4

SIMULTANEOUS ROUTING AND BUFFER INSERTION ALGORITHM FOR MINIMIZING INTERCONNECT DELAY IN VLSI LAYOUT DESIGN

Nasir Shaikh-Husin
Mohamed Khalil-Hani

4.1 INTRODUCTION

In deep submicron fabrication technology, transistors can now switch much faster, but wire resistances are now larger, and delay due to wires can exceed gate delay. Consequently, the interconnect delay is the dominant factor in the construction of wire routing in very large scale integrated (VLSI) circuits, which today, has feature dimensions in the nanometer range. Today, the state-of-the-art circuit design involves as much the engineering of the wires as the design of transistors. Hence, a successful VLSI design today depends heavily on a successful interconnect design.

An effective approach for reducing the interconnect delay is buffer insertion (van Ginneken, 1990). In this method, a wire is divided into segments with a buffer inserted between the segments (Cong et al., 1996). Traditionally, buffer insertion is a post-layout optimization technique, implying that the routing paths are first found, and then buffers are inserted in these paths. However, today's VLSI designs typically apply some form of design reuse utilizing pre-designed cells, or macro blocks. Clearly, buffers cannot be inserted into areas in the VLSI layout occupied by these

macro blocks (here on, referred to as "buffer obstacle" areas). However, it is possible to route a wire over these buffer obstacles (e.g. through a higher level interconnect). Nevertheless, a wire routed for long stretches over the buffer obstacle areas can result in a huge wire delay. Hence, the buffer insertions must take into account these buffer obstacle areas. It is clear that the two-stage "routing then insert buffer" approach cannot give an optimal solution; that is, with the macro blocks, a shortest path cannot guarantee minimum delay. Furthermore, since a wire adds loading to a buffer, proper wire sizing can result in minimum delay. Proper wire sizing can reduce the interconnect delay by a further 10% (Lai and Wong, 2002). For nets that have macro blocks where wires can pass through but not buffers, we can find the optimal interconnect path if routing, buffer insertion, and wire sizing are considered simultaneously. The problem at hand is essentially an interconnect optimization problem, and is conceptually summarized as follows. Given a source and sink node of a net in a VLSI layout, we wish to find a buffered routing path that connects these two nodes such that the end-to-end delay is a minimum.

This paper proposes an efficient algorithm to solve the above interconnect optimization problem. The key goal of the algorithm is to construct a maze routing path, simultaneously with buffer insertion and wire sizing in the presence of wire and buffer obstacles, such that the Elmore delay from source to sink is minimized. The problem is formulated as a shortest-path problem in a weighted graph. The proposed algorithm, named S-RABILA (*S*imultaneous *R*outing *a*nd *B*uffer *I*nsertion with *L*ook-*A*head), utilizes three key concepts in order to find a solution efficiently. They are k-shortest path search, dominant path, and a novel look-ahead feature. This look-ahead scheme, which significantly speeds up execution time of the algorithm, is a key contribution of this work. The result is that the proposed algorithm is accurate, fast, scalable with problem size, and can handle large routing graphs. The paper is organized as follows. In Section 1.2, previous work

that are related to this work are presented, followed by, in Section 1.3, a discussion on the buffered path search problem, including Elmore delay formulations as applied in this work. Section 1.4 describes the proposed algorithm, S-RABILA. The experimental result of the performance tests conducted on S-RABILA is provided in Section 1.5. Finally, we present the conclusion in Section 1.6.

4.2 PREVIOUS WORKS

Many algorithms have been proposed to solve the interconnect optimization problem. van Ginneken pioneered the optimum buffering in a routing tree structure (van Ginneken, 1990). This algorithm, which applies dynamic programming, can only insert buffers after the routing tree is known. An existing algorithm called Fast Buffer Insertion (FBI), proposed by Li and Shi (Li and Shi, 2006), inserts buffers optimally for a given routing tree. This algorithm can accommodate multiple buffers, although it cannot handle multiple wires. However, these algorithms are post-routing buffer insertion techniques. Lillis, Cheng, and Lin have enhanced the van Ginneken algorithm to obtain even better delay optimality by solving the routing problem with simultaneous buffer insertion and wire sizing (Lillis, Cheng, and Lin, 1996).

More recently, much improved algorithms for simultaneous routing and buffer insertion have been proposed, that are more suitable for today's VLSI technology. Zhou et al. proposed such an algorithm, in which the routing path is obtained with simultaneous buffer insertion, and in addition, took into account wire and buffer obstacles (Zhou et al., 2000). Since the search technique applied is based on van Ginneken dynamic programming approach, we shall refer to Zhao's algorithm as the DP algorithm in this paper. This algorithm was further enhanced to include wire sizing by Lai and Wong (Lai and Wong, 2002). They call their algorithm SP, for Shortest Path algorithm. SP is a graph-based algorithm. It first converts the input maze graph into a bigger graph, which is called

buffer planning (BP) graph. Dijkstra's algorithm is then applied on the BP graph to find the optimal delay path, which is determined to be the shortest path between the source and sink vertices in the BP graph. The algorithms mentioned above utilized Elmore delay models. In (Huang et al., 2003), Huang et al. extended the work by Lai and Wong (Lai and Wong, 2002) to include wire inductance using transmission line model for the delay computation. However, only fixed wire sizes are considered, and the solution is restricted to small problems. It should be noted here, that the DP and SP algorithms mentioned above are used in benchmarking the performance of S-RABILA algorithm proposed in this paper.

A serious limitation of these existing algorithms is that their execution times are not adequately fast for application in today's deep-submicron VLSI layout design, particularly when applied to large problems. They do not use look-ahead schemes, or the like, which can speed up the algorithm runtimes, and furthermore, they cannot provide exact solutions. In this aspect, Van Mieghem and Kuipers have proposed an algorithm called SAMCRA that uses a novel look-ahead concept in finding an exact solution in multi-weighted graph problems applied in Quality-of-Service (QoS) routing (Van Mieghem and Kuipers, 2004). This look-ahead concept is adapted in this work for application in VLSI routing problems.

4.3 BUFFERRED PATH SEARCH PROBLEM

The maze routing with simultaneous buffer insertion and wire sizing problem in VLSI layout design is essentially a buffered routing path search problem. In this work, it is formulated as a shortest-path problem in a weighted graph, and is specified as follows. Given a routing grid graph $G = (V, E)$ corresponding to a VLSI layout, a buffer library B, a buffer function p with $p(v) = 1$ indicating buffer insertion is allowed at vertex v, a wire library W, two vertices s, $sink \in V$, find a buffered path $P = (v_1, v_2, ..., v_n)$, with $v_1 = s, v_n = sink, b(v_i) \in B \cup \{-1\}$ where $b(v_i) = -1$ indicates

that no buffer is inserted at v_i, and $w(v_j, v_{j+1}) \in W$, such that the interconnect delay for path P is minimized. $V = \{s\} \cup \{sink\} \cup V_n$ and E is the set of edges. Vertex s is the source vertex, $sink$ is the sink vertex, and V_n is the set of internal vertices. A vertex $v \in V_n$ may belong to the set of buffer obstacle vertices (vertices in areas where buffers are not allowed), denoted V_{OB}, or to the set of wire obstacle vertices (vertices in areas where wires are not allowed), denoted as V_{OW}. A buffer library B contains different types of buffer and a wire library W contains wires with different widths. A buffer in library B is denoted B_i and a wire in wire library W is denoted W_i. For each edge $e = u \rightarrow v$, signal travels from u to v, where u is the upstream vertex while v is the downstream vertex and $u, v \notin V_{OW}$.

Consider a sample problem shown in Figure 4.1, where the grid graph, buffer obstacles (grey areas) and wire obstacles (dark areas) are given. The model parameters used are as follows: source resistance is 140 Ω, load capacitance at the sink is 0.002 pF, and resistance and capacitance of a wire segment are 58 Ω and 0.042 pF respectively. We assume there is only one type of buffer with input capacitance, output resistance and intrinsic delay of 0.002 pF, 140 Ω and 40 ps respectively. One approach of the path search is simply to find the shortest distance between the source and sink points without going into obstacle areas. Once the path is determined, buffers are placed along this path so that the delay is minimized. The result on our sample problem is depicted in Figure 4.1(a), where the total path delay is found to be 680.6 ps. Another method is to ignore only wire obstacles (i.e. buffer obstacles are taken into account) and find the shortest path between the end points. Once this path is found, buffers are placed at the allowable positions. The resulting path is shown in Figure 4.1(b), and the total path delay is found to be 621.8 ps, which is an improvement over the previous case. Both these cases apply post-routing buffer insertion techniques.

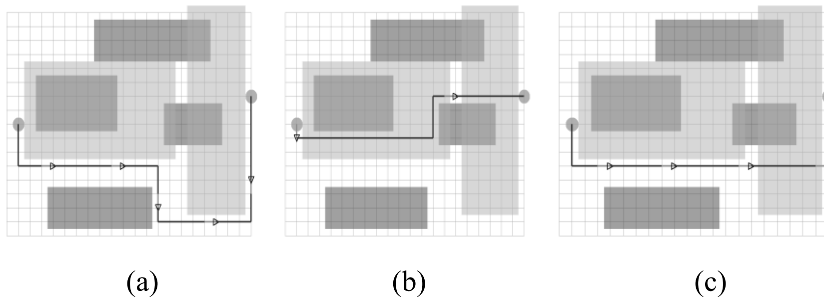

Figure 4.1 A sample maze routing problem (the dark areas are wire obstacles while the grey areas are buffer obstacles.) (a) Routing path avoids all obstacles. (b) Routing path with buffer obstacles considered. (c) Optimal routing.

The best result is obtained by applying both path search and buffer insertion simultaneously. The optimum solution where buffer restrictions are considered when searching for the best path is shown in Figure 4.1(c). The total path delay in this case is found to be 521.7 ps, which is a significant improvement. This clearly indicates that simultaneous routing and buffer insertion is the approach to take in order to find the optimal delay routing. The problem however becomes more challenging if we have several buffer types to choose from. Each buffer differs in terms of input capacitance, output resistance and intrinsic delay. The problem complexity further increases when we consider several wire sizes (i.e. different resistance and capacitance characteristics) to connect any buffer-to-buffer section. In these latter cases, where buffer types and wire sizes are considered, the runtime of existing algorithms are unacceptable, especially in large problems.

4.3.1 Interconnect Delay Model Formulations

Recalling the problem at hand, we wish to find a buffered path from a source to a sink node such that the interconnect delay is minimized. For the target technology (i.e. deep sub-micron VLSI), where wire resistance can no longer be ignored, an interconnect is typically modeled as an RC distributed network. A wire segment in

the network is represented by a π-model RC circuit as shown in Figure 4.2(a), and a buffer is modeled by the circuit shown in Figure 4.2(b). The labels r_w and c_w are resistance and capacitance of the wire segment, and c_b, r_b, and d_b are input capacitance, output resistance, and intrinsic delay of the buffer.

In this work, Elmore delay is used to model the interconnects. Elmore delay has excellent correlation to the actual delay, and in fact, it is the upper bound for the actual delay (Rabaey, Chandrakasan, and Nikolić, 2003). This work applies an important property of Elmore delay that allows its computation to be performed incrementally and iteratively. The technique is based on the equations formulated by van Ginneken (van Ginneken, 1990), which we now summarized. At the outset, note that we will have different Elmore delay equations for the case when the path expansion begins from the source node and progresses to the sink, and for the case when the path expansion begins from the sink and ending at the source.

In path expansion scheme beginning at the source node, each node in the wire is labeled with a resistance-delay pair *(r, t)*, where *r* and *t* are the resistance and delay accumulated up to that segment, respectively (Lai and Wong, 2002; Zhou et al., 2000). Given a resistance-delay pair value *(r, t)* at the output of a wire segment, the resistance-delay pair value at the output of the subsequent downstream segment *(r', t')* can be computed as follows. If the downstream segment is a wire, then *(r', t')* is

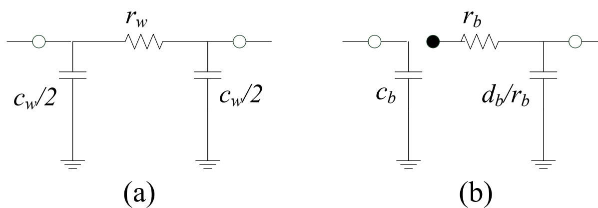

Figure 4.2 (a) Wire segment model. (b) Buffer model.

$$r' = r_w + r \tag{4.1a}$$
$$t' = (r + r_w/2)c_w + t. \tag{4.1b}$$

If the downstream segment is of a wire terminated with a buffer, then *(r', t')* is

$$r' = r_b \tag{4.2a}$$
$$t' = r(c_w + c_b) + r_w(c_w/2 + c_b) + d_b + t. \tag{4.2b}$$

In contrast, in the path expansion scheme beginning at the sink node, instead of a resistance-delay pair, each node v is labeled with a capacitance-delay pair, *(c, t)*, where c represents the total ground capacitance $C_T(v)$ rooted at node v. With an initial (*c, t*) given, a new capacitance-delay pair *(c', t')* for the preceding segment can be determined, as follows. If the preceding upstream segment is a wire only, then *(c', t')* is given by

$$c' = c_w + c \tag{4.3a}$$
$$t' = r_w(c_w/2 + c) + t. \tag{4.3b}$$

If the preceding segment is a wire terminated with a buffer, then new capacitance-delay pair becomes

$$c' = c_w + c_b \tag{4.4a}$$
$$t' = r_w(c_w/2 + c_b) + d_b + r_b c + t. \tag{4.4b}$$

This suggests that we can perform the computations for both path traversals concurrently. We maintain the *(r, t)* pairs at the nodes for the path traversal originating from the source, while (*c, t)* pairs are maintained at the nodes for the path traversal originating from the sink. The end-to-end delay (denoted by *EndDelay*) is obtained, when both *(r, t)* and *(c, t)* pairs are available at a particular node. Hence, at node M, where both *(r, t)* and *(c, t)* pairs are already determined,

$$EndDelay = t_M + t_m + r_M c_M \tag{4.5}$$

where (r_M, t_M) is the resistance-delay pair, and (c_M, t_m) is the capacitance-delay pair values at node *M*.

4.3.2 *k*-Shortest Path Search

The routing problem at hand now becomes a multi-weighted single constraint graph problem. Each routing path is weighted with either resistance-delay pair or capacitance-delay pair, depending on the direction of path expansion. Dijkstra's algorithm cannot be applied to graphs with multi-weight edges, as the algorithm requires the property that the shortest path must consist of shortest sub-paths (Cormen, Leiserson, and Rivest, 1990). In multi-weight graphs the shortest path to any vertex is not unique anymore. Consider the example given in Figure 4.3, which shows a two-weighted graph. There are two shortest paths to reach vertex *d* from *a*. The cost for path $a \rightarrow b \rightarrow d$ is (5, 6) while the cost for path $a \rightarrow c \rightarrow d$ is (3, 7). The first path has a lower second weight while the other path has a lower first weight. We should not discard either path in our search for the final solution.

When dealing with a multi-weight graph, it is necessary to store multiple sub-paths for any vertex. There are multiple valid shortest paths in reaching a vertex; each path has at least one lower weight component compared to other paths stored for that vertex. This

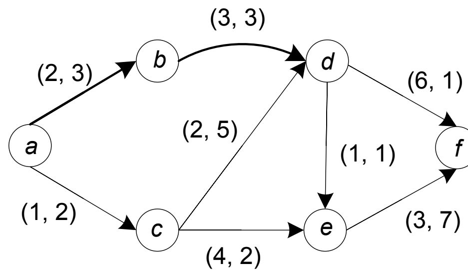

Figure 4.3 A multi-weight graph.

gives rise to so-called "*k*-shortest path" approach. In the problem here, since each intermediate node is characterized by two weights for our Elmore delay calculation, we have to store the shortest delay, second shortest delay, third shortest delay, etc., up to *k*-shortest delay path for each intermediate node. If *k* is not restricted, a *k*-shortest path algorithm returns all possible paths from source to the destination. An algorithm may restrict the limit for *k*, for example in order to reduce memory requirements and runtime. However, this algorithm cannot guarantee end-to-end shortest delay path anymore (Van Mieghem and Kuipers, 2004). This dilemma is compounded by the fact that the parameter *k* is different for each intermediate node. Fixing *k* to a large value may result in unnecessary large memory allocation for most nodes that do not require a large *k*. To guarantee optimal end-to-end delay and efficient memory allocation, S-RABILA determines the parameter *k* adaptively for each intermediate node. The fact that S-RABILA does not restrict the parameter *k*, alluding to computation of all feasible paths between source and destination, gives an impression that the algorithm may exhibit NP-complete traits. However, the number of feasible paths can be reduced significantly by applying the "path-dominance" property.

As mentioned above, in the construction of the routing path, a vertex *u* may have several path candidates (or sub-paths), which correspond to different combinations of wire and buffer sizes for the paths that reach vertex *u*. Each candidate is weighted with a resistance-delay pair (r, t), where *r* is the accumulated resistance at vertex *u* (from the source to *u*) and *t* is the delay up to vertex *u*. A new candidate is added to the list of existing candidates if it is not dominated. Between any two candidates $\alpha_1 = (r_1, t_1)$ and $\alpha_2 = (r_2, t_2)$, α_1 is said to "dominate" α_2 if $t_1 \leq t_2$ and $r_1 \leq r_2$. The "dominated" candidate α_2 is not added or, if already in the list, it is removed. This check for path-dominance property between the candidates stored at a vertex is essentially a state space reduction technique. It contributes significantly to an efficient path search,

and is applied in S-RABILA.

4.4 S-RABILA ALGORITHM

The proposed algorithm, S-RABILA solves the simultaneous maze routing and buffer insertion problem in a VLSI layout design. The goal is to find a buffered path (with wiresizing) between the source and a sink such that the interconnect delay of the routed path is minimized. As in other maze routing, the entire routing area is represented as a 2-dimensional (2D) grid graph, where areas of buffer and wire obstacles are known, and the source and destination vertices are specified. The top-level behavior of S-RABILA algorithm is shown in Figure 4.4, and its MAIN( ) procedure in pseudo-code is provided in Figure 4.5.

The algorithm is made up of four core stages. Stage 1 involves the extraction of the graph parameters and formation of the problem grid graph. Based on these parameters, the grid graph $G(N, E)$ modeling the problem, with N vertices and E edges, is formed

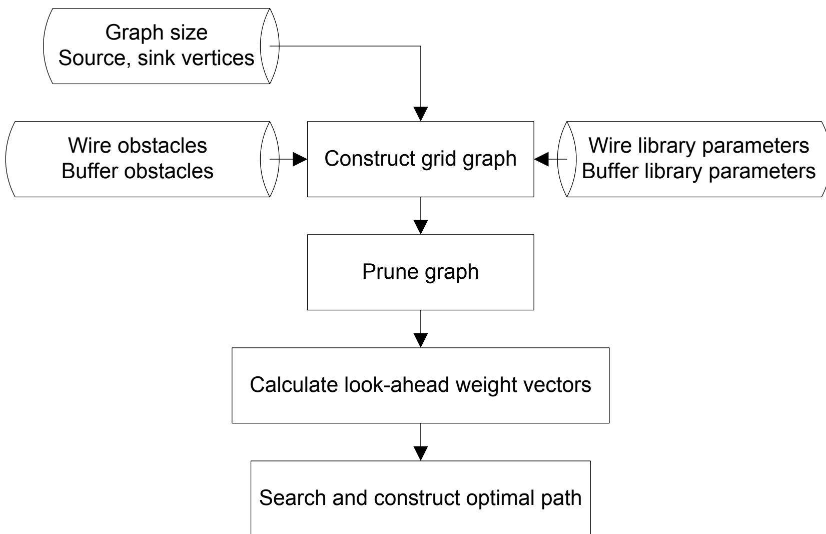

Figure 4.4 Top-level description of S-RABILA algorithm.

```
Function: MAIN()
Function called: READGRAPHINFO(), GRAPHPARAM(), READOBSTACLES(),
                 GRIDTOPOL(ColumnSize, RowSize, V_OW), DIJKSTRA(G, terminal),
                 LA-WEIGHT(L, WireTotal, BufTotal),
                 PATH-TRAVERSAL( G, WireTotal, BufTotal, LACount[ ], WeightLA[ ] )
Output:          EndDelay, Path[ ], WireType[ ], BufType[ ]

    // CONSTRUCT GRID GRAPH
1   ColumnSize, RowSize, s, sink ← READGRAPHINFO()
2   WireTotal, BufTotal, wire and buffer parameters ← GRAPHPARAM()
3   N = ColumnSize x RowSize
4   V_OW, V_OB ← READOBSTACLES()
5   G(V, E) ← GRIDTOPOL(ColumnSize, RowSize, V_OW)
    // FIND SHORTEST PATHS
6   L_ToEND[] ← DIJKSTRA(G, sink)          // wrt to sink
7   L_ToSTART[] ← DIJKSTRA(G, s)           // wrt to source
8   for i = 1, …, N
9       if i in V_OB
10          for each j in Adj[i] e(i, j) is broken end for      // convert V_OB to V_OW
11      end if
12  end for
    // FIND SHORTEST PATH AVOIDING BUFFER OBSTACLES
13  L_StartEnd ← DIJKSTRA(G, s)
    // PRUNE GRAPH
14  for i = 1, …, N
15      if L_ToEND[i] + L_ToSTART[i] > L_StartEnd
16          i in V_OW
17      end if
18  end for
    // COMPUTE LOOK-AHEAD WEIGHT VECTORS
19  L = L_ToEND[s] + 1
20  LACount[], WeightLA[] ← LA-WEIGHT(L, WireTotal, BufTotal)
    // FIND & CONSTRUCT OPTIMAL ROUTING PATH
21  EndDelay, Path[], WireType[], BufType[] ← PATH-TRAVERSAL( G, WireTotal,
                                              BufTotal, LACount[], WeightLA[] )
```

Figure 4.5 MAIN( ) routine of S-RABILA.

through the GRIDTOPOL() subroutine. The second stage performs graph pruning. Stage 3 involves the computation of the look-ahead weight vectors. The look-ahead weight vectors are obtained by executing the function LA-WEIGHT(). Finally, in stage 4 of the algorithm, the required optimal routing path is searched, traced, and constructed, through the execution of the PATH-TRAVERSAL() function. The path search using the PATH-TRAVERSAL() function is illustrated via flowchart diagram in Figure 4.6.

4.4.1 Graph Pruning

The effort in constructing a routing path depends on the search space of the graph. We can reduce the effort by pruning the graph, whereby *redundant* vertices are removed from the graph. In S-RABILA, a vertex is considered *redundant* in the search space if $L_ToEND[u] + L_ToSTART[u] > L_StartEnd$, where $L_ToEND[u]$ is the shortest path length from vertex u to the sink vertex *END* while $L_ToSTART[u]$ is the shortest path length from vertex u to the source vertex *START*. These paths can pass through areas where buffer is not allowed (i.e. V_{OB}). Now we define a reference path length, denoted by *L_StartEnd*, which is the length of the shortest path from *START* to *END* vertices, without passing through V_{OB}. It is assigned an infinite value if no such path exists. It is obvious that no path can pass through V_{OW}, the areas where wiring is not allowed. Since the reference path avoids buffer obstacle areas, buffers may be inserted anywhere along its route, hence ensuring that the end-to-end delay is the smallest possible. Thus any vertex found in a *START* to *END* path with a path length larger than the reference is redundant, because routing through this vertex produces a delay larger than that through the reference path. The above path lengths are computed by executing Dijkstra's algorithm. In the algorithm in Figure 4.5, DIJKSTRA() function is invoked three times to obtain the required data used in graph pruning. The first call gives the distance (topological) of each vertex in *G* to the sink vertex *END*, the second call gives the distances between the source vertex *START* to all vertices in *G*, and the third call gives another type of distance data, that is, distances from *START* to all other vertices, but this distances are through paths that exclude buffer obstacle vertices. From this distance, a reference distance *L_StartEnd* is obtained.

4.4.2 Look-Ahead Scheme

The runtime of S-RABILA is significantly improved by the

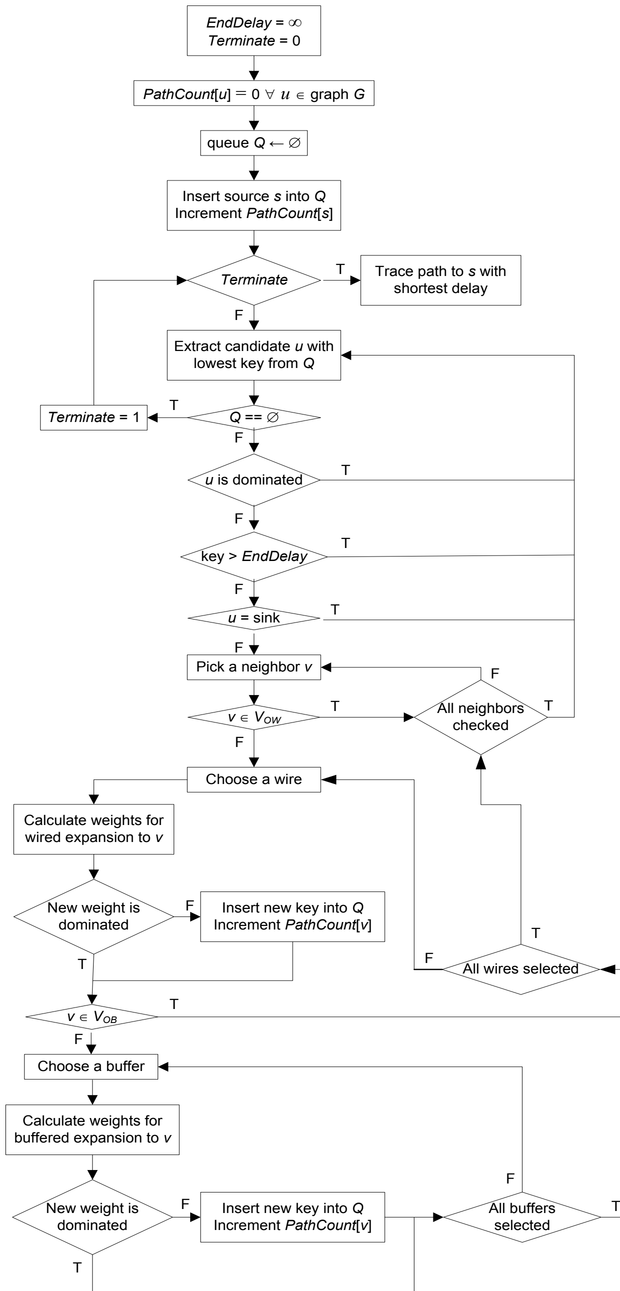

Figure 4.6 Flowchart for PATH-TRAVERSAL meta-function.

application of a novel look-ahead scheme. The idea is to derive for each vertex, look-ahead weights which are non-dominated capacitance-delay pairs. These capacitance-delay pairs can be combined with any resistance-delay pair to obtain the end-to-end delay. During path expansions, which are from source to sink, S-RABILA maintains resistance-delay pairs for all sub-paths. Availability of corresponding capacitance-delay pairs at each vertex permits the computation of an estimate of the end-to-end delay, and hence guides the path expansion. The challenge is how to compute these look-ahead weights (capacitance-delay pairs) for all vertices efficiently. The look-ahead scheme can be explained as follows. S-RABILA first transforms the original 2D graph into a 1D graph, and then computes the capacitance-delay pairs for all vertices in this 1D graph. A scheme is devised to assign look-ahead weights for all vertices in the original 2D graph, using the capacitance-delay pairs of the vertices in the 1D graph.

As an example, consider the sample problem given in Figure 4.7, in which the dark area represents the area where wire is not allowed (V_{OW}), and the grey area represents area where buffers are not allowed (V_{OB}). Vertex-5 is the source and the sink is at vertex-4. We assume, without loss of generality, that only one wire size and one buffer type is available. We first determine the topological distance between the source vertex and the sink vertex, which is the length of the shortest path from source to sink that avoids the wire obstacle areas, but can pass through buffer obstacles. A corresponding 1D grid graph with length equal to the source-to-sink topological distance is created. This new graph has no obstacles whatsoever (neither wire nor buffer obstacles). For the example graph in Figure 4.7, the source-to-sink topological distance is six. Figure 4.8 shows the corresponding 1D grid graph with the grid length of six.

In this 1D grid graph, vertex-7 corresponds to the sink vertex in the

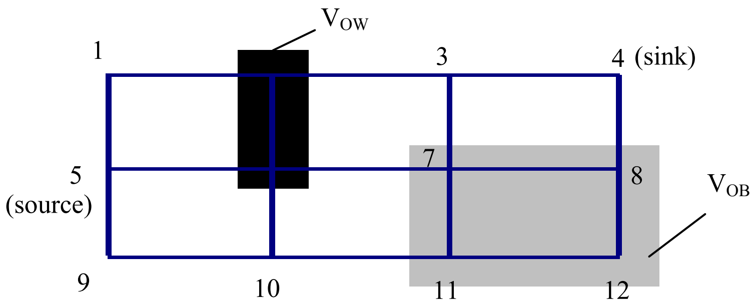

Figure 4.7 Sample problem to illustrate look-ahead concept.

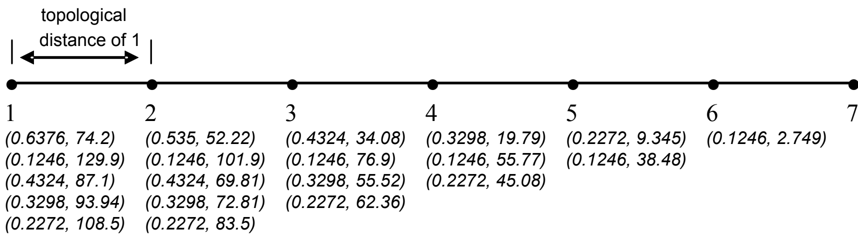

Figure 4.8 1-dimensional graph with look-ahead weight vectors, *(c, t)* (unit for *c* is pF while unit for *t* is ps.)

original 2D graph. Vertex-6 corresponds to all the vertices in the original 2D graph that are one grid distance from the sink, while vertex-5 corresponds to vertices two grids away from the sink, and so on. The delay of the path from each vertex to the sink vertex is computed. Since the delay to be computed is with respect to the sink, the computation scheme using (c, t) pairs is applied. The weight vectors (capacitance-delay pairs) for each vertex are computed using the conventional dynamic programming method, with only the non-dominated vectors stored for each vertex. As shown in the graph of Figure 4.8, the weight vectors are listed under the respective vertex. In this example, we use the following parameter values: load capacitance at sink vertex is 0.022 pF, wire resistance is 37.5 Ω, wire capacitance is 0.1026 pF, buffer input capacitance is 0.022 pF, buffer output resistance is 104.2 Ω, and intrinsic buffer delay is 20 ps. We also assume that the source has

an output resistance of 104.2 Ω.

Note again that the 1D graph has no obstacles (i.e. neither wire nor buffer obstacles). It is clear then, that the weight vectors stored at each vertex in the 1D grid graph provide the absolute lower bound of the delay from a vertex to the sink, since buffer can be inserted anywhere as necessary along the path. For example, consider vertex-1 in the 1D graph (Figure 4.8), which corresponds to all vertices six grids away from the sink in the 2D graph (Figure 4.7). Then, the look-ahead weights at vertex-1 in the 1D graph are non-dominated, absolute lower bound weight vectors from the sink to a vertex six-grid distance away in the original 2D grid graph. Hence these vectors can be viewed as look-ahead weights. The look-ahead weights at vertex u in the 2D graph are denoted by $WeightLA[u]$ in our algorithm, and the number of these weights associated with a vertex u is denoted by $LACount[u]$.

Note that the look-ahead 1D grid graph is based on the source-to-sink topological distance. Consequently, vertices further away from the sink will not have any look-ahead vectors. For example, the distance of vertex-1 to the sink in the original 2D graph is 7. This distance exceeds the topological source-to-sink distance, which is only six. Look-ahead vectors are not calculated for these "far-away" vertices. A special value, *WeightMax*, is assigned as the look-ahead weight for these far-away vertices. *WeightMax* is the minimum end-to-end delay for the 1D graph, after taking into account the source resistance, and is given by: $WeightMax = \min(R_{Source} * c + t, \forall (c, t)$ weights at vertex 1), where R_{Source} is the source resistance. For the example problem in Figure 4.7, where $R_{Source} = 104.2\ \Omega$, the minimum delay is 128.3 ps, which is obtained using capacitance-delay pair (0.3298, 93.94). This constant is now used as the look-ahead delay for all vertices in the original 2D grid graph with topological distance from the sink greater than six. Now, all vertices in the 2D graph will have its own set of look-ahead vectors. Figure 4.9, which is a redraw of

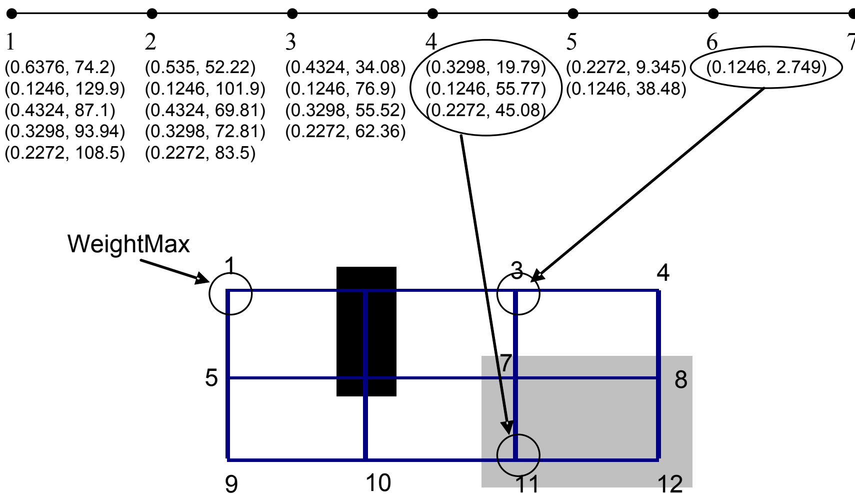

Figure 4.9 Association of look-ahead weight vectors to input grid graph (source is vertex-5 and vertex-4 is the sink).

Figure 4.7, illustrates the association between vertices in the 1D graph and the original 2D graph.

For example, vertex-11 is three grids away from the sink; therefore it is associated with vertex-4 in the corresponding 1D grid graph. Hence, the look-ahead weights of vertex-11, *WeightLA*[*11*] = { (0.3298, 19.79), (0.1246, 55.77), (0.2272, 45.08) }. These three weight vectors represent non-dominated (*c, t*) pairs, one of which is due to path expansion with buffer insertion at vertex-5 in 1D grid graph. When the same look-ahead weight vectors are passed on to vertex-11 in the original 2D graph, similarly, one of the weight vectors assigned to vertex-11 is associated with buffer insertion at vertex-7. Since vertex-7 actually cannot have buffer insertion, one of the look-ahead weight vectors for vertex-11 is not valid. Since it is too troublesome to tract invalid weight vectors, all look-ahead weight vectors (including any non-valid weight vectors) are assigned to the corresponding 2D vertices. The algorithm will handle the non-valid weight vectors such that optimal delay can

still be computed.

The above discussion suggests that we can utilize the look-ahead weights as an estimate of actual (*c, t*) pairs for the vertices in the 2D graph. Consequently, when we have computed the (*r*, *t*) value for a candidate, and knowing the (*c*, *t*) pairs for the candidate's vertex, we can then calculate the (predicted) end-to-end delay, *EndDelay*, by using Equation 4.5. If *EndDelay* is greater than the actual known minimum source-to-sink delay, then this candidate is considered dominated, and therefore is removed. In this way, the number of candidates at the vertices can be substantially reduced, thus speeding up the process of constructing the routing path significantly.

4.5 EXPERIMENTAL RESULTS

Three benchmarking experiments are conducted to measure the performance of S-RABILA. The tests are conducted on a PC with a 3 GHz Pentium D processor and 504 MB RAM. In the first experiment, we benchmark S-RABILA against DP algorithm (Zhou et al., 2000), which has been introduced in Section 1.2. The performance metrics used in the comparison of the two algorithms are algorithm execution runtime (*T*), and the total number of sub-path candidates that are created *(N*). The parameters used here represent typical interconnect wires used in a 65 nm fabrication process. We select six wires with wire capacitance in the range 16.7 – 36.7 fF, and 1.4 – 44.9 Ω for wire resistance. Similarly, six buffers are chosen. The buffer parameters are as follows: 20 fF < c_b < 32.2 fF, 118.1 Ω < r_b < 200 Ω, 10 ps < d_b < 75.9 ps. Two randomly generated test graphs are used in this performance test: Test graph 1 and Test graph 2. Test graph 1 is a small-size graph with a 22x17 grid, 44.4% buffer obstacle area and 26.7% wire obstacle area. Test graph 2 is a medium-size graph with an 80x40 grid, 26% buffer obstacle area and 5.3% wire obstacle area. The algorithm is run for six cases, which are differentiated by the size of wire/buffer library (we set the wire library size = buffer library

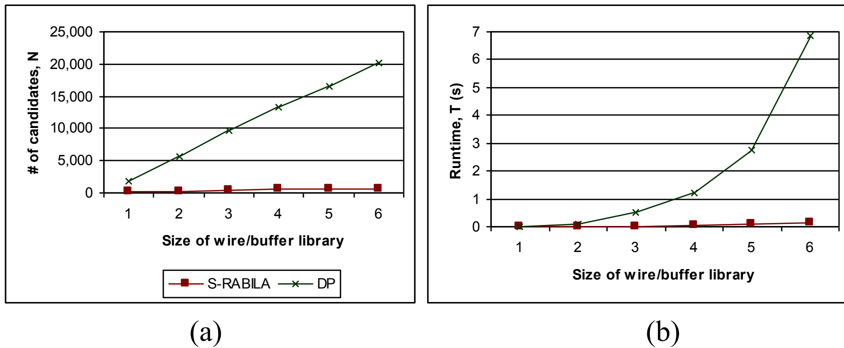

Figure 4.10 Small-size Graph 1. (a) Plot of *N* versus library size. (b) Plot of runtime (*T*) versus library size.

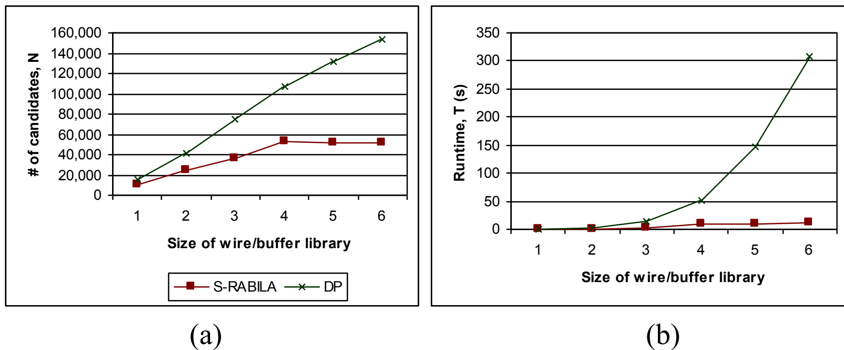

Figure 4.11 Medium-size Graph 2. (a) Plot of *N* versus library size. (b) Plot of runtime (*T*) versus library size.

size). The results are shown in Figure 4.10(a) and (b) for Test graph 1. Figure 4.11(a) and (b) show the results of executing the algorithms on Test graph 2. The results show that S-RABILA executes significantly faster than DP. The runtime of S-RABILA is linear to the library size, whereas it is exponential for DP. This clearly indicates that the look-ahead scheme in S-RABILA contributes significantly to the computational effectiveness of the algorithm, as S-RABILA is similar in execution as DP when it is without the look-ahead feature.

In the second experiment, we benchmark S-RABILA against the

SP (Shortest Path algorithm) by (Lai and Wong, 2002), which has been described in Section 1.2. In (Lai and Wong, 2002), the authors perform a runtime comparison between their SP algorithm and the DP algorithm. They used eight different graph sizes in that comparison, with three buffers and five different wires. The value of wire capacitances that were chosen was in the range 22.3 – 102.6 fF, and 6.9 – 37.5 Ω for wire resistance. The buffer parameters are as follows: 22 fF < c_b < 158.4 fF, 104.2 Ω < r_b < 1064.1 Ω, 20 ps < d_b < 40 ps. The source resistance and load capacitance was 104.2 Ω and 158.4 fF respectively. Lai and Wong did provide the delay that they obtained for each graph. Based on all these information, a graph that should be more or less equal to what they had can be approximated. We created eight graphs that are similar to the graphs used by Lai and Wong to be used as the test graphs in this performance test on algorithm runtime *T*. The wire and buffer parameters that are used in the tests are provided in Tables 4.1. and 4.2.

The results are summarized in Table 4.3. It can be concluded that S-RABILA is faster than SP algorithm. For example, in the case of the 22x32 graph, we can deduce that S-RABILA is 21.5 (66.7/3.1 = 21.5) times faster than SP. The authors exclude the overhead time for determining the weights for the edges in their BP graph in the SP runtime results. In other words, the SP runtime measures only

Table 4.1 Wire library from (Lai and Wong, 2002)

Name	Length (μm)	r_w (Ω)	c_w (fF)
Wire1	500	37.5	22.2
Wire2	500	30	42
Wire3	500	22	62
Wire4	500	15	83
Wire5	500	6.9	102.6

Table 4.2 Buffer library from (Lai and Wong, 2002)

Name	Intrinsic delay d_b (ps)	Output resistance r_b (Ω)	Input capacitance c_b (fF)
Buf1	40	1064.1	22
Buf2	30	584	90
Buf3	20	104.2	158.4

Table 4.3 Runtime comparison between DP, SP, and S-RABILA algorithms

Graph size	Runtime[#] (s)		Runtime[*] (s)		Runtime improvement over DP	
	DP	SP	DP	S-RABILA	SP	S-RABILA
20x24	148.1	5.4	11.17	0.2637	27.4 x	42.4 x
28x22	197.1	93.2	23.01	2.148	2.1 x	10.7 x
20x30	231.2	20.4	29.45	0.7203	11.3 x	40.9 x
22x32	303.5	97	72.55	1.088	3.1 x	66.7 x
28x28	258.2	63.8	28.21	0.8903	4.0 x	31.7 x
28x24	269.5	25.3	51.04	0.6726	10.7 x	75.9 x
24x28	213.6	11.1	18.88	0.522	19.2 x	36.2 x
24x20	124.4	12.6	11.37	0.5777	9.9 x	19.7 x

[#] Run on Sun Ultra Sparc 5 w/station
[*] Run on 3 GHz Pentium D PC, 504 MB RAM

the time taken for their Dijkstra's algorithm to solve the new BP graph after edge costs had been already assigned. The performance of SP algorithm probably is slightly worse if the overhead time is taken into account.

In the third experiment, S-RABILA is tested on a large graph, Test

graph 3, shown in Figure 4.12. Its characteristics are summarized in Table 4.4. The same wire and buffer parameters as in the first experiment are used. The algorithm is run for six cases, which are differentiated by the size of buffer library. In all six cases, only *Wire1* is used. The results of executing S-RABILA on Test graph 3 are shown in Figure 4.13(a) and (b). The runtime of S-RABILA in this test is again not exponential to the buffer library size. It seems to be linear for the first three cases, after which the runtime performance even improves slightly. Unfortunately, however, this linearity apparently is not applicable anymore when wiresizing is to be performed. When the wire library is increased, the runtime is

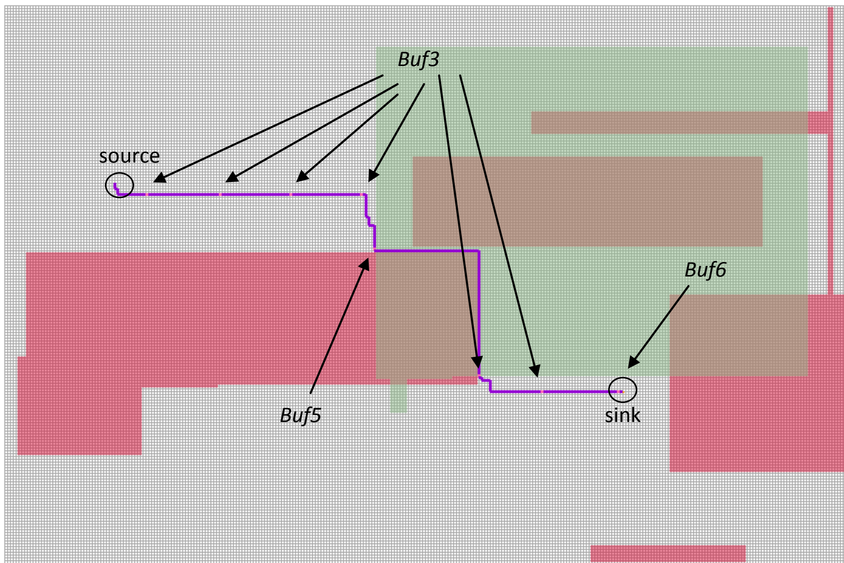

Figure 4.12 Large-size graph (Test graph 3).

Table 4.4 Characteristics of Test graph 3 (large graph)

Size	*Equivalent layout area*	*Buffer obstacle*	*Wire obstacle*	*Effective wire obstacle*
300x200	30x20 mm^2	30%	30%	59.2%

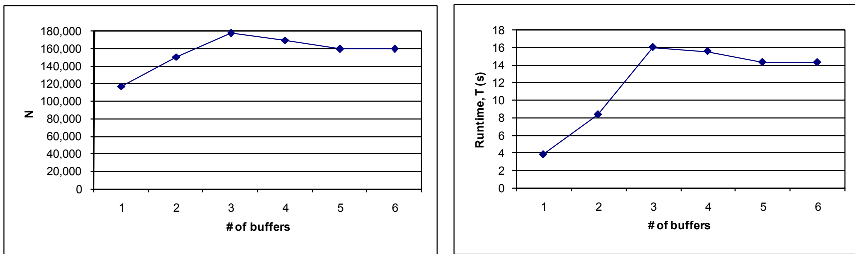

Figure 4.13 Large-size Graph 3. (a) Plot of N versus library size. (b) Plot of runtime (T) versus library size.

simply too long. In conclusion, we can assert that for most practical cases, the runtime for S-RABILA is linear to problem size.

4.6 CONCLUSION

This paper describes a new algorithm for optimal interconnect delay for two-terminal nets in VLSI layout routing design. This algorithm, named S-RABILA, inserts buffers while considering wire and buffer obstacles and simultaneously constructs interconnect routing with wire sizing. The search mechanism in the algorithm is based on the van Ginneken dynamic programming approach. The key contribution of the work is a novel look-ahead scheme, which has been shown to improve significantly the computational speed of the search in S-RABILA. In most of the performance tests, the runtime of S-RABILA is an order of magnitude faster than DP. Also, when compared to SP routing algorithm, it is at least 1.5 times faster. This speed improvement clearly indicates the computational effectiveness of the look-ahead scheme. Other tests performed have also shown that S-RABILA can handle large graphs with acceptable runtimes. Thus, it has been shown that S-RABILA is accurate, fast, scalable with problem size, and can handle large routing graphs.

REFERENCES

Cong, J. et al. (1996). Performance optimization of VLSI interconnect layout. *INTEGRATION*, *the VLSI Journal*. 21:1-94.

Cormen, T. H., C. E. Leiserson and R. L. Rivest. (1990). *Introduction to Algorithms*, Cambridge, MA: MIT Press.

Huang, L-D. et al. (2003). Maze routing with buffer insertion under transition time constraints. *IEEE Trans*. *Computer–Aided Design of Integrated Circuits and Systems*. 22(1):91-96.

Lai, M. and D. F. Wong. (2002). Maze routing with buffer insertion and wiresizing. *IEEE Trans*. *Computer–Aided Design of Integrated Circuits and Systems*. 21(10):1205-1209.

Li, Z. and W. Shi. (2006). An $O(bn^2)$ time algorithm for optimal buffer insertion with b buffer types. *IEEE Trans*. *Computer–Aided Design of Integrated Circuits and Systems*. 25(3):484-489.

Lillis, J., C.-K. Cheng and T.-T. Y. Lin. (1996). Optimal wire sizing and buffer insertion for low power and a generalized delay model. *IEEE Journal of Solid-State Circuits*. 31:437-447.

Rabaey, J. M., A. Chandrakasan and B. Nikolić. (2003). *Digital Integrated Circuits: A Design Perspective*. 2nd ed. Upper Saddle River, NJ: Pearson Education, Inc.

van Ginneken, L. P. P. P. (1990). Buffer placement in distributed RC-tree networks for minimal Elmore delay. *Proc*. *Int*. *Symp*. *Circuits and Systems*. :865-868.

Van Mieghem, P. and F. A. Kuipers. (2004). Concepts of exact QoS routing algorithms. *IEEE/ACM Trans*. *Networking*. 12(5):851-864.

Zhou, H. et al. (2000). Simultaneous routing and buffer insertion with restrictions on buffer locations. *IEEE Trans*. *Computer–Aided Design of Integrated Circuits and Systems*. 19(7):819-824.