# MERLIN: Semi-Order-Independent Hierarchical Buffered Routing Tree Generation Using Local Neighborhood Search*

Amir H. Salek, Jinan Lou, Massoud Pedram
Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089
{ amir, jlou, massoud } @sahand.usc.edu

**ABSTRACT - This paper presents a solution to the problem of performance-driven buffered routing tree generation in electronic circuits. Using a novel bottom-up construction algorithm and a local neighborhood search strategy, this method finds the best solution of the problem in an exponential size solution subspace in polynomial time. The output is a hierarchical 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 driver required time subject to a total buffer area constraint and minimizing the total buffer area subject to a minimum driver required time constraint, are handled by propagating three-dimensional solution curves during the construction phase. Experimental results prove the effectiveness of this technique compared to the other solutions for this problem.**

## I. INTRODUCTION

This paper presents a solution for simultaneously solving fanout optimization and routing tree generation problems. Both of these design tasks are difficult optimization problems which have a considerable effect on reducing the circuit delay. Fanout optimization is effectual by boosting the transmitted signal via insertion of sized buffers whereas performance-driven routing generation is effective by generating suitable wire structures. In conventional design flows, these two tasks are often performed in a sequential manner. Consequently, a solution made by one of these optimizations becomes a constraint for the other. This flow reduces the flexibility and impact of these operations. Solving the unified problem, i.e. generating a buffered routing tree for a set of sinks and a driver, helps capture the intrinsic interactions between the combined design steps and produces higher-quality implementations by systematically searching a much larger solution space. This type of solution technique is referred to as a *unification-based approach* [SLP98].

The core optimization engine proposed in this paper, named *BUBBLE_CONSTRUCT*, optimally solves the aforementioned problem for a local neighborhood of an initial sink order. It exploits all the similar sub-solutions among the members of the neighborhood in order to reduce the time complexity of the algorithm. Although a complete buffered routing structure is not generated for every member of the neighborhood, the sink order which results in the best buffered routing structure is automatically chosen from among the members of the neighborhood. *MERLIN*, an iterative optimization method based on the idea of local neighborhood search, takes this new sink order and uses it as the input for the next call to *BUBBLE_CONSTRUCT*. Experimental results reported in this paper prove that this method converges very quickly for most practical examples. *BUBBLE_CONSTRUCT* uses an enhanced version of P_Tree [LCLH96], called **P_Tree*, and generates and propagates three-dimensional required time and load versus total buffer area solution curves in a bottom-up fashion. In the three-dimensional solution curves, the existence of the load and the required time dimensions ensure the validity of the principle of dynamic programming [Be57] for solving the problem. The third dimension (total buffer area) allows the user to solve the problem for either one of the following variants: I) minimizing the required time subject to an area constraint, II) minimizing the area subject to a required time constraint. The *P_Tree structure is used in a certain hierarchy, called *Cα_Tree*, which is formally defined in this paper.

The remainder of the paper is organized as follows. In section II, prior work is given. Section III introduces the proposed algorithm and its constituting building blocks. In sections IV and V, our experimental results and concluding remarks are presented.

## II. PRIOR WORK

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 delay is not incorporated in this operation because the locations of sinks 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 the fanout optimization algorithms, the one proposed by [To90] introduced a special class of tree topologies, called *LT-Tree*, for which the fanout problem is solved using dynamic programming with polynomial complexity. A *LT-Tree of type-I* (see Figure 4) 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.

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 positions so as to maximize the required time at the driver. [CHKM96] gives a thorough overview of the algorithms for solving this problem.

The inherent complexity of this problem has forced researchers to either solve it heuristically or to impose constraints on the structure of the resulting interconnect. Among the recent works in this area, the algorithm presented by Lillis et al. in [LCLH96] should be mentioned. They proposed the Permutation-Constrained Routing Tree or *P-Tree* structure and solved the above problem with respect to the P-Tree structure, see Figure 1. Their approach consists of two major phases: finding a proper order for the sinks heuristically, and then 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*† using a dynamic programming approach. In PTREE, the routing solutions are stored in the form of two dimensional, non-inferior solution curves of total area versus required time for every *Hanan point* (the vertices of the Hanan grid).

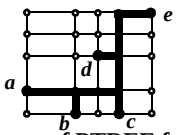


**Figure 1: An output of PTREE for "*dcba*" order**

* This work was funded in part by SRC under contract no. 98-DJ-606 and by NSF contract no. MIP-9628999.


DAC 99, New Orleans, Louisiana


†. 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].

**Lemma 1:** If the individual capacitive values are polynomially bounded integers or can be mapped to such with sufficient precision, PTREE has $O(n^5q)$ *pseudo-polynomial complexity (*see [GJ79]) where $n$ is the number of sink nodes and $q$ is the maximum number of distinct load values [LCLH96].

Local neighborhood search as a member of iterative solution methods is a widely used, general approach to solving hard optimization problems. To obtain a local search framework for an optimization problem, one superimposes a neighborhood structure on the solutions, i.e. for each solution a set of neighboring solutions is specified. This method starts from some initial solution that is constructed by some other algorithm, or generated randomly, and from then on it keeps moving to a better neighboring solution as long as there is one, until finally it terminates at a locally optimal solution for which there is no better neighbor. This method has been applied both in the context of continuous and discrete optimization [Ya92]. In general, *simulated annealing* is a special case of local neighborhood search that sometimes allows uphill moves.

**Definition 1:** A function $N:F \rightarrow 2^F$, which associates a subset $N(x)$ with each $x \in F$, is a *neighborhood function* over $F$, if $|N(x)|$ is larger than *1* and $\forall x \in F, x \in N(y) \rightarrow y \in N(x)$.

In our method, the optimization engine induces a well-defined neighborhood function (see Definition 4) in which the optimization algorithm optimally finds the best solution. That definition of neighborhood is used by MERLIN to conduct a local search.

# III. MERLIN

## III.1 Problem Formulation

For a given net, the problem is to drive a set of sink nodes, $S=\{s_1, s_2, \ldots, s_n\}$, by the driver of the net, $s$, via a buffered routing structure. The objective is to generate a buffered routing structure that satisfies a combination of the maximum required time at the root and the minimum total buffer area constraints. More specifically, the problem may be stated in two ways: I) minimize the required time subject to an area constraint, II) minimize the area subject to a required time constraint.

The following information is required and used by the proposed algorithm:

1. The position of the source, $s=(s^x,s^y)$, where $s^x$ and $s^y$ are the horizontal and the 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 required time at node $s_i$.
3. A library of buffers, $B=\{b_1, b_2, \ldots, b_m\}$, containing $m$ buffers with different strengths.
4. A set of $k$ *candidate locations* for placing buffers, $P=(p_1, p_2, \ldots, p_k)$.
5. A linear ordering of the sinks, $(s_1, s_2, \ldots, s_n)$.
6. Two parameters $\gamma$ and $\alpha$, to be described in the next subsections.

There are many choices for $P$: it can be the set of Hanan points [Ha66] (similar to what has been proposed in [LCLH96]), a set of reserved buffer locations (generated by the placement phase), or the center of masses of some subsets of sinks. Our experiments, in agreement with a conclusion made in [LCLH96], demonstrate neither one of the above choices would alter the final result significantly as long as $k$ is large enough with respect to $n$, e.g. $k$ is a linear function of $n$.

## III.2 Basic Elements

### *3.2.1 Cα_Trees*

In this sub-section, we introduce a new class of trees, called *Cα_Trees* (read as ***si-alpha trees***) which is used to capture the hierarchy in the buffered routing construction algorithm presented later in this paper.

**Definition 2**: A tree is a *degree-restricted alphabetic buffer chain tree* (*Cα_Tree)* for a given order of sinks - say $(s_1,s_2,\ldots,s_n)$ - iff:

- every internal node has at most one internal node among its immediate children,
- at every internal node, the branching edges are ordered, so as to preserve the order of sink nodes under the internal node,
- the maximum branching factor is α.

Note that in any Cα_Tree, a reverse depth-first search (respecting the immediate children order at every internal node) visits the sinks in the $(s_1,s_2,\ldots,s_n)$ order.

Figure 2 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.

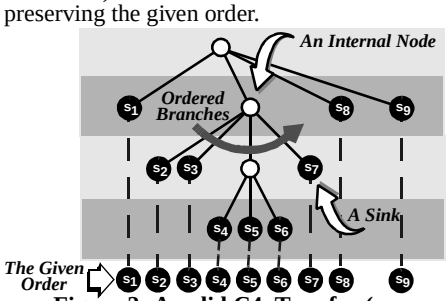


**Figure 2: A valid C4_Tree for $(s_1,s_2,\ldots,s_9)$**

**Lemma 2:** In a Cα_Tree, the internal nodes construct a unique path (chain).

In our application, every internal node is a buffer and in the resulting buffer chain (c.f Lemma 2), a less critical sink (considering both timing and physical information) tends to be connected to the farthest (in terms of the number of intermediate stages) buffers from the root in the chain.

The parameter α represents the maximum number of fanouts for every buffer. Our experience shows that for structures with no restriction on the maximum number of fanouts for each buffer, the maximum fanout count is usually bounded by a certain value which is dependent on the library parameters and not the problem size (number of sinks). By eliminating the parameter α from the definition, the main structure and properties of Cα_Trees do not breakdown. In that case, the only disadvantage would be larger (still polynomial) runtime needed for optimally constructing such a structure.

Although there is a large number of Cα_Trees for every sink order, the optimal Cα_Tree can be found in a 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 $L'$ (smaller than $L$) 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 Figure 3. Note that the final Cα_Tree structure satisfies the given sink order.

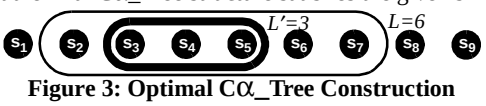


**Figure 3: Optimal Cα_Tree Construction**

**Lemma 3:** LT_Tree Type-I (see Figure 4) [To90] is a special case of Cα_Tree where $\alpha = +\infty$ and no internal node has a left sibling (see Figure 4).

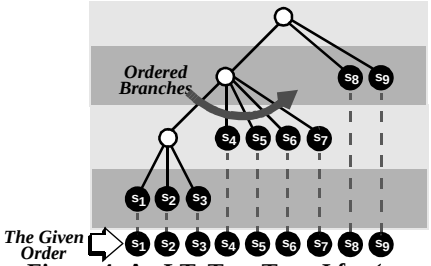


**Figure 4: An LT_Tree Type-I for $(s_1,s_2,\ldots,s_9)$**

Note Cα_Trees can be relaxed with respect to the first property given in Definition 2, 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 grows significantly.

*3.2.2 Local Order-Perturbation (Bubbling)*

Some NP-Complete problems become solvable by dynamic programming when an order (see Definition 3) is imposed on their elements. In that case, the final solution is optimal only with respect to that specific order. The works presented in [LCLH96] and [To90] are two examples for this case.

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

**Example 1:** $\Pi$= { $(1\rightarrow4)$ , $(2\rightarrow5)$ , $(3\rightarrow2)$ , $(4\rightarrow1)$ , $(5\rightarrow3)$ , $(6\rightarrow6)$ , $(7\rightarrow8)$ , $(8\rightarrow7)$ , $(9\rightarrow9)$} or equivalently $(s_4,s_3,s_5,s_1,s_2,s_6,s_8,s_7,s_9)$ is an order on $\{s_1,s_2,\ldots,s_9\}$.

In this paper, the idea of *local order-perturbation (Bubbling)* is introduced and discussed in the context of Cα_Tree and *P_Tree (to be introduced in sub-section 3.2.3) construction. However, its extension to other applications is possible and rather straightforward.

Although an algorithm which constructs an optimal structure for any given order is a useful tool, the main difficulty of the problem remains in how to come up with a "good" sink order such that the resulting structure demonstrates superior properties. In the problem of buffered routing generation, required times, input loads, and physical locations of sink nodes should be all considered for generating an appropriate order. How we incorporate those independent and sometimes opposing parameters in an order is a question that does not have an easy solution. The exponential number of possible orders forces us to use either a heuristic which combines the effect of those parameters in an ad-hoc fashion or an iterative method which tries a subset of orders. In either case, the limitation imposed by working with one order at a time is very restrictive.

The local order-perturbation is a technique that works in a neighborhood of sink orders. No matter how we come up with an order (heuristically or by iteration), our semi-order-independent dynamic programming formulation performs a systematic search in the neighborhood of that order. If the initial order is not a local/global optimal structure but is close to it, this method generates the local/global optimal structure automatically. The main advantage of such technique is its efficiency while preserving the optimality which is considerably better than that of an exhaustive search method. Its superiority primarily originates from its enhanced dynamic programming nature that enables the method to take advantage of all similar sub-problems among all the neighboring orders and avoid recomputing the sub-solutions.

By allowing the bottom-up technique to make perturbations, the sink order in the resulting solution can deviate from the initial order. A simple case is shown in Figure 5, where the right-side border of a sub-group ($L'$) has been perturbed (c.f. Figure 3). Consequently, the order in the resulting sub-group ($L$) 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; in the new order $s_5$ has been swapped with $s_6$. The hole in the right-side of $L'$ is called a *bubble* (see Figure 5) and when $L'$ is used in a larger sub-group, the bubble is moved to the other side of the corresponding border of $L'$ (this operation is called *Bubble Out*).

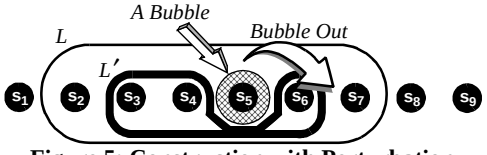


**Figure 5: Construction with Perturbation**

**Definition 4:** For a set of sinks $\{s_1, s_2, \ldots, s_n\}$, the *neighborhood* of $\Pi$ 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 $\Pi$ and $\Pi'$ 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 $\Pi=(s_1,s_2,s_3,s_4,s_5,s_6,s_7,s_8,s_9)$.

**Definition 5:** If $n>1$, swapping the element $i$ $(1\le i\le n-1)$ of $\Pi$ is defined as swapping the value of $\Pi(\Pi^{-1}(i))$ with $\Pi(\Pi^{-1}(i+1))$. In other words, it means the location of $s_{\Pi^{-1}(i)}$ is swapped with the location of $s_{\Pi^{-1}(i+1)}$.

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

**Lemma 4:** Every $\Pi' \in N(\Pi)$ can be built from $\Pi$ using a series of non-overlapping swap operations.

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

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

The above formula involves square root of 5 (an irrational number) yet it always gives an integer for all (integer) values of $n$.

Theorem 1 proves 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 an exponential complexity task, if a simple enumeration-based technique is used. However, all the common sub-solutions of different orders can be shared in a dynamic programming algorithm that utilizes the aforementioned idea of bubbling. This in turn allows us to investigate the whole neighborhood in a polynomial time.

In Figure 5, we noticed that if we allow bubbles on the sides of sub-groups we can alter the resulting sink order. Figure 6 presents a set of *abstract grouping structures* $\{\chi_0, \chi_1, \chi_2, \chi_3\}$ which cover a whole neighborhood of orders. $\chi_0$ has no bubble on its sides and $\chi_1$, $\chi_2$, and $\chi_3$ have bubbles on the right-side, left-side, and both sides, respectively. For instance, the grouping $L'$ of Figure 5 is a $\chi_1$-type structure. A full neighborhood would not be covered, unless 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; Figure 7 illustrates one example.


**Figure 6: Grouping Structures**

**Example 4:** The example in Figure 7 illustrates the use of $\chi_3$ structure to generate a $\chi_1$-type solution for $L$. 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.


**Figure 7: Construction with Perturbation**

The algorithm proposed in sub-section III.3 (Figure 9) contains the pseudo-code for the construction of perturbed Cα_Trees (lines 5 to 13). Lemma 5 and Lemma 6 prove that for any given sink order every member of the neighborhood can be made by the above grouping structures and also every combination of the grouping structures results in a valid order in the neighborhood.

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 that consequently results in a significant slow down in the corresponding construction algorithm.

*3.2.3 Buffered P_Tree (*P_Tree)*

PTREE [LCLH96] finds the best rectilinear routing embedded in the Hanan grid of all sinks for a given sink order. In this sub-

section, we will present an enhanced version of PTREE, called **PTREE*, which has the following properties:

- generating rectilinear buffered routing tree structures with buffers located on the routing Steiner points,
- generating and propagating three-dimensional curves to allow trade-off between required time and total buffer area
- working on a neighborhood of orders using the idea of local order-perturbation.

The buffered routing structures generated by *PTREE which are basically P_Trees with the possibility of having buffers at the Steiner points are referred to as *P_Tree in this paper.

*PTREE starts with an ordered set of sinks $\Pi=(s_1,s_2,\ldots,s_\alpha)$ and a given set of candidate locations, $P=\{p_1, p_2,\ldots,p_k\}$. As the base case, it generates

$$S(e,p,i,i)\ \forall p\in P,\ 0<i<\alpha,\ \text{and}\ 0\le e\le 3$$

*solution curves* (see Figure 8) which are a collection of minimum Manhattan distance routings from $p$ to $s_i$, driven with or without a buffer. All buffers of the given library $L$ are tried to drive the routing structure and for each of them, the required time and the load at the root as well as the total buffer area (*0* if the structure uses no buffer) are measured. These solutions and their corresponding attributes are compared against each other and in each $S(e,p,i,i)$ only the non-inferior (see Definition 6) solutions are stored. Variable $e$ encodes the grouping structures that is being considered. Note that for the base case, all $\chi_0$, $\chi_1$, $\chi_2$, and $\chi_3$ structures result in the same structure.

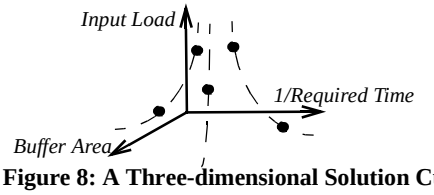


**Figure 8: A Three-dimensional Solution Curve**

**Definition 6:** Suppose $\sigma_1$ and $\sigma_2$ are two buffered routing structures that connect a candidate location to the same subset of sinks. $\sigma_2$ is said to be inferior to $\sigma_1$, iff $load(\sigma_1)\le load(\sigma_2)$, $reqTime(\sigma_2)\le reqTime(\sigma_1)$, and $area(\sigma_1)\le area(\sigma_2)$.

Consequently, *PTREE generates three dimensional curves for sub-groups consisting of sinks $s_i$ to $s_j$ in $\Pi$. These solution curves are calculated using the following recursive equations.

$$S_b(e,p,i,j)=\min\{S(e',p,i,u)+S(e'',p,u+1,j)\}$$

where the minimum is taken over $1\le i<j\le n$, $i\le u<j$, and $e, e', e''\in\{0,1,2,3\}$.

$S_b$ denotes the solution curves for the sub-solutions that contain direct connections from p to smaller sub-solutions. However, *P_Tree (similar to P_Tree) allows one other possibility where p is connected to another candidate location p′ and then p′ is connected to smaller sub-solutions. In other words:

$$S(e,p,i,j)=\min\{d(p,p')+S(e,p,i,j)\}$$

The minimum is taken over $p'\in P$.

The construction of $S$ and $S_b$ three-dimensional solution curves in a dynamic programming fashion results in generating a final solution curve. The sink order of each solution in that curve is within the neighborhood of the initial sink order.

**Theorem 2:** If the individual capacitive values are polynomially bounded integers or can be mapped to such with sufficient precision, *PTREE has $O(k\alpha^4 q)$ pseudo-polynomial complexity where $k$ is the total number of buffer candidate locations, $\alpha$ is the number of sinks, and $q$ is the maximum number of distinct load values.

### III.3 BUBBLE_CONSTRUCT: The Inner Optimization Engine

The proposed tools and techniques presented in sub-section III.2 are employed in the following algorithm that generates hierarchical buffered routing trees in a neighborhood of orders. The resulting hierarchies are consistent with the C$\alpha$_Tree structure and the routing inside each layer of hierarchy is a *P_Tree. In the following paragraphs the details of this algorithm, called BUBBLE_CONSTRUCT, are given.

BUBBLE_CONSTRUCT (see Figure 9) is called by MERLIN (see Figure 14) along with a set of parameters, *s, P, B,* and $\Pi$. The parameters $s$ and $\Pi=(s_1,s_2,\ldots,s_n)$ represent the root and an ordered set of sinks of a subject net. The parameter $P=\{p_1, p_2, \ldots, p_k\}$ represents a set of candidate locations for the placement of buffers and Steiner points in the final buffered routing structure. Finally, $B=\{b_1, b_2, \ldots, b_m\}$ is a library of buffers.

BUBBLE_CONSTRUCT operates on three dimensional solution curves, $\Gamma$, each associated with a candidate buffer location $p$ and a sub-problem identified by the variables *l, e,* and *r* (to be described below). At each step of this method, the already generated solution curves are combined and manipulated in order to generate solution curves for new sub-problems. This step is repeated until the solution curve for the main problem is found. From among the solutions of the final $\Gamma$, the solution with the best trade-off between required-time and total buffer area is chosen. At the end, the corresponding structure is generated by tracing back the pointers of the constituting sub-problems. The detailed description of the algorithm is given below.

```
algorithm BUBBLE_CONSTRUCT( s , P , B , Π ) {
    INITIALIZATION
1.  for e′ = 0 to 3
2.    for r′ = n to 1
3.      foreach p′∈P
4.        set Γ( 1 , e′ , r′ , p′ ) = {  The set of all non-inferior paths
          extended from p′ to s_r′ , driven with or without a buffer };
    CONSTRUCTION
5.  for L = 1 to n {
6.    for E = 0 to 3 {
7.      set L′ = L + STRETCH(E); // see Figure 10
8.      for R = n to L′ {
9.        set G = SINK_SUBSET( Π , R , L′ , E ); // see Figure 13
10.       for l = max( 1 , L-α+1 ) to L-1
11.         for e = 0 to 3 {
12.           set l′ = l + STRETCH(e); // see Figure 10
13.           for r = R to R-l′+1 {
14.             set g = SINK_SUBSET( Π , r , l′ , e ); // see Figure 13
15.             if g-G≠ϕ then continue;
16.             foreach p ∈ P
17.               foreach γ ∈ Γ( l , e , r , p )
18.                 *PTREE( γ ,G-g , Γ( L, E , R , . ) , P , B );
            } }
19.       foreach p∈ P
20.         prune Γ( L, E , R , p );
    } } }
    EXTRACTION
21. find the solution, ρ, in Γ( n , 0 , 1 , s ) which best satisfies the
    constraints;
22. retrieve the buffered routing tree structure, ℜ, of ρ by
    following the pointers stored during the generation of the
    solution curves;
23. return ℜ;
}
```

**Figure 9: BUBBLE_CONSTRUCT**

Before performing any operation, a set of solution curves are initialized in lines 1 through 4. In this part of the algorithm, sub-groups of length 1 are considered and the corresponding solution curves for every candidate buffer location, sink, and grouping structure are initialized. These initial solutions consist of the minimum Manhattan distance paths from the candidate location $p'$ to the target sink $s_{r'}$. At the root of these paths, both options of inserting or not inserting a buffer are examined. Note that for sub-groups with length 1, all four grouping structures ( $\chi_0$, $\chi_1$, $\chi_2$, and $\chi_3$ ) are the same, however for the sake of simplicity in the rest of the pseudo-code we generate separate (although similar) solution curves for each case; a similar situation occurs for $\chi_1$ and $\chi_2$ where $L=2$. In these initialized solution curves, like any other ones in the

rest of this algorithm, only the non-inferior solutions (see Definition 6) are stored.

BUBBLE_CONSTRUCT starts from $L=1$ (goes up to $L=n$) and groups every $L$ neighboring sinks. For each new sub-group of sinks, all possible grouping structures (coded by numbers *0* to *3*) are enumerated in line 6. For the case of $\chi_0$ ($E=0$), 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, to capture the effect of inserting one or two bubbles on the sides. This new length is calculated and stored in $L'$ (refer to line 7 and Figure 10). In line 8, all possible sub-strings of length $L'$ are considered from right to left of $\Pi$. In fact, the variable $R$ points to the right-most element of the sub-strings of $L'$ elements.

```
algorithm STRETCH( E ) {
    switch E {
      case 0: return 0; case 1 , 2: return 1; case 3: return 2;
}   }
```

**Figure 10: STRETCH**

Every sub-group of sinks can potentially constitute an internal node in the final Cα_Tree structure, therefore according to Definition 2, it can contain at most one immediate internal node (smaller sub-group). Consequently, during the process of grouping a set of $L$ sinks, we should consider cases in which a sub-set of them have already been grouped. 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, we do not process all the sinks at once, instead at any time we work on a subset of sinks and combine them together in agreement with the Cα_Tree structure. Later, each combination is treated as one node in the next level of hierarchy.

Lines 10 through 13, similar to lines 5 through 8, investigate all the possible sub-group lengths with different grouping structures and positions for which the solution curves have already been generated and they fit inside the sub-group being constructed. Figure 11 illustrates an example where a sub-group of 5 sinks, $\Omega$ , is being generated using a combination of an already generated sub-group of 3 sinks, $\omega$ , and two other sinks, i.e. $s_2$ and $s_4$.

In line 10, the term **max(*1*,*L*-α-*1*)** ensures that $\Omega$ does not drive more than α other internal and sink nodes, following the third property of Cα_Tree's given in Definition 2. In line 13, the term ***R* to *R*-*l'*+*1*** ensures that $\omega$ remains within $\Omega$ .

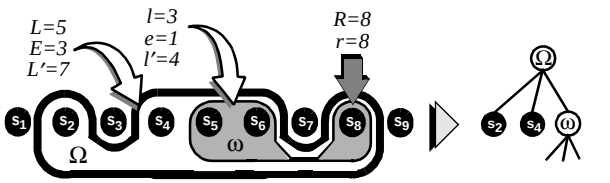


**Figure 11: An Illustration for the Grouping Steps**

It can be seen that in some cases $\Omega$ and $\omega$ are not compatible. As an example, see the situation shown in Figure 12 where the difference between the values of $r$ and $R$ is such that the grouping structure of $\omega$ does not fit in the grouping structure of $\Omega$ . These cases are detected and skipped in line 15 of the pseudo-code. In that line, cases in which a sink node belongs to $\omega$ but not to $\Omega$ are detected and skipped. Note that sets $G$ and $g$ - calculated in lines 9 and 14 - represent the sets of sinks included by $\Omega$ and $\omega$, respectively; also refer to Figure 13.

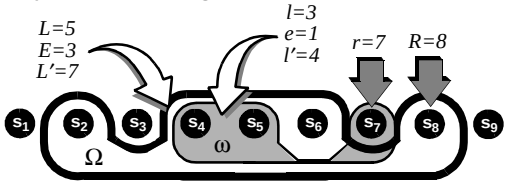


**Figure 12: An Illegal Grouping Case**

After line 15, it is determined which sub-group $\omega$ is to be combined with which sink node(s) to generate the new sub-group, $\Omega$. Also, it is guaranteed that $\omega$ is compatible with $\Omega$. However as mentioned earlier, there are many solutions associated with each $\omega$; in fact for every buffer candidate location, there is a solution curve for $\omega$ which has to be considered in the merge operation. Line 16 enumerates all the candidate locations by variable $p$, and line 17 retrieves the non-inferior solutions in the solution curve of $p$ that corresponds to $\omega$.

```
algorithm SINK_SET( Π , R , L' , E ) {
1.  switch E {
2.    case 0 : set G = { s_{R-L'+1} , s_{R-L'+2} , s_{R-L'+3} , ... , s_{R-2} , s_{R-1} , s_R } ;
3.    case 1 : set G = { s_{R-L'+1} , s_{R-L'+2} , s_{R-L'+3} , ... , s_{R-2} , s_R } ;
4.    case 2 : set G = { s_{R-L'+1} , s_{R-L'+3} , ... , s_{R-2} , s_{R-1} , s_R } ;
5.    case 3 : set G = { s_{R-L'+1} , s_{R-L'+3} , ... , s_{R-2} , s_R } ;
    }
6.  return G;
}
```

**Figure 13: SINK_SET**

In line 18, *PTREE is called to generate a new set of solutions for all the candidate locations (i.e. members of $P$). Every solution created by *PTREE shows the combination of $\omega$ with the rest of sink nodes of $\Omega$ . They are combined by a buffered routing structure rooted at a candidate location. For every routing structure generated by *PTREE, all the buffers in the library are tried to drive that structure and the solutions are stored in the corresponding solution curves. Along with every solution, a set of pointers are stored that later during the extraction phase are used to reconstruct the best solution. Pruning operation (based on Definition 6) is performed in lines 19 and 20.

This process continues until the solution curve for the whole problem, i.e. $L=n$, is generated. From among all the final non-inferior solutions, the one which best satisfies the input constraint is chosen. The buffered routing structure corresponding to that solution is retrieved in lines 21 and 22 by following the stored pointers. Finally, in line 23 the constructed solution is returned to MERLIN. Note that the order of sinks in this final solution may be different from the initial given order, and this new order is used by MERLIN to perform its local neighborhood search, as will be described in sub-section III.4.

In the following statements that formally describe the properties of BUBBLE_CONSTRUCT, it is assumed that in the solution curves the individual capacitive values are polynomially bounded integers or can be mapped to such with sufficient precision. Also, in these statements $q$ is the maximum number of distinct load values.

**Lemma 5:** Any order generated by BUBBLE_CONSTRUCT is in the neighborhood of the initial order $\Pi$.

**Lemma 6:** Any $\Pi' \in N(\Pi)$, is considered by BUBBLE_CONSTRUCT.

**Lemma 7:** Any identical sub-problem among the members of $N(\Pi)$ is shared and processed only once.

**Theorem 3:** The solution space of BUBBLE_CONSTRUCT is the Cartesian product of the space of *P_Tree and the space of Cα_Tree for the neighborhood of the initial given order.

**Lemma 8:** BUBBLE_CONSTRUCT is monotone with respect to required time, load, and buffer size.

**Lemma 9:** In BUBBLE_CONSTRUCT, the use of the pruning operation does not result in the loss of any non-inferior solution.

**Theorem 4:** Subject to restriction imposed by the *P_Tree and Cα_Tree structures, BUBBLE_CONSTRUCT finds all the non-inferior solutions with respect to required time and total buffer area in the neighborhood of a given order.

**Lemma 10:** The number of non-inferior solutions in any solution curve is bounded by $O(nmq)$, where $n$ and $m$ are the number of sinks and the number of library buffers, respectively.

**Theorem 5:** BUBBLE_CONSTRUCT has $O(n^3mkq)$ pseudo-polynomial memory complexity, where $n$, $m$, and $k$ are the numbers of sinks, library buffers, and candidate locations, respectively.

**Theorem 6:** BUBBLE_CONSTRUCT has $O(n^4\alpha^5q^2k^2m)$ pseudo-

polynomial runtime complexity, where $n$ is the number of sinks, $\alpha$ is the maximum immediate fanout for buffers, $k$ is the number of candidate locations, and $m$ is the number of library buffers.

**Corollory 1:** Considering that $m$ is a constant determined by the number of library buffers, and assuming $\alpha$ is a number determined by the library and can be thus considered constant, the effective complexity of BUBBLE_CONSTRUCT for a fixed library is $O(n^4q^2k^2)$.

### III.4 MERLIN: The Outer Search Engine

The behavior and the structure of BUBBLE_CONSTRUCT makes it an appropriate tool for performing local neighborhood search in the space of sink orders. In this sub-section, our local search algorithm, *MERLIN*, is presented.

**Lemma 11**: The properties required by Definition 1 are consistent with the properties of neighborhood in Definition 4.

There exists at least two sink orders , i.e. $\Pi$ and $\Pi'$, in common between the neighborhood of two consecutive iterations of MERLIN's local search. In fact, often this overlap, $OVERLAP(N(\Pi),N(\Pi'))$, is relatively large. Obviously, the overlapping sub-space is considered twice which is clearly wasteful. However, this can be prevented by keeping the solution curves of the very last iteration. For similar sub-problems, between the two iterations simply copy the corresponding solution curve. Obviously, this speed up is achieved at the cost of doubling the memory usage.

**Theorem 7:** The best cost associated with order $S'$ (see line 7) strictly decreases during the operation of MERLIN, except in the last time that the loop (lines 4 thorough 8) is visited.

**algorithm** *MERLIN* {
1. **read** *s*, *P*, and *B* where,
   *s* is the source,
   $P=\{ p_1 , p_2 , \dots , p_k \}$ is a set of candidate locations for buffers,
   $B = \{ b_1 , b_2 , \dots , b_m \}$ is the library of buffers;
2. **read** $\Pi = ( s_1 , s_2 , \dots , s_n )$ , an ordered list of sinks;
3. **set** $\Pi' = \Pi$;
4. **do** {
5. **set** $\Pi = \Pi'$;
6. **set** $\Re$ = *BUBBLE_CONSTRUCT( s , P , B ,* $\Pi$ *)*;
7. **set** $\Pi'$ = *SINK_ORDER(* $\Re$ *)*;
8. } **while** ( $\Pi$ != $\Pi'$ );
9. **return** $\Re$;
}

**Figure 14: MERLIN**

## IV. Experimental Results

In this section, we report two sets of experimental results to verify the effectiveness of MERLIN. In the first table the results have been presented for a set of individual nets taken from a number of benchmarks. The second table reports post-layout areas and delays of a set of benchmark circuits when MERLIN and the conventional techniques for routing and fanout tree generation are employed.

Table 1 reports the area and delay for 18 randomly selected nets from a set of mapped benchmark circuits; therefore the load and required time of each sink node are known. For every extracted net, the locations of sinks are determined randomly and a priori in a bounding box which is sized such that the delay of interconnect is approximately equal to the delay of gate.

As shown in the table, for every net three different experimental setups have been considered. For each case the total delay, buffer area, and runtime have been reported. The three experimental setups are as follow:

- Setup I: Fanout optimization using LTTREE is followed by PTREE. The sink order for the LTTREE phase is based on the required times of sinks and for the PTREE phase is based on the solutions to the TSP (Traveling Salesman Problem), as suggested by [LCLH96].
- Setup II: Routing tree generation using PTREE is followed by buffer insertion using the method of [Gi90]. The sink order for PTREE is again the TSP order.
- Setup III: MERLIN is used for hierarchical buffered routing generation. The initial order is the TSP order although our experimental results show that initial orders have very small effect on the final quality of results. The last columns reports the number of loops that MERLIN takes for convergence to a local minimum. In this setup, the candidate locations for buffer placement are the complete Hanan points and $\alpha$ equals 15.

Table 2 reports the post-layout (after detailed routing) area and delay of a set of benchmark circuits to evaluate the overall effect of the existing buffered routing generation algorithms over a full design flow. The runtimes are the total runtimes (from mapping to detailed routing). For each circuit every one of the aforementioned experimental setups have been used to generate the buffered routing structures for every net. For the MERLIN setup, the number of iterations for each net is bounded by 3, the candidate locations are the reduced Hanan points (generated by a simple heuristic), and $\alpha$ equals 10.

All of the above experimental setups have been implemented in the SIS [SSLM92] environment and have been run on Sun Sparc workstations. In these experiments, we have used an industrial standard cell library (0.35u CMOS process) that contains 34 buffers. Gate and wire delays are calculated using a 4-parameter delay equation [LSP98] and the Elmore delay [El48], respectively.

## V. Conclusions

In this paper, the problem of distributing a signal among a set of sinks with different placements, loads, and required times has been addressed. The proposed algorithm generates a set of non-inferior buffered routing structures which provides different trade-offs between the total required-time and the buffer area. The introduced solution consists of an iterative optimization block which uses a local neighborhood search strategy and an optimization engine based on dynamic programming which 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 exponential size solution space in a polynomial time. The experimental results show a major delay improvement with little area penalty compared to the conventional buffer and routing tree generation techniques.

## VI. References


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

[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 the 30th 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.

[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*, W. H. Freeman, SF, CA, 1979.

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

[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 33th Design Automation Conference*, pp. 395-400, 1996.

[LSP98] J. Lou, A. H. Salek, and M. Pedram, "An integrated flow for technology remapping and placement of sub-half-micron circuits," In *Proceedings of Asia and South Pacific Design Automation Conference*, pp. 295-300, 1998.

[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 *Proceed-*

*ings of the 5'th ACM/SIGDA physical Design Workshop*, pp. 1-6, 1996.

[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.

[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*, November 1990.

[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, January 1989.

[Ya92] M. Yannakakis, "The Analysis of Local Search Problems and Their Heuristics," In *Proceedings of 7'th Annual Symposium on Theoretical Aspects of Computer Science*, pp. 298-311, 1990.

| Taken from circuit | Net name | Num of sinks | Flow I: LTTREE + PTREE | | | Ratios over Flow I | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | Flow II: PTREE+Buffer Insertion | | | Flow III: MERLIN | | | |
| | | | Area X1000$\lambda^2$ | 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 Set of Nets**

| Circuits | Flow I: LTTREE + PTREE | | | Ratios over Flow I | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | Flow II: PTREE+Buffer Insertion | | | Flow III: MERLIN | | |
| | Area X1000$\lambda^2$ | 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 Benchmarks**