methods:

- full integration of fanout optimization and routing tree generation using a dynamic-programming method,
- employment of a novel *local order-perturbation technique*

This research was sponsored in part by the NSF PECASE award number MIP-9628999.

A. H. Salek is with PMC-Sierra, Inc., Santa Clara, CA 95054 USA.

J. Lou is with Synopsys, Inc., Mountain View, CA 94043 USA.

M. Pedram is with the Department of Electrical Engineering-Systems, University of Southern California, Los Angeles, CA 90089 USA.

[1] The initial versions of the presented techniques have appeared in [SLP98] and [SLP99].

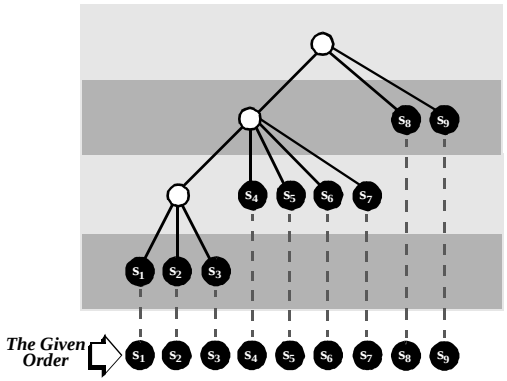

Fig. 2. An LT-Tree Type-I for a net with 9 sinks.

that enables the optimization engine to find (in polynomial time) the best buffered routing tree structure in an exponential-size sink order neighborhood of the initial order,

- propagation of *three-dimensional solution curves* that allows the algorithm to trade-off required time with total area and vice versa,
- definition and use of *Cα-Tree and *P-Tree structures* that expand the power of the optimization algorithms, resulting in highly optimized solutions,
- employment of the *local neighborhood search strategy* along with the core optimization algorithm to find the best solution in the neighborhood of the input sink. The resulting method is less sensitive to the initial order.

The remainder of the paper is organized as follows. In section II, prior work is given. Section III presents the problem formulation. Sections IV and V introduce FANROUT and the local order-perturbation technique. In section VI, MERLIN and its constituting elements are presented and discussed. Finally, sections VII and VIII give the experimental results and the concluding remarks, respectively.

II. Prior Work

A. Fanout Optimization

Fanout optimization, an operation performed in the logic domain, addresses the problem of distributing an electrical signal to a set of sinks with known loads and required times so as to maximize the required time at the signal driver (root of the net). Interconnect delays are either ignored or loosely modeled in this operation because the sink locations are not known at this stage. The general form of this problem is NP-hard [To90]; however, its restriction to some special families of topologies is known to have polynomial complexity.

Among many fanout optimization techniques - e.g., [Go76], [BCD89], [SS90], and [VP93] - the one proposed by [To90] has been proven to be very effective. That algorithm introduces a special class of tree topologies, called *LT-Tree*, for which the fanout problem is solved optimally with respect to a given order of sinks using dynamic programming. An *LT-Tree of type-I* is a tree that permits at most one internal node among the immediate children of its internal nodes and also does not allow any left sibling for the internal nodes (see Fig. 2).

Touati proposed a dynamic-programming based algorithm

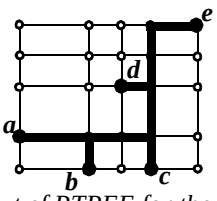

Fig. 3. An output of PTREE for the "dcba" order.

for the fanout optimization problem where the buffer structure is restricted to the LT-Tree topology and sinks with larger required times are placed farther from the root of the tree. The algorithm first sorts the sinks in their non-decreasing required time order and then, starting from the least critical sink, it enumerates all the left-most grouping of the sinks to be driven by a buffer. Finally for each grouping, it enumerates all possible ways of adding either zero or one buffer to drive the leftmost subset of the sinks. Touati gives sufficient conditions for the LT-Tree construction algorithm *LTTREE* to be optimal. For more details, see [To90].

Lemma 1: LT-Tree construction algorithm shows $O(n^2)$ complexity where n is the number of sink nodes [To90].

B. Routing Tree Construction

Performance-driven interconnect design, an operation performed in the physical domain, addresses the problem of connecting a signal driver to a set of sinks with known loads, required times and locations so as to maximize the required time at the driver. [CHKM96] gives a comprehensive review of the algorithms for solving this problem.

The inherent complexity of the problem has forced the researchers to focus on heuristic solutions and/or impose constraints on the structure of resulting interconnect. Among the recent works in this area, the algorithm presented by Lillis et al. in [LCLH96] has been shown to be quite effective. The authors proposed the Permutation-Constrained Routing Tree or *P-Tree* structure and solved the above problem with respect to the P-Tree structure; see Fig. 3 for an example. This approach consists of two major phases: I) heuristically finding a proper order for the sinks and II) generating the routing structure based on the order. The second phase of the algorithm is referred to as *PTREE* throughout this paper. Given an order for the sink nodes, PTREE finds the optimal embedding of the net into the *Hanan grid*[2] using a dynamic-programming approach. In PTREE, the intermediate routing solutions are stored in the form of two dimensional, non-dominated solution curves of total area versus required time for every *Hanan point*[3].

Lemma 2: For a given order on the sinks and with the restriction that the Steiner points lie on the Hanan points, PTREE computes the set of all rectilinear Steiner trees with non-dominated required time and total capacitance [LCLH96].

Lemma 3: If the individual capacitive values of wires and gate inputs are polynomially bounded integers or can be mapped to such with sufficient precision, then PTREE has $O(n^5q)$ *pseudo-polynomial complexity* (see [GJ79]), where n is

[2] The Hanan grid of a net is defined as the grid formed by the intersection of horizontal and vertical lines running through the terminals of the net [Ha66].

[3] Hanan points of a net are the vertices of the Hanan grid of the net.

the number of sink nodes and q is the maximum number of distinct load values [LCLH96].

Corollory 1: If the PTREE function is called with α sinks and uses k candidate locations instead of Hanan points, its complexity is $O(k\alpha^3 q)$.

C. Other Related Works

Lukas van Ginneken in [Gi90] proposed an algorithm to insert buffers on appropriate internal nodes of a given routing tree in order to maximize the required time at the driver. The application of van Ginneken's method after constructing the routing tree is usually more effective than applying fanout optimization followed by routing tree generation [SLP98].

The first attempts to combine fanout optimization and routing generation were presented in [OC96a] and [LCL96]. In [OC96a], the authors proposed a combination of A-Tree routing generation [CLZ93] and van Ginneken's buffer insertion [Gi90] methods. They later extended the work in [OC96b] to include wire sizing as well. Their algorithm takes the placement information of the source and the sinks in addition to the signal required times and heuristically generates a buffered routing structure that maximizes the required time at the source of the net. In these works, the subtrees are combined using a weighted addition function with a user-specified parameter to heuristically decide which two subtrees are to be merged. The algorithms in [OC96a] and [OC96b] have no guarantee of optimality. In [LCL96], Lillis et al. introduced a new version of PTREE which systematically solves the integrated problem of buffering and routing. That algorithm, called *B_PTREE* in the rest of this paper, uses a dynamic-programming formulation and generates three dimensional solution curves. Similar to PTREE, B_PTREE is optimal only with respect to a given sink order.

III. Problem Formulation

Given a net with $n+1$ pins, the problem is to drive the set of sink pins, $S=\{s_1, s_2, \ldots, s_n\}$, by the driver of the net s via a buffered routing structure that satisfies a combination of the maximum required time at the root and the minimum total area constraints. The area constraint can be stated in the form of total buffer area or total capacitance; the total capacitance is considered as a metric indicating the total buffer and interconnect area. More specifically, the problem may be stated in two ways: I) maximize the required time subject to an area constraint and II) minimize the area subject to a required time constraint.

The following information is provided as input:

1. The position of the source $s=(s^x,s^y)$ where s^x and s^y are the horizontal and vertical coordinates of s.
2. The properties of each sink node $s_i=(s_i^x,s_i^y,s_i^l,s_i^r)$ for $1\le i\le n$ where s_i^x and s_i^y are the horizontal and vertical coordinates, s_i^l is the capacitive load, and s_i^r is the signal required time of node s_i.
3. A library of buffers $B=\{b_1, b_2, \ldots, b_m\}$ containing m buffers with different strengths.

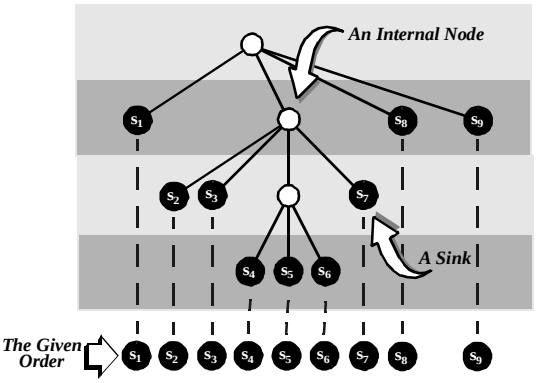

Fig. 4. A valid C4-Tree for $(s_1,s_2,\ldots,s_9)$.

4. A set of k candidate locations for placing the buffers $P=\{p_1, p_2, \ldots, p_k\}$.
5. A linear ordering of the sinks $\Pi=(s_1, s_2, \ldots, s_n)$.

There are many candidates for P; it can be the set of Hanan points [Ha66] (similar to what [LCLH96] has proposed) or a set of reserved buffer locations (identified after performing the initial placement step). Our experiments, in agreement with a conclusion made in [LCLH96], demonstrate that neither one of the above choices would significantly alter the final results as long the following two conditions are satisfied: I) k is large enough with respect to n and II) the candidate locations are distributed within the bounding box of the net with higher concentration in regions with a high density of sink pins.

IV. Order-Dependent Hierarchical Buffered Routing Tree Construction

This section presents FANROUT, an algorithm for solving the problem of simultaneous fanout optimization and routing generation. The resulting buffered routing tree contains a logical hierarchy that captures the hierarchical sink groups used during the construction. The hierarchy tree has a certain structure, which is formally defined below.

A. Cα-Trees

A desired property for a hierarchical algorithm is independence from any specific class of hierarchy graph structures. However, in many cases the complexity is so high that there is no choice but to restrict the solution space to a family of hierarchies. In this case, the problem is to identify and construct a set of structures that are consistent with the nature of the problem, both of which require a reasonable effort.

In this sub-section, a new class of structures, referred to as *Cα-Trees* (read as *si-alpha trees*), used to capture the hierarchy in the buffered routing construction algorithm is introduced.

Definition 1: A tree is a *degree-restricted alphabetic buffer chain tree* (*Cα-Tree*) for a given order of sinks $\Pi=(s_1,s_2,\ldots,s_n)$ if and only if:

- every internal node has at most one internal node among its immediate children,
- there is a depth-first traversal that visits the sinks in the $(s_1,s_2,\ldots,s_n)$ order,
- the maximum branching factor is α.

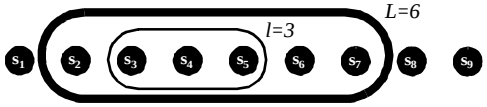

Fig. 5. Optimal Cα-Tree construction.

Fig. 4 illustrates an example for C4-Trees. In this figure the maximum branching factor is four and every internal node (shown by white circles) is connected to at most one other internal node while preserving the given order.

Lemma 4: In a Cα-Tree, the internal nodes construct a unique path (chain).

Proof: This is an immediate conclusion from Definition 1. ■

In this application, every internal node is a buffer, and in the resulting buffer chain, a more critical sink (considering both timing and physical information) tends to be connected closer (in terms of the number of intermediate stages) to the root.

Parameter α represents the maximum number of fanouts for every buffer or branching point. Our experience shows that even when no restriction is imposed on the maximum number of fanouts for each buffer, the maximum fanout count in the optimal buffer tree solution is usually bounded by a small number. That value is generally dependent on the characteristics of the cells (sink nodes) and the buffer library and not the problem size (number of sinks). Note that eliminating the parameter α from the definition does not cause the main structure and properties of Cα-Trees to breakdown. The only disadvantage is the longer (still polynomial) runtime needed for optimally constructing such a structure.

Although there are a large number of Cα-Trees for every sink order, the optimal Cα-Tree can be found in polynomial time using dynamic programming. Briefly, the optimal Cα-Tree for an ordered set of sinks is generated by starting from small *L*'s and combining every *L* neighboring sinks, until *L=n*. At every step, the best solutions for the sub-groups with length *l* (<*L*) are available - due to the bottom-up flow of the method - and are used to generate the solution for the length *L* sub-problem, see Fig. 5. Note that the final Cα-Tree structure satisfies the given sink order. This algorithm will be referred to as *CαTREE* in the rest of this paper.

Lemma 5: LT-Tree Type-I [To90] is a special case of Cα-Tree where $\alpha = +\infty$ and no internal node has a left sibling.

Proof: The proof directly follows the definitions of LT-Tree Type-I and Cα-Tree. ■

Note Cα-Trees can be relaxed with respect to the first property given in Definition 1, i.e., each internal node may have more than one internal node (but bounded by a certain parameter) among its immediate children. Although the optimal structure can still be achieved using dynamic programming, the complexity of the corresponding optimal construction algorithm is significantly higher.

B. FANROUT

FANROUT incorporates the Cα-Tree and P-Tree construction techniques into a unified framework such that the resulting routing structure is both Cα-Tree, in terms of the overall topology, and a P-Tree, in terms of the detailed physical

```
algorithm FANROUT( s, P, B, Π=(s_1 , s_2, ... , s_n) )
    INITIALIZATION
1.  for r = n downto 1
2.    foreach p∈P
3.      set Γ( 1, r, p ) = {The set of all non-inferior paths extended
        from p to s_r and driven with or without a buffer}
    CONSTRUCTION
4.  for L = 2 to n
5.    for R = n downto L
6.      for l = max(1, L-α+1) to L
7.        for r = R downto R-l+1
8.          foreach p∈P
9.            foreach γ ∈ Γ( l , r , p )
10.             set Δ = PTREE(P, {s_{R-L+1} , ... , s_{r-l} , γ , s_{r+1} , ... , s_R})
11.             foreach δ ∈ Δ
12.               set p' = Location of the root of δ
13.               foreach b ∈ B
14.                 set δ' = A buffered routing structure created by
                    driving δ by b located at p'
15.                 set < c, t, a > to the input capacitance, the input
                    required time and the area of δ', respectively
16.                 if < c, t, a > is a non-inferior solution in Γ(L , R , p')
17.                   insert < c, t, a > in Γ(L , R , p')
    EXTRACTION
18. find the solution ρ in Γ( n , n , s ) which best satisfies the constraints
19. retrieve the buffered routing tree structure ℜ of ρ by following the
    pointers stored during the generation of the solution curves
20. return ℜ
```

Fig. 6: Pseudo-code for FANROUT.

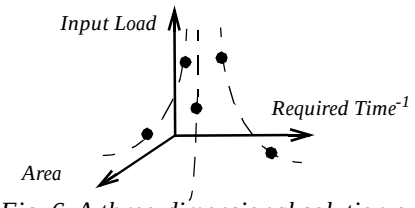

Fig. 6. A three-dimensional solution curve.

structure. FANROUT requires an ordering of the sinks and guarantees the optimality of the solution with respect to that ordering only. In the following paragraphs, the details of FANROUT are given.

FANROUT (see Fig. 6) is called with a set of parameters: *s*, *P*, *B*, and Π as defined in section III. It operates on three dimensional solution curves Γ(L, R, p) (see Fig. 6), each associated with a candidate buffer location *p* and a sub-group of sinks identified by variables *L* and *R*. *L* is the length of the sub-group and *R* indicates the position of the rightmost sink of the sub-group in Π . For example, if Π=(s_1, s_2, ..., s_9), Γ(*3, 7, p_i*) stores all the buffered routing structures that connect p_i to {s_5, s_6, s_7}.

In FANROUT, only non-inferior solutions - as defined below - are stored in the solution curves.

Definition 2: Suppose σ_1 and σ_2 are two different buffered routing structures that connect a candidate location to set of sinks. σ_2 is said to be inferior to σ_1, iff $load(\sigma_1) \le load(\sigma_2)$, $reqTime(\sigma_2) \le reqTime(\sigma_1)$, and $area(\sigma_1) \le area(\sigma_2)$.

As shown in Fig. 6, FANROUT consists of three main sections: *Initialization*, *Construction*, and *Extraction.* The Initialization section deals with creating and initializing solution curves corresponding to sub-problems consisting of only one sink, i.e., *L=1*. FANROUT is a dynamic-programming based technique and at each step it generates new curves by

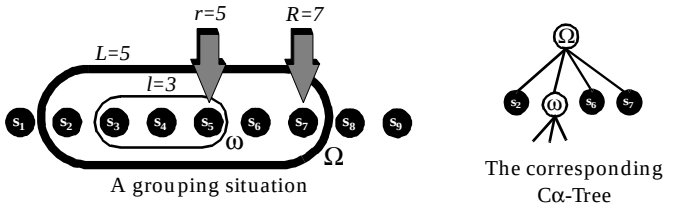

Fig. 7. An illustration for the grouping steps.

combining and manipulating already available curves for smaller sub-problems. In the Construction section, this bottom-up step is repeated until the solution curve for the main problem is found. Finally in the Extraction section, from among the solutions of the final Γ, the solution with the best trade-off between required-time and total area is chosen. At the end, the corresponding structure is generated by tracing back the pointers of the constituting sub-problems. The following is a detailed description of the algorithm.

1) Initialization: Before performing any operation, a set of solution curves are initialized in lines 1 through 3. In this part of FANROUT, sub-groups of length 1 are considered and the corresponding solution curves for every candidate buffer location and sink sub-group are initialized. These initial solutions consist of the minimum Manhattan distance paths from the candidate location p to the sink s_r . At the root of these paths, both options of inserting and not inserting a buffer are examined.

2) Construction: FANROUT starts by working on the groups consisting of only one sink, i.e., $L=1$, and proceeds until $L=n$. At each step, it constructs buffered routing structures that connect L neighboring sinks within Π. Line 4 enumerates all the possible values for L, and line 5 detects every legal sub-group Ω of Π that contains L sinks.

Every sub-group of sinks can potentially constitute an internal node in the final Cα-Tree structure; therefore, according to Definition 1, it can contain at most one other internal node (smaller sub-group) as its immediate child. During the process of grouping a set of L sinks, the algorithm considers cases in which a subset of sinks (call it ω) are already grouped; see Fig. 7 and lines 6 and 7 in the pseudo-code. That way, the Cα-Tree structure which captures the hierarchy of design is generated and maintained. In this context, the hierarchy implies that during the generation of a buffered routing structure, all the sinks are not processed at once; instead, a subset of sinks are combined together at any time in agreement with the Cα-Tree structure. Later, each combination is treated as one node in the next level of the hierarchy.

In line 6, the term *max(1,L-α-1)* ensures that Ω does not drive more than α other internal and sink nodes, following the third property of a Cα-Tree in Definition 1. In line 7, the term R to *R-l'+1* ensures that ω remains within Ω .

After line 7, it is known which sub-group ω is to be combined with which sink node(s) to generate the new sub-group Ω. However, there are many solutions associated with each ω. In fact, for every buffer candidate location p and sub-group ω there is a solution curve that is used by the merge operation; see Fig. 8. Line 8 enumerates all the candidate points, and line 9 retrieves the non-inferior solutions γ from a

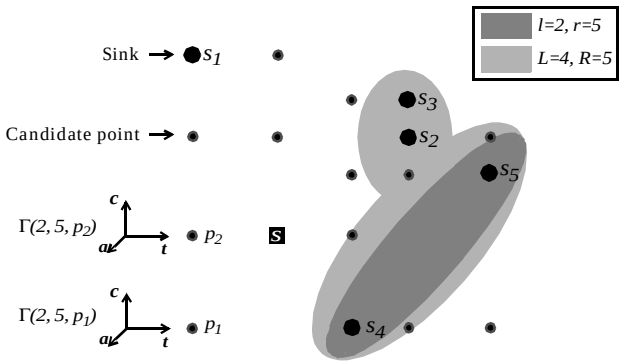

Fig. 8. Employing existing sub-solutions to generate larger sub-solutions.

solution curve of p which corresponds to ω.

In line 10, PTREE is called on the root of γ [4] and the remaining sinks in Ω in order to combine them by routing structures whose roots can be located at any candidate location. PTREE returns a collection of solution curves and stores them in Δ. Then, for every buffered routing structure in Δ, all the buffers in the library are used to drive its root, and the non-inferior combinations are stored in the corresponding solution curves; see lines 11 through 17. Along with every solution, a set of pointers is stored to be used during the extraction phase. The operations performed in lines 11 through 17 can be performed internally by a modified PTREE with no change in its worst case complexity. Hence, the complexity of those steps is not considered during the complexity analysis of FANROUT.

3) Extraction: The above bottom-up construction process continues until the solution curve for the whole problem, i.e., $L=n$, is generated. At that point, the solutions are stored in Γ(n, n, s) because they are rooted at s and are connected to all sinks. From among all the non-inferior solutions of Γ(n, n, s), the one which best satisfies the input constraints is chosen. The buffered routing structure corresponding to that solution is retrieved in lines 18 and 19 by following the stored pointers. Finally, in line 20 the best buffered routing structure is returned.

C. Quality and Complexity Analysis

FANROUT is an optimal polynomial algorithm based on a set of assumptions as explained previously and in the following lemmas and theorems.

Theorem 1: The solution space of FANROUT is the product of those of PTREE and CαTREE.

Proof: Analysis of the pseudo-code shows any P-Tree structure whose buffers directly drive at most one other buffer is considered by FANROUT. Also, any Cα-Tree whose buffers' output nets are implemented using PTREE is considered by FANROUT. ■

Lemma 6: The following statements are true for any routing structure ℜ that connects a source to a set of sinks:

I) By decreasing the load of any sink, the capacitance observed at the root of ℜ does not increase.

II) By increasing the required time of any sink, the required time at the root of ℜ does not decrease.

[4] PTREE sees the root of γ as a pseudo-sink whose required time and load are the ones of γ.

Proof: The above two statements are proven using simple circuit and graph theory rules. ■

Lemma 7: PTREE is monotone with respect to the load and the required time of the sinks.

Proof: This lemma is proven by induction and Lemma 6. ■

Lemma 8: The use of the prune operation by FANROUT does not result in the loss of any non-inferior solution.

Proof: Assume that σ_2 is inferior with respect to σ_1. By induction, if σ_2 is the whole net and its input is directly connected to the net driver, the required time does not decrease and the load does not increase by replacing σ_2 with σ_1. If σ_2 is a solution to a sub-problem, its input is driven by another internal node, called *g*. Due to the monotonic behavior of PTREE (c.f. Lemma 7), at *g* the required time and the input load of the implementation, including σ_2, is guaranteed to be no better than those of the implementation containing σ_1. A similar argument is then valid for *g* and the rest of the internal nodes down to the leaf nodes. ■

Theorem 2: FANROUT is an optimal algorithm.

Proof: An examination of the dynamic-programming structure of FANROUT shows that if no pruning is performed on the solution curves, all the possible solutions will be considered. Therefore, to prove the optimality of the algorithm it is enough to prove that for an optimal solution, replacing a non-inferior solution with an inferior solution cannot improve the whole implementation; this, however, was proven in Lemma 8. ■

Lemma 9: Depending on what metric is used for measuring the area, the number of solutions in a solution curve is bounded either polynomially or pseudo-polynomially.

Proof: The load of any solution is the input capacitance of the driving buffer. However, the number of distinct input capacitances of the buffers is bounded by the total number of available buffers in the library *m*.

In every solution the maximum number of inserted buffers is bounded by *O(n)*. Therefore, the number of distinct buffer areas is also bounded by *O(n)* since the area of every buffer is smaller than a constant number, and the smallest non-zero difference between the area of every two solutions is always greater than a constant number. Both these limits are determined by the library and do not depend on the size of the problem.

If the total buffer area is the metric used for measuring the area, the number of non-inferior solutions in a solution curve is bounded by *O(mn)*. The reason is that the prune operation keeps at most one solution per each distinct area and input load values.

The bound becomes pseudo-polynomial when the total capacitance is the metric used for measuring the area. In that case we assume that the number of distinct capacitive loads (called *q*) is polynomially bounded and is larger than the number of inserted buffers. As a result the number of non-inferior solutions is pseudo-polynomially bounded by *O(mq)*. ■

For the sake of simplicity, in the following theorem it is assumed that the total buffer area is used as the metric to measure the area cost of a solution.

Theorem 3: FANROUT has $O(kmn^3)$ memory complexity, where *k*, *m*, and *n* are numbers of candidate locations, buffers, and sinks, respectively.

Proof: There are *k* candidate locations, and for each combination of *L* and *R* (a total of *n(n+1)/2* combinations), there is a solution curve. Each solution curve stores *O(mn)* solutions, and as a result, the claim is proven. ■

Theorem 4: FANROUT has $O(mqk^2\alpha^5n^3)$ runtime complexity, where *k*, *m*, and *n* are the numbers of candidate locations, buffers, and sinks, respectively. Also, α is the maximum branching factor in Cα-Tree, and *q* is a polynomially bounded number of distinct capacitive loads.

Proof: In Fig. 6, the number of iterations performed in lines 4 through 7 is $O(\alpha^2n^2)$. Lines 8 and 9 introduce *O(k)* and *O(mn)* complexity, respectively. Calling PTREE in line 10 costs $O(k\alpha^3q)$, because the number of sinks provided to PTREE is always less than α ; see Corollory 1. The complexity of FANROUT is determined by considering all of the above factors. ■

V. Local Order-Perturbation

This section presents a new technique that can enhance any order-dependent dynamic-programming based algorithm - such as PTREE, CαTREE, and FANROUT - to generate optimal solutions with respect to a neighborhood of solutions.

Definition 3: An *order* Π on *n* sinks is a one-to-one function defined as $\Pi:\{1, 2, \ldots, n\}\rightarrow\{1, 2, \ldots, n\}$, and $j=\Pi(i)$ is called the *position of* s_i *in* Π. Also, Π^{-1} is the inverse function of Π, and $i=\Pi^{-1}(j)$ gives the sink's index of the *j*th element in Π.

Example 1: $\Pi=\{(1\rightarrow4), (2\rightarrow6), (3\rightarrow1), (4\rightarrow5), (5\rightarrow3), (6\rightarrow2), (7\rightarrow8), (8\rightarrow7), (9\rightarrow9)\}$, or equivalently, $(s_3,s_6,s_5,s_1,s_4,s_2,s_8,s_7,s_9)$ is an order on $\{s_1,s_2,\ldots,s_9\}$. Also, $\Pi(3)=1$ means s_3 is the first element in Π, and $\Pi^{-1}(2)=6$ means that s_6 is the second element in Π .

Although an algorithm that constructs an optimal structure for any given order is a useful tool, determining a "good" sink order remains as the main challenge. In the problem of buffered routing generation, required times, input loads, and physical locations of sink nodes should all be considered in generating a suitable order. Incorporating those independent and sometimes opposing parameters into the construction of an order is not an easy task. Due to the exponentially large number of possible orders, designers are forced to use heuristic approaches to combine the effects of those parameters in an ad-hoc manner. In general, the limitation imposed by working with one order at a time is very restrictive and undesirable.

The local order-perturbation method is a technique that works in a neighborhood of sink orders. No matter how one has determined an order, the semi-order-independent dynamic-programming formulation performs a systematic search in the neighborhood of that order. If the initial order is not a locally optimal order but close to it, this method chooses the optimal order automatically. The main advantage of such a technique is that it maintains an efficiency that is exponentially better than that of an exhaustive search method while preserving the optimality. Its superiority originates primarily from its

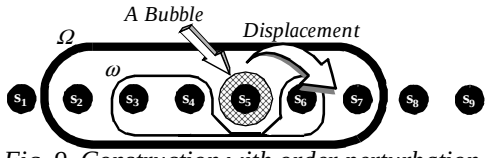

Fig. 9. Construction with order perturbation.

enhanced dynamic programming nature that enables the method to take advantage of all similar sub-problems among all the neighboring orders and thus avoid recomputing any sub-solution.

By allowing the bottom-up semi-order-independent algorithm to apply order perturbation operations, the sink order in the resulting solution can deviate from the initial order. A simple case is shown in Fig. 9 where the right-side border of a sub-group ω has been perturbed. Consequently, the order in the resulting group Ω is $(s_2,s_3,s_4,s_6,s_5,s_7)$ as opposed to the initial $(s_2,s_3,s_4,s_5,s_6,s_7)$ order; that is, in the new order s_5 has been swapped with s_6. In Fig. 9, s_5 has been left out from ω; it is called a *bubble*. When ω is used in a larger sub-group, it can be assumed that the bubble has been moved to the other side of the border of ω in the final structure. That operation causes the swapping of the position of two neighboring sinks.

Definition 4: For a set of sinks $\{s_1, s_2, \ldots, s_n\}$, the *neighborhood* of Π is defined as:

$$N(\Pi)=\{\Pi' \mid \forall s_i, |\Pi(i) - \Pi'(i)| \le 1\}.$$

In other words, the difference between the position of every s_i in Π and Π' is at most one.

Example 2: $\Pi'=(s_1,s_3,s_2,s_4,s_5,s_6,s_8,s_7,s_9)$ is in the neighborhood of $\Pi=(s_1,s_2,s_3,s_4,s_5,s_6,s_7,s_8,s_9)$, but $\Pi''=(s_3,s_2,s_1,s_4,s_5,s_6,s_7,s_8,s_9)$ is not in the neighborhood of Π.

Definition 5: For $n>1$, *displacing* the element i $(1\le i\le n-1)$ of Π (also referred to as the *displacement operation*) is defined as swapping the location of $s_{\Pi^{-1}(i)}$ with the location of $s_{\Pi^{-1}(i+1)}$ in Π. The displacement operation on element i always involves two neighboring elements in Π, i.e., $s_{\Pi^{-1}(i)}$ and $s_{\Pi^{-1}(i+1)}$. The two elements are called the *elements* of the displacement operation.

Definition 6: Two displacement operations are *non-overlapping* if and only if they have no element in common. A set of displacement operations are non-overlapping if and only if every two operations in the set are non-overlapping.

Example 3: Displacing the *4th* element of $\Pi'=(s_1,s_3,s_2,s_4,s_5,s_6,s_8,s_7,s_9)$ results in $\Pi''=(s_1,s_3,s_2,s_5,s_4,s_6,s_8,s_7,s_9)$

Lemma 10: Every $\Pi' \in N(\Pi)$ can be built from Π using a series of non-overlapping displacement operations.

Proof: This is a proof by induction. Let us represent a sub-string of Π that consists of the i left-most elements of Π by $sub\_string(\Pi, i)$. For $i=0$, it is trivial that $sub\_string(\Pi', 0)=sub\_string(\Pi, 0)$. Suppose for $i=\kappa-1$, $sub\_string(\Pi', \kappa-1)$ can be obtained from $sub\_string(\Pi, \kappa-1)$, using a series of non-overlapping displacement operations. Let $j=\Pi^{-1}(\kappa)$. Since $\Pi' \in N(\Pi)$, there are three neighborhood cases according to Definition 4.

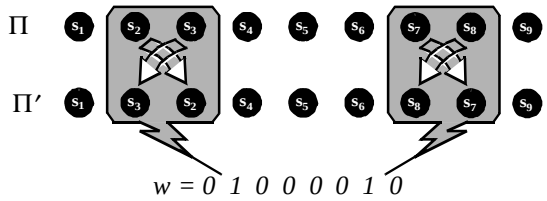

Fig. 10. Equivalence of W and N(Π).

- $\Pi(j)=\Pi'(j)$: This means that the κth elements in Π and Π' are the same. Therefore, the statement given in the above lemma holds for $i=\kappa$ as well.
- $\Pi(j)=\Pi'(j)-1$: This means that the $\kappa+1$th element in Π' is the same as the κth element in Π. Let $j'=\Pi'^{-1}(\kappa)$. In this case, we will have $\Pi(j')=\Pi'(j')+1$, otherwise $\Pi(j')-\Pi'(j')>1$ (in violation of Definition 4) because the $\Pi'(j')-1$ and $\Pi'(j')$ slots have already been taken by sinks other than $s_{j'}$. Therefore, we have a displacement at the κth element of Π, and the statement given in the above lemma holds for $i=\kappa+1$ as well.
- $\Pi(j)=\Pi'(j)+1$: This case cannot happen because it implies s_j is the $\kappa-1$th element of Π' which is in conflict with the above assumption. ■

Definition 7: Any arbitrary (n-1)-bit binary number w is called a *non-overlapping displacement code*, if and only if, it contains no two adjacent bits of 1. Also, W is defined as the set of all non-overlapping displacement codes.

Definition 8: The position of a bit b in a binary number w is defined as the number of bits on the left-side of b plus one.

Lemma 11: There exists a one-to-one relationship between the members of W and $N(\Pi)$.

Proof: First, we prove that for $\forall w \in W$ there exists a corresponding $\Pi' \in N(\Pi)$. For every 1 bit in w, set i to the position of that bit in w and displace the ith element of Π; call the resulting order Π'. Before the first displacement operation the inequality given in Definition 4 holds. Also, if the inequality between the initial order and the resulting order after j displacement operations is valid, it still holds after the $j+1$th displacement operation. That is because every displacement operation changes the location of the two swapped elements by ± 1 and keeps the location of the other elements unchanged. In addition, since w is non-overlapping code, no element is displaced more than once. Consequently, using induction we conclude that $\Pi' \in N(\Pi)$.

Now, we prove that for $\forall \Pi' \in N(\Pi)$ there exists a corresponding $w \in W$. According to Lemma 10, Π' can be generated from Π using a unique set of non-overlapping displacements. Those displacement operations can be coded in a non-overlapping displacement code which belongs to W. ■

Example 4: Fig. 10 illustrates the equivalence of W and $N(\Pi)$ for a simple example.

Theorem 5: For $n>1$, the number of distinct orders in the neighborhood of a given order Π is equal to:

$$\frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{n+2} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+2}\right)$$

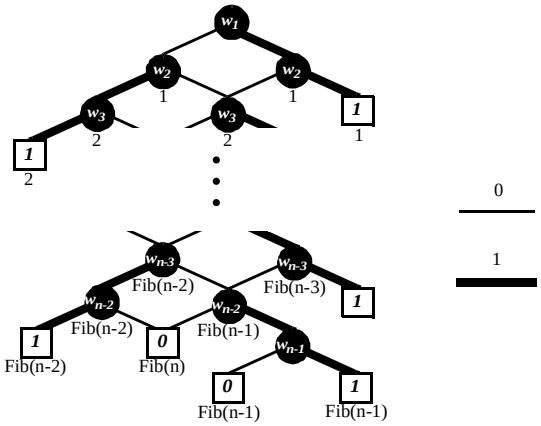

Fig. 11. BDD of f.

Proof: According to Lemma 11 there is a one-to-one relationship between $N(\Pi)$ and W. Therefore, these two sets have equal cardinality, and we can equivalently prove the above equation for W. So, we have to find out how many binary numbers in the form of $w=w_1w_2\ldots w_{n-1}$ exist that have no two adjacent 1s. The population of such numbers is equal to the off-set size of the following Boolean equation:

$$f=w_1w_2+w_2w_3+\ldots+w_{n-3}w_{n-2}+w_{n-2}w_{n-1}$$

Fig. 11 is the binary decision diagram (BDD) representation of the above equation. By induction, it can be verified that this structure is valid for $n>0$. In this figure, the number under each BDD node gives the number of distinct paths that exist from that node to the root.

Due to the symmetric structure of the equation and the corresponding BDD, the number of paths from a node to the root follows the Fibonacci number series. In the Fibonacci number series, the $k+1$th number is the sum of the k-1th and kth numbers in the series. As shown in Fig. 11, the number of distinct paths from the leaf node zero to the root is $2\times Fib(n)+Fib(n-1)$. Note that the factor of 2 appearing in the equation represents the fact that during the decomposition a zero sub-space has been reached while the decomposition is not yet over with respect to the last variable. By a few simple manipulations we get the final result:

$$2\times Fib(n)+Fib(n-1)=(Fib(n)+Fib(n-1))+Fib(n)$$
$$=Fib(n+1)+Fib(n)=Fib(n+2)$$

There is a direct closed-form for calculating the $n+2$th number in a Fibonacci series. According to the formula given in [MCS], we derive the equation given in the theorem. ■

Note the formula that returns the nth Fibonacci number involves square root of 5 (an irrational number), yet it always returns an integer for all (integer) values of n [MCS].

Theorem 5 proves that the size of $N(\Pi)$ is an exponential function of the number of sinks. Consequently, finding the best order in that sub-space of orders is a task of exponential complexity if a simple enumeration-based technique is used. However, all the common sub-solutions of different orders can be shared in a dynamic-programming based algorithm that utilizes the aforementioned idea of local order-perturbation. This in turn allows us to investigate the whole neighborhood in polynomial time.

Fig. 12. Grouping structures.

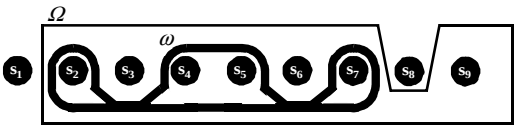

Fig. 13. Construction with perturbation.

Fig. 12 presents a set of *abstract grouping structures* $\{\chi_0, \chi_1, \chi_2, \chi_3\}$ by which one can cover a whole neighborhood of orders. χ_0 has no bubble on its sides, and χ_1, χ_2, and χ_3 have bubbles on the right-side, left-side, and both sides, respectively. For instance, the grouping ω of Fig. 9 is a χ_1-type structure. A full neighborhood is covered, if at each level of dynamic programming and for each sub-group of sinks, all the grouping structures are generated from all the grouping structures of their internal sub-groups; Fig. 13 shows an example.

Example 5: The example in Fig. 13 illustrates the use of χ_3 structure to generate a χ_1-type solution for Ω In this case, the resulting order is $(s_3,s_2,s_4,s_5,s_7,s_6,s_9)$. This new sub-solution will be used to generate larger sub-solutions that contain it.

The local order-perturbation technique can be extended to structures with more than one bubble on each side. Those structures in turn result in covering larger neighborhoods. However in that case, the number of grouping structures grows exponentially and, consequently, results in a significant slow-down of the corresponding construction algorithm.

VI. Semi Order-Independent Hierarchical Buffered Routing Tree Construction

In this section, the local order-perturbation theory is applied to FANROUT and B_PTREE (sub-sections A. and B.), and the resulting algorithm, MERLIN, is presented in sub-section C.

A. BUBBLE_CONSTRUCT

The technique presented in the following generates hierarchical buffered routing trees in a neighborhood of orders. The resulting hierarchies are consistent with the Cα-Tree structure, and in addition, the routing inside each layer of the Cα-Tree hierarchy is a P-Tree. Some parts of the BUBBLE_CONSTRUCT code (Fig. 14) are similar to the code of FANROUT and are not discussed again. It is assumed that the reader is familiar with the algorithm presented in section IV.

BUBBLE_CONSTRUCT operates on three dimensional solution curves Γ, each associated with a distinct set of values for l, r, p, and e. The first three variables are the same as the ones defined for FANROUT. The variable e encodes the grouping structure used to generate the solution curve.

1) Initialization: In this section, a set of solution curves are initialized. Here, sub-groups of length 1 are considered, and the corresponding solution curves for every candidate buffer location, sink, and grouping structure are initialized. Note that

algorithm *BUBBLE_CONSTRUCT(s, P, B,* $\Pi=(s_1, s_2, \ldots, s_n)$ *)*

INITIALIZATION

1. **for** $e = 0$ **to** 3
2. // similar to the lines 1 through 3 in Fig. 6 except that
   // $\Gamma(1, r, p)$ should be replaced with $\Gamma(1, r, p, e)$

CONSTRUCTION

3. **for** $L = 1$ **to** n
4. **for** $E = 0$ **to** 3
5. **set** $L' = L + STRETCH(L, E)$ // see Fig. 15
6. **for** $R = n$ **downto** L'
7. **set** $G = SINK\_SUBSET(\Pi, R, L, E)$ // see Fig. 16
8. **for** $l = \max(1, L\text{-}\alpha+1)$ **to** $L\text{-}1$
9. **for** $e = 0$ **to** 3
10. **set** $l' = l + STRETCH(l, e)$ // see Fig. 15
11. **for** $r = R$ **downto** $R\text{-}l'+1$
12. **set** $g = SINK\_SUBSET(\Pi, r, l, e)$ // see Fig. 16
13. **if** $g\text{-}G \neq \phi$ **continue**
14. **foreach** $p \in P$
15. **foreach** $\gamma \in \Gamma(l, r, p, e)$
16. **set** $G' = REORDER(G, g, \gamma)$
17. **set** Δ = *PTREE(P, B, G')
18. // similar to the lines 11 through 17 in Fig. 6 except that
    // $\Gamma(L, R, p')$ should be replaced with $\Gamma(L, R, p', E)$

EXTRACTION

19. // similar to the lines 18 through 20 in Fig. 6 except that
    // $\Gamma(n, n, s)$ should be replaced with $\Gamma(n, n, s, 0)$

Fig. 14. The pseudo-code for BUBBLE_CONSTRUCT.

algorithm *STRETCH(L, E)*

1. **set** $g=0$
2. **if** $L=2$ **and** $E>0$ **set** $g=1$
3. **else if** $L>2$ **and** $E=1$ **set** $g=1$
4. **else if** $L>2$ **and** $E=2$ **set** $g=1$
5. **else if** $L>2$ **and** $E=3$ **set** $g=2$
6. **return** g

Fig. 15. The pseudo-code for STRETCH.

algorithm *SINK_SUBSET(* $\Pi=(s_1, s_2, \ldots, s_n)$*, R, L, E)*

1. **if** $L=1$ **set** $G = \{ s_R \}$
2. **else if** $L=2$
3. **switch** E
4. **case** 0 : **set** $G = \{ s_{R-1}, s_R \}$
5. **case** $1, 2, 3$: **set** $G = \{ s_{R-2}, s_R \}$
6. **else**
7. $L'=STRETCH(L, E)$
8. **switch** E
9. **case** 0 : **set** $G = \{ s_{R-L'+1}, s_{R-L'+2}, s_{R-L'+3}, \ldots, s_{R-2}, s_{R-1}, s_R \}$
10. **case** 1 : **set** $G = \{ s_{R-L'+1}, s_{R-L'+2}, s_{R-L'+3}, \ldots, s_{R-2}, s_R \}$
11. **case** 2 : **set** $G = \{ s_{R-L'+1}, s_{R-L'+3}, \ldots, s_{R-2}, s_{R-1}, s_R \}$
12. **case** 3 : **set** $G = \{ s_{R-L'+1}, s_{R-L'+3}, \ldots, s_{R-2}, s_R \}$
13. **return** G

Fig. 16. The pseudo-code for SINK_SUBSET.

for sub-groups with length 1, all four grouping structures (χ_0 , χ_1 , χ_2 , and χ_3) are the same; however, for the sake of simplicity in the rest of the pseudo-code, separate (although similar) solution curves are generated for each case. A similar situation occurs for χ_1 and χ_2 where $L=2$.

2) Construction: The main difference between this algorithm and FANROUT is in the grouping phase, i.e., lines 3 through 13 in Fig. 14. BUBBLE_CONSTRUCT starts from $L=1$ and goes up to $L=n$. For each new sub-group of sinks, all possible grouping structures (coded by numbers *0* to *3*) are enumerated in line 4. For the case of χ_0 ($E=0$), the length of the sub-group is equal to L, but for the other cases the actual length of the sub-group is larger by one or two units in order to capture the effect of inserting one or two bubbles on the sides. This new length is

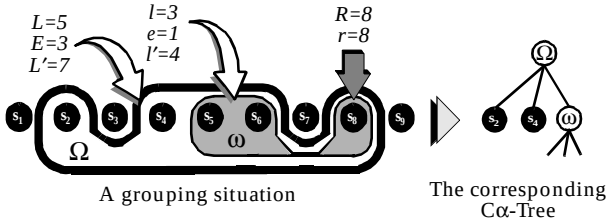

Fig. 17. A legal grouping scenario for BUBBLE_CONSTRUCT.

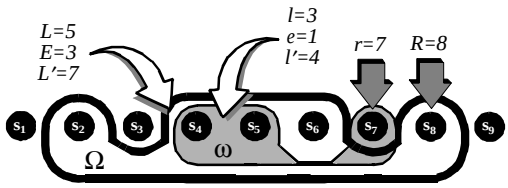

Fig. 18. An illegal grouping case for

calculated and stored in L' (refer to line 5 and Fig. 15). In line 6, all the possible sub-strings of length L' are considered from the right to the left of Π. In fact, the variable R points to the right-most element of the sub-strings of L' elements.

Lines 8 through 11, similar to lines 3 through 6, investigate all possible sub-group lengths with different grouping structures and positions which fit inside the sub-group being constructed. Fig. 17 illustrates an example where a sub-group of 5 sinks, Ω, is being generated using a combination of an already generated sub-group of 3 sinks, ω, and two other sinks, i.e., s_2 and s_4.

It can be seen that in some cases Ω and ω are not compatible. As an example, consider the situation shown in Fig. 18 where the difference between the values of r and R causes the grouping structure of ω to not fit in the grouping structure of Ω. Those cases are detected and skipped in line 13 of the pseudo-code. Note that sets G and g - calculated in lines 7 and 12 - represent the sets of sinks included in Ω and ω, respectively (also refer to Fig. 16).

In line 16, REORDER updates G by replacing all the sinks that belong to g with a pseudo-sink that represents the root of γ (a solution to ω). The resulting order is called G′.

In line 17, an enhanced version of B_PTREE (called *PTREE) is called to generate a new set of solutions for all candidate locations. Every solution created by *PTREE shows the combination of ω with the rest of sink nodes of Ω. The details of *PTREE are presented in the following sub-section.

3) Extraction: In this section, a solution from $\Gamma(n, n, s, 0)$ that best satisfies the input constraints is selected and reconstructed by tracing back the stored pointers.

The quality and complexity of BUBBLE_CONSTRUCT is further discussed in sub-section D.

*B. *PTREE*

*PTREE is a solution to the problem of non-hierarchical buffered routing tree construction. As mentioned earlier, *PTREE is called by BUBBLE_CONSTRUCT within each level of the Cα-Tree hierarchy. Consequently, *PTREE is responsible for conducting the order-perturbation task within each level of the hierarchy, otherwise BUBBLE_CONSTRUCT would not be optimal with respect to

```
algorithm *PTREE(P, B, Π=(s1 , s2, ... , sn))
    INITIALIZATION
1.  // the same as the one in BUBBLE_CONSTRUCT, see Fig. 14
    CONSTRUCTION
2.  for L = 2 to n
3.    for E = 0 to 3
4.      set L′ = L + STRETCH(L, E); // see Fig. 15
5.      for R = n downto L′
6.        set G = SINK_SUBSET( Π, R, L, E ); // see Fig. 16
7.        for l1 = 1 to L-1
8.          for e1 = 0 to 3
9.            set l1′ = l1 + STRETCH(l1, e1); // see Fig. 15
10.           set r1 = R
11.           set g1 = SINK_SUBSET( Π, r1, l1, e1 ); // see Fig. 16
12.           if g1-G ≠ ϕ continue;
13.           set l2 = L - l1
14.           switch e1
15.             case 0, 1 : set r2 = r1-l1′
16.               switch E
17.                 case 0, 1 : set e2 = 0
18.                 case 2, 3 : set e2 = 2
19.             case 2, 3 : set r2 = r1-l1′+2
20.               switch E
21.                 case 0, 1 : set e2 = 1
22.                 case 2, 3 : set e2 = 3
23.           set g2 = SINK_SUBSET( Π, r2, l2, e2 ); // see Fig. 16
24.           if g2-G ≠ ϕ or g2-g2 ≠ ϕ continue;
25.           // combine Γ( l1, r1, p, e1 ) and Γ( l2, r2, p, e2 ) in
              // the same way PTREE combines the solution curves
```

*Fig. 19. The pseudo-code for *PTREE.*

the complete neighborhood of orders.

The algorithm is an enhanced version of B_PTREE [LCL96] with two main differences: I) it uses the local order-perturbation technique, and as a result, it is optimal with respect to a neighborhood of orders and II) as an input it takes a set of candidate buffer locations, and therefore, the locations of buffers and Steiner points are not restricted to Hanan points.

At every step of dynamic programming in *PTREE, a sub-group of sinks Ω is solved using the existing solutions to two smaller sub-groups (ω_1 and ω_2) which partition Ω into two segments. Fig. 20 illustrates a legal grouping situation in which ω_1 and ω_2 properly partition Ω

For the design of *PTREE, three issues have been considered:

- ω_1 and ω_2 do not share any sinks,
- ω_1 and ω_2 cover all (and only) the sinks of Ω,
- All the possible combinations of grouping structures and sizes must be considered for ω_1 and ω_2.

The pseudo-code of *PTREE is given in Fig. 19. In lines 2 through 5, all combinations of size *L*, grouping structure *E*, and position *R* are constructed for Ω . Then, ω_1 is constructed in lines 7 through 10. As shown in the code, the rightmost element of ω_1 is always the rightmost element of Ω . Some combinations of Ω and ω_1 may not be legal, which means Ω does not contain at least one element of ω_1 . Although those cases could be avoided during the construction of ω_1, for the sake of presentation, they are explicitly pruned by the condition in line 12. In that line, any ω_1 incompatible with Ω is detected and disregarded.

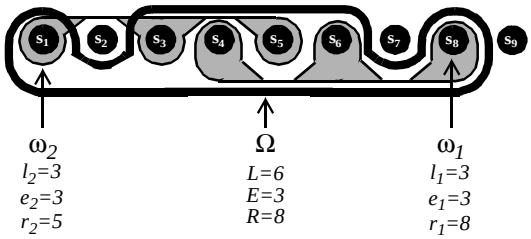

*Fig. 20. A legal grouping scenario for *PTREE.*

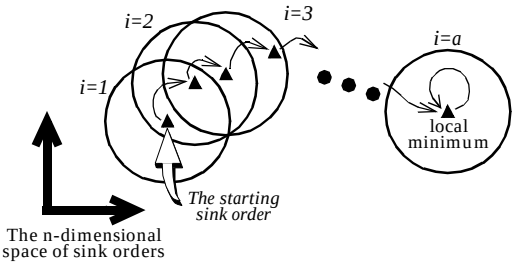

Fig. 21. Local neighborhood search in MERLIN.

.In lines 13 through 22, ω_2 is generated by considering Ω and ω_1. The size, grouping structure, and position of ω_2 are determined so that ω_2 contains all the sink nodes of Ω that are not included in ω_1 . If ω_1 has a bubble on its left side, i.e., grouping structures χ_2, χ_3, that bubble is the rightmost element of ω_2; see line 19. Similarly, if Ω has a bubble on its left-side, ω_2 should have a grouping structure with a bubble on its left, and so on. For more details refer to the pseudo-code in Fig. 19. Again, some illegal cases may occur but are pruned in line 24.

After line 24, ω_1 and ω_2 are known and their solution curves are combined the same way that B_PTREE combines the solution curves. Interested readers may refer to [LCL96] for the details of B_PTREE.

C. MERLIN

This sub-section presents MERLIN, a local neighborhood search algorithm, which employs BUBBLE_CONSTRUCT to find a local optimum sink order and the optimum buffered routing tree corresponding to that order.

Generally, an optimization problem has a set of solutions and a cost function that assigns a value to every solution. The goal is to find an optimal solution, i.e., one that has the minimum (or maximum) cost. The local neighborhood search, as a member of iterative solution methods, is a widely-used, general approach for solving optimization problems.

To obtain a local search *(LS)* algorithm for solving an optimization problem, one superimposes a neighborhood structure on the solutions, i.e., for each solution a set of neighboring solutions is specified. This LS algorithm starts from some initial solution, which may be constructed by some other algorithm or generated randomly, and from then on keeps moving to a better neighboring solution, until finally it terminates at a locally optimal solution. This method has been applied both in the context of continuous and discrete optimizations [Ya92]. In general, *simulated annealing* is a special case of local neighborhood search that allows uphill moves. Fig. 21 illustrates the behavior of a local neighborhood

```
algorithm MERLIN( s , P, B, Π=(s_1 , s_2 , … , s_n) )
1.  set Π′ = Π
2.  do {
3.    set Π = Π′
4.    set ℜ = BUBBLE_CONSTRUCT( s , P , B , Π )
5.    set Π′ = SINK_ORDER( ℜ )
6.  } while ( Π != Π′ )
7.  return ℜ
```

Fig. 22. The pseudo-code for MERLIN.

search.

Definition 9: A function $N:F \rightarrow 2^F$, which associates a subset $N(x)$ with each $x \in F$, is a *neighborhood function* over F iff $\forall x \in F$, $x \in N(x)$ and $\forall x \in F$, $x \in N(y) \Rightarrow y \in N(x)$.

BUBBLE_CONSTRUCT induces a well-defined neighborhood function in which it finds the best solution. The same definition is also used by MERLIN.

Lemma 12: The properties required by Definition 9 are consistent with those of neighborhood introduced in Definition 4.

Proof: In Theorem 5, we proved that the size of the neighborhood, $N(\Pi)$, is always greater than *1*, independent of the choice of Π. Also, for every $\Pi' \in N(\Pi)$ there is a unique non-overlapping displacement code, *w*, that transforms Π to Π'. To prove that $\Pi \in N(\Pi')$also, we need to prove that there is a non-overlapping displacement code, *w′*, that transforms Π' to Π. It can be shown that $w'=w$ is, in fact, the solution. ■

There exist at least two sink orders , i.e., Π and Π', in common between the neighborhood of two consecutive iterations of MERLIN's local search (see Fig. 22). In fact, this overlap, $OVERLAP(N(\Pi),N(\Pi'))$, is often relatively large. Intuitively, when the corresponding non-overlapping displacement code has more 1s, $OVERLAP(N(\Pi),N(\Pi'))$ is smaller. Obviously, it is a waste to consider the overlapping sub-space twice. This can be prevented by keeping solution curves of the very last iteration. For similar sub-problems simply copy the corresponding solution curve between the two iterations. However, this speed-up is achieved at the cost of doubling memory usage.

D. Quality and Complexity Analysis

Theorem 6 : *PTREE executes in $O(k\alpha^3 q)$ where *k* is the total number of buffer candidate points, α is the number of sinks, and *q* is a polynomially bounded number of distinct capacitive loads.

Proof: In Fig. 19, lines 2, 5, and 7 each introduce $O(\alpha)$ complexity. Note that in the pseudo-code, *n* is the number of sinks that is referred to as α in this theorem. The merge operation (line 25), which is the same as in B_PTREE, has a $O(kq)$ complexity [LCL96]. ■

Lemma 13: Orders generated by BUBBLE_CONSTRUCT are in the neighborhood of the initial order.

Proof: This is a proof by induction. The pseudo-code directly forces the grouping structures to cover each other like nested shells. Starting from the innermost shell, we analyze the effect of grouping structures. Case *i=1*: after the bubble-out step (see Fig. 9), for the innermost grouping structure, the order of all the sinks remains unchanged except for the two which are on the border of the bubbled sub-group. Consequently, the

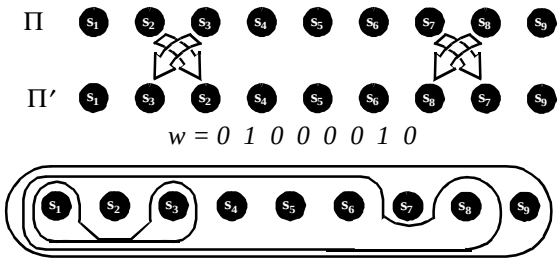

Fig. 23. An illustration for the proof of Lemma 14.

inequality relation in Definition 4 remains valid, and the resulting order is within the neighborhood of the initial order. Case *i=n:* suppose that after the bubble-out step for the *n-1* innermost grouping structures, the inequality of Definition 4 still holds. The order for the sinks on the border of the *n*th grouping structure must still be unchanged because no overlap is allowed between the borders of two grouping structures. Therefore, even after the bubble-out step for the *n*th grouping structure, the resulting order is within the neighborhood of the initial order. ■

Lemma 14: Any $\Pi' \in N(\Pi)$, is considered by BUBBLE_CONSTRUCT.

Proof: BUBBLE_CONSTRUCT implicitly tries all the possible valid combinations of grouping structures on Π. Therefore, it is enough to prove that $\forall \Pi' \in N(\Pi)$ there exists a combination of grouping structures that result in that order. Suppose that *w* is the non-overlapping displacement code of $\Pi' \in N(\Pi)$, as given in Definition 7. Starting from the left-most 1-bit in *w* (*j* is the position of that bit in *w*) extend a χ_1-type sub-group from the left-most sink to the *j+1*th sink. After the bubble-out step the resulting order is similar to Π' for the *j* left-most sinks and similar to Π for the rest of the sinks. Repeat this operation for the next bit 1 in *w* in order from left to right. There are no two neighboring 1s in *w*; therefore at each step the left portion of the resulting order resembles Π' and the other portion is like Π . At the last step when there is no 1 left in *w*, we cover the initial order from left to right with a χ_0-type sub-group. The resulting order, after the bubble-out step for all the sub-groups, is Π', and since it has a valid grouping structure it is considered by the pseudo-code of Fig. 14. ■

The example in Fig. 23 illustrates the proof of Lemma 14.

Lemma 15: Any identical sub-problem among the members of $N(\Pi)$ is shared and processed only once.

Proof: Any sub-problem is uniquely identified by *l*, *e*, and *r* values. $\forall p \in P$, $\Gamma(l,e,r,p)$ is generated only once, no matter in which compatible and larger grouping structure it will be used later. Note that according to Lemma 13 and Lemma 14, BUBBLE_CONSTRUCT covers the whole space of $N(\Pi)$. ■

Theorem 7: The solution space of BUBBLE_CONSTRUCT is the product of the spaces of *P-Tree and Cα-Tree for the neighborhood of the initial given order.

Proof: $\forall \Pi' \in N(\Pi)$, all the corresponding Cα-Trees with the boundary of their sub-groups on the displaced sinks' locations are visited and for every one of them all the *P-Tree structures are considered by *PTREE. However, there are some Cα-Trees that correspond to Π' whose displacements are not at

the boundary of the sub-groups. *PTREE considers all the necessary displacements inside one layer of those Cα-Tree. ■

Lemma 16: BUBBLE_CONSTRUCT is monotone with respect to required time, load, and area.

Proof: By considering that *PTREE is monotone with respect to the required time, load, and area, we can conclude that in a Cα-Tree, decreasing the load of either an internal or a sink node results in the decrease of load in its immediate parent. A similar argument is valid for required time and total area. ■

Lemma 17: In BUBBLE_CONSTRUCT, the pruning operation does not eliminate any non-inferior solution.

Proof: The proof follows Lemma 16 and Definition 2. ■

Theorem 8: Subject to restrictions imposed by the *P-Tree and Cα-Tree structures, BUBBLE_CONSTRUCT finds all the non-inferior solutions with respect to required time and total area in the neighborhood of a given order.

Proof: If no pruning is performed all the space is explicitly constructed (see Theorem 7). Lemma 17 states that the prune operation drops the sub-solutions that are only used in inferior solutions. Therefore, all the non-inferior solutions remain in the final curves of BUBBLE_CONSTRUCT. ■

For the sake of simplicity, in the following it is assumed the total buffer area is the metric used to measure solution area.

Theorem 9: BUBBLE_CONSTRUCT has $O(kmn^3)$ memory complexity where k, m, and n are numbers of candidate locations, buffers, and sinks, respectively.

Proof: The proof is the same as for Theorem 3. The only difference is that the number of solution curves is four times higher in BUBBLE_CONSTRUCT than in FANROUT, since for every grouping structure a solution curve is stored. ■

Theorem 10: BUBBLE_CONSTRUCT has $O(mqk^2\alpha^5n^3)$ runtime complexity where k, m, and n are the number of candidate locations, buffers, and sinks, respectively. Also, α is the maximum branching factor in Cα-Trees, and q is polynomially bounded number of distinct capacitive loads.

Proof: The proof is similar to that of Theorem 4. ■

Corollary 2: Assuming that m, q, and α are parameters independent from the size of the problem n and are determined by the library and technology, the effectual worst-case complexity of BUBBLE_CONSTRUCT is $O(k^2n^3)$.

Theorem 11: The cost associated with orders produced by iterations of MERLIN (but the last one) is strictly decreasing.

Proof: BUBBLE-CONSTRUCT always returns the best order in the neighborhood; thus if a different order is returned, it must correspond to a lower cost. In the last iteration, the cost of the given order is the best in the neighborhood, and that is how the iteration is terminated. ■

VII. Experimental Results

In this section, three experimental setups have been tested and compared on a set of benchmark circuits.

- *Setup-I:* For every net, fanout optimization using LTTREE is followed by a routing tree construction phase using PTREE. In LTTREE, the net sinks are sorted with respect to their required times. However, in PTREE the net sinks are sorted by a solution to the TSP (Traveling Salesman Problem) using the method suggested in [LCLH96].
- *Setup-II:* Routing tree generation using PTREE is followed by buffer insertion using the van Ginneken's method [Gi90]. The sink order for PTREE is again the TSP order.
- *Setup-III:* Finally, hierarchical buffered routing generation is performed using MERLIN and an initial TSP order.

All the experiments have been implemented and executed in SIS [SSLM92] and on a dual-processor Ultra-2 Sun Sparc workstation with 256MB memory. In these experiments, an industrial standard cell library (0.35μm CMOS process) consisting of 34 buffers has been used. Gate and wire delays are calculated using a 4-parameter delay equation and the Elmore delay model [El48], respectively.

A. Comparison on Individual Nets

Table 1 reports the results of running the above three experimental setups on 18 individual nets randomly selected from a set of benchmark circuits. For every extracted net, the sink locations are determined randomly in a bounding box. The size of the box has been determined such that the delay of a wire segment whose length is half the perimeter of the box is approximately equal to the delay of an average gate driving that wire. In addition, the load and required time sink data have been selected randomly from a nominal range.

In Table 1, the reported area and delay values are the total buffer area and the maximum delay at the root of the net in the resulting buffered routing structures. Also, the runtimes have been reported in seconds for every net and setup. Note, the data of Setup-I has been reported in absolute values; however, for the other two setups the results have been scaled with respect to their corresponding data in Setup-I.

For each net, the last column in Table 1 reports the number of iterations performed during the execution of MERLIN. For about 28% of the cases reported in the table, MERLIN converges in one iteration. That indicates that the initial sink order is a local minimum in its neighborhood. This effect can be used as a metric to measure the effectiveness of the heuristics used to generate the initial order. The experiments indicate that the TSP heuristic tends to perform better with respect to this metric compared to a few other heuristics. Hence, the TSP order has been used in all experimental setups.

B. Comparison on Circuits

Table 2 reports the post-layout total area and delay values for a set of benchmark circuits. In these experiments the above three setups have been plugged into a full design flow that extends from the logic synthesis all the way down to the detailed routing. The resulting design flows have been named *Flow-I*, *Flow-II*, and *Flow-III*, respectively. Again, the data for Flow-I is the absolute to which the rest are scaled.

VIII. Conclusions

In this paper, the problem of distributing a signal among a set of sinks with different placement, load, and required time values has been addressed. The proposed technique generates a

set of non-inferior buffered routing structures that provide different trade-offs between the required-time at the root and the total buffer area. The introduced solution consists of an iterative optimization block that uses a local neighborhood search strategy and an optimization engine based on dynamic programming that generates all the non-inferior structures in the neighborhood of a given sink order. This optimization engine generates and propagates 3-dimensional solution curves and employs a novel local order-perturbation method to cover an exponentially sized solution space in polynomial time. The experimental results show significant delay improvement with little area penalty compared to the conventional buffer and routing tree generation techniques.

ACKNOWLEDGEMENTS

The authors wish to thank Dr. John Lillis of the University of Illinois at Chicago for helpful discussions and comments.

REFERENCES

[Be57] R. Bellman, *Dynamic Programming*. Princeton University Press, 1957.

[BCD89] C. L. Berman, J. L. Carter, and K. F. Day, "The fanout problem: From theory to practice," In *Proceedings of Advanced Research on VLSI*, pp. 69-99, 1989.

[CHKM96] J. Cong, L. He, C. Koh, and P. Madden, "Performance optimization of VLSI interconnect layout," In *Integration, the VLSI Journal 21*, pp. 1-94, 1996.

[CLZ93] J. Cong, K. Leung, and D. Zhou, "Performance-driven interconnect design based on distributed RC delay model," In *Proceedings of Design Automation Conference*, pp. 606-611, 1993.

[El48] W. C. Elmore, "The transient response of damped linear network with particular regard to wideband amplifiers," In *Journal of Applied Physics* 19, pp. 55-63, 1948.

[Go76] M. C. Golumbic, "Combinatorial merging," *IEEE Transactions on Computers*, vol. 25, pp. 1164-1167, Nov. 1976.

[Gr92] L. K. Grover, "Local search and the local structure of NP-complete problems," In *Operations Research Letters 12*, pp. 235-243, Oct. 1992.

[Gi90] L.P.P.P. van Ginneken, "Buffer placement in distributed RC-tree networks for minimal Elmore delay," In *Proceedings of International Symposium on Circuits and Systems*, pp. 865-868, 1990.

[GJ79] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco: W. H. Freeman, 1979.

[Ha66] M. Hanan, "On Steiner's problem with rectilinear distance," *SIAM Journal of Applied Mathematics*, No. 14, pp. 255-265, 1966.

[LCL96] J. Lillis, C. K. Cheng, and T. Y. Lin, "Simultaneous routing and buffer insertion for high performance interconnect," In *Proceedings of the Sixth IEEE Great Lakes Symposium on VLSI*, pp. 148-153, 1996 and *Proceedings of ACM/SIGDA Physical Design Workshop*, pp. 7-12, 1996.

[LCLH96] J. Lillis, C. K. Cheng, T. Y. Lin, and C. Ho, "New performance driven routing techniques with explicit area/delay tradeoff and simultaneous wire sizing," In *Proceedings of the 33rd Design Automation Conference*, pp. 395-400, 1996.

[LSP97] J. Lou, A. H. Salek, and M. Pedram, "An exact solution to simultaneous technology mapping and linear placement problem," In *Proceedings of International Conference on Computer-Aided Design*, pp. 671-675, 1997.

[MCS] http://www.mcs.surrey.ac.uk/Personal/R.Knott/Fibonacci/fibFormula.html .

[OC96a] T. Okamoto and J. Cong, "Buffered Steiner tree construction with wire sizing for interconnect layout optimization," In *Proceedings of International Conference on Computer-Aided Design*, pp. 44-49, 1996.

[OC96b] T. Okamoto and J. Cong, "Interconnect layout optimization by simultaneous Steiner tree construction and buffer insertion," In *Proceedings of ACM/SIGDA Physical Design Workshop*, pp. 1-6, 1996.

[Pe98] M. Pedram, "Logical-physical co-design for deep submicron circuits: challenges and solutions," In *Proceedings of Asia and South Pacific Design Automation Conference*, pp. 137-142, Feb. 1998.

[SLP98] A. H. Salek, J. Lou, and M. Pedram, "A simultaneous routing tree construction and fanout optimization algorithm," In *Proceedings of International Conference on Computer-Aided Design*, 1998.

[SLP99] A. H. Salek, J. Lou, and M. Pedram, "MERLIN: Semi-order-independent hierarchical buffered routing tree generation using local neighborhood search," In *Proceedings of Design Automation Conference*, pp. 472-478, 1999.

[SS90] K. J. Singh and A. Sangiovanni-Vincentelli, "A heuristic algorithm for the fanout problem," In *Proceedings of Design Automation Conference*, pp. 357-360, 1990.

[SSLM92] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli, "SIS: A system for sequential circuit synthesis," *Memorandum No. UCB/ERL M92/41*, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, May 1992.

[To90] H. Touati, "Performance-oriented technology mapping," Ph.D. thesis, *University of California, Berkeley, Technical Report UCB/ERL M90/109*, Nov. 1990.

[VP93] H. Vaishnav and M. Pedram, "Routability-driven fanout optimization," In *Proceedings of Design Automations Conference*, pp. 230-235, 1993.

[WM89] W.S. Wong and R.J.T. Morris, "A new approach to choosing initial points in local search," In *Information Processing Letters 30*, pp. 67-72, Jan. 1989.

[Ya92] M. Yannakakis, "The analysis of local search problems and their heuristics," In *Proceedings of the 7'th Annual Symposium on Theoretical Aspects of Computer Science*, pp. 298-311, 1990.

						Ratios normalized w.r.t. Setup I						
Taken from circuit	Net name	Num of sinks	Setup-I: LTTREE + PTREE			Setup-II: PTREE + Buffer Insertion			Setup-III: MERLIN			
			Area *1000 λ^2	Delay (ns)	Runtime (s)	Area	Delay	Runtime	Area	Delay	Runtime	Loops
C432	net1	16	58	38.54	22	0.33	0.87	0.36	0.28	0.39	25.09	2
	net2	16	83	35.49	41	0.27	0.71	1.66	0.69	0.48	5.24	1
	net3	10	51	32.19	44	1.31	0.88	4.27	0.56	0.70	15.27	7
C1355	net4	9	35	26.69	16	0.64	0.88	1.88	0.82	0.57	3.00	4
	net5	9	16	23.42	15	0.80	0.95	0.86	3.80	0.47	2.33	5
	net6	13	29	25.42	14	0.33	0.95	3.43	0.56	0.30	78.00	6
C3540	net7	12	58	41.03	29	0.50	0.88	1.79	1.44	0.55	23.59	12
	net8	35	93	47.05	99	0.17	0.83	4.42	0.17	0.49	7.92	1
	net9	73	214	60.73	229	1.55	0.69	1.83	0.12	0.42	1.98	1
C5315	net10	49	70	40.29	302	0.64	0.78	2.34	0.36	0.33	6.09	2
	net11	21	80	38.20	111	1.12	0.66	1.02	0.40	0.26	4.32	4
	net12	50	128	58.79	829	0.65	0.53	0.64	0.20	0.27	13.20	9
C6288	net13	16	58	44.65	52	0.83	0.73	1.12	2.11	0.49	9.33	5
	net14	20	58	45.67	28	0.67	0.91	1.71	1.00	0.73	3.54	1
	net15	60	90	90.29	197	0.25	0.74	1.42	0.29	0.55	16.20	4
C7552	net16	12	54	32.20	26	1.35	0.90	3.00	1.18	0.54	12.38	2
	net17	16	58	31.35	54	0.94	0.86	1.11	1.56	0.39	9.72	5
	net18	23	54	38.38	43	0.35	0.91	2.16	0.29	0.39	5.70	1
					Average:	**0.71**	**0.81**	**1.95**	**0.88**	**0.46**	**13.49**	

Table 1: Total buffer area, delay, and runtime for a number of individual nets.

Circuits	Flow-I: LTTREE + PTREE			Ratios normalized w.r.t. Flow I					
				Flow-II: PTREE + Buffer Insertion			Flow-III: MERLIN		
	Area*1000 λ^2	Delay (ns)	Runtime (s)	Area	Delay	Runtime	Area	Delay	Runtime
C1355	3630	8.18	1276	0.97	0.97	0.99	0.93	0.72	2.23
C1908	7768	14.47	2560	1.03	1.10	0.95	1.02	0.80	2.55
C2670	9428	12.40	1699	0.99	0.99	1.09	1.06	0.96	2.05
C3540	15762	22.17	5436	1.21	1.57	0.79	1.27	0.88	0.98
C432	3574	10.13	1382	1.16	1.06	0.79	1.57	1.00	1.17
C6288	28497	52.94	13547	0.96	1.03	0.88	1.00	0.90	1.00
C7552	35189	19.80	9250	0.78	1.06	0.95	0.85	0.74	1.36
Alu4	8191	15.69	2842	1.22	0.99	0.86	1.02	0.96	1.62
B9	1210	2.81	271	0.98	1.25	0.82	1.36	0.99	4.18
Dalu	10344	18.59	3465	0.73	0.88	0.66	0.88	0.67	1.74
Desa	32388	27.00	19427	1.12	1.12	0.75	1.19	0.82	0.83
Duke2	5499	9.00	2554	1.15	0.91	0.74	1.04	0.83	0.80
K2	22823	26.66	5831	0.85	0.75	1.73	0.93	0.63	2.56
Rot	8315	7.80	1572	0.91	1.02	0.83	1.00	0.81	3.40
T481	8917	10.12	5239	1.22	1.01	0.78	0.92	1.08	1.26
			Average:	**1.02**	**1.05**	**0.91**	**1.07**	**0.85**	**1.85**

Table 2: Post-layout area, delay, and runtime for a set of benchmark circuits.

Amir H. Salek (S'95-M'01) received the B.S. degree in Electrical Engineering from Sharif University of Technology, Tehran, Iran in 1995. He received the M.S. degree in Electrical Engineering and the Ph.D. degree in Computer Engineering from the University of Southern California, Los Angeles, California, in 1998 and 2000, respectively.

Since 2000, he has been with PMC-Sierra, Inc., working on the design and verification of Gigabit Ethernet, SONET, ATM and network processing integrated circuits. He has held short term appointments at Cadence Design Systems, Inc., and Magma Design Automation, Inc., in summers of 1996, 1997 and 1998. Dr. Salek is a recipient of the 2000 IEEE Circuits and Systems Society Outstanding Young Author Award.

Jinan Lou (S'99-M'00) received the B.S. degree in Computer Engineering and Computer Science, the M.S. and Ph.D. degrees in Computer Engineering, from the University of Southern California, Los Angeles, in 1993, 1995 and 1999, respectively.

He is currently a Sr. R&D Engineer in the Physical Compiler Product Group at Synopsys. His research interests include physical optimization, layout driven logic synthesis and post-layout optimization for deep-submicron technologies. Dr. Lou is also a member of the technical program committee for Intentional Symposium on Physical Design in 2002.

Massoud Pedram (S'88-M'90-SM'98-F'01) received a B.S. degree in Electrical Engineering from the California Institute of Technology in 1986 and M.S. and Ph.D. degrees in Electrical Engineering and Computer Sciences from the University of California, Berkeley in 1989 and 1991, respectively. He then joined the department of Electrical Engineering - Systems at the University of Southern California where he is currently a professor.

Dr. Pedram has served on the executive and technical program committee of a number of design conferences. He has published three books and more than 190 journal and conference papers. His research has received a number of awards including two Best Paper Awards from International Conference on Computer Design, a Distinguished Paper Citation from International Conference on Computer Aided Design, a Best Paper Award from the Design Automation Conference, and an IEEE Transactions on VLSI Systems Best Paper Award. He is a recipient of the NSF's Young Investigator Award (1994) and the Presidential Faculty Fellows Award (a.k.a. PECASE Award) (1996).

Dr. Pedram is an IEEE Fellow, a member of the Board of Governors for theIEEE Circuits and systems Society, an IEEE Solid State Circuits Society Distinguished Lecturer, a board member of the ACM Interest Group on Design Automation, and an associate editor of the IEEE Transactions on Computer Aided Design and the ACM Transactions on Design Automation of Electronic Systems.

His current work focuses on developing computer aided design methodologies and techniques for low power design, system-level dynamic power management, smart battery design and management, and integrated RT-level synthesis and physical design.