Fast Algorithms For Slew Constrained Minimum Cost Buffering

Shiyan Hu†, Charles J. Alpert‡, *Fellow, IEEE*, Jiang Hu†, *Senior Member, IEEE*, Shrirang Karandikar‡, Zhuo Li‡, Weiping Shi†, *Senior Member, IEEE*, and C. N. Sze‡

***Abstract*— As a prevalent constraint, sharp slew rate is often required in circuit design which causes a huge demand for buffering resources. This problem requires ultra-fast buffering techniques to handle large volume of nets, while also minimizing buffering cost. This problem is intensively studied in this paper. First, a highly efficient algorithm based on dynamic programming is proposed to optimally solve slew buffering with discrete buffer locations. Second, a new algorithm using the maximum matching technique is developed to handle the difficult cases in which no assumption is made on buffer input slew. Third, an adaptive buffer selection approach is proposed to efficiently handle slew buffering with continuous buffer locations. Fourth, buffer blockage avoidance is handled, which makes the algorithms ready for practical use.**

Experiments on industrial netlists demonstrate that our algorithms are very effective and highly efficient: we achieve about $90\times$ speed up and save up to 20% buffer area over the commonly-used van Ginneken style buffering. The new algorithms also significantly outperform previous works that indirectly address the slew buffering problem.

***Index Terms*— Buffer insertion, slew constraint, interconnect, NP-Complete, input slew.**

I. Introduction

As VLSI technology moves to the 65 nm node and beyond, it has been well documented [1], [2] that the number of buffers on a chip is rising dramatically. Osler [2] cites two IBM ASIC designs where one-fourth of the gates are buffers. For some multi-million gate ASICs, more than a million buffers are required today. This is a surprise to no one as devices continue to scale more quickly than interconnects. Higher relative interconnect resistance forces buffers to be placed closer together to achieve optimal performance. In addition, interconnect resistivity also causes signal integrity to degrade more quickly with each advancing technology. Thus, buffers need to be inserted on long interconnects to meet slew constraints, even if these nets are not timing critical.

In reality, slew constraint is much more prevalent than timing constraint: it is reported in [2] that only a fraction (roughly 5-10%) of nets need to be re-buffered for delay optimization; for the remaining fraction (roughly 90-95%), the slew based buffer insertion was sufficient to meet the net's timing constraint. In other words, it is sufficient to buffer all nets to fix slew violations without worrying about delay. Those small fraction of buffered nets that subsequently show up as critical can then be re-buffered with a delay based objective function. In the IBM physical synthesis methodology [2], buffers are inserted for satisfying slew constraints early, so that timing analysis uses legal slew constraints. Later, buffers on critical nets are ripped up and re-buffered for delay.

The sheer number of buffers can degrade overall design performance by forcing the rest of the logic to be spread further apart to accommodate those buffers. The buffers themselves are a drain on power and can cause other gates to be sized to higher power levels since they are now further apart on the chip. Therefore, a significant part of the performance of the design depends on using as little buffering resources as possible. van Ginneken's algorithm [3] and its derivative extensions [4], [5], [6], [7], [8] are very effective for delay optimization. Further, Lillis' data structure [4] allows trading off delay for cost to more efficiently use buffer resources, yet this is still suboptimal for area.

From a practical point of view, slew buffering should be as important as timing driven buffering. Unfortunately, there is very little previous work on it. For related works that consider slew and/or noise constraints [4], [9], [5], [10], they still optimize for delay instead of handling these constraints separately. Buffering of non-critical nets using these techniques may result in unnecessary runtime and resource overhead. Note that the work of [11] also addresses slew constraints without regards to delay. However, that work does not actually model slew; it simplifies the slew constraint to be equivalent to a capacitance constraint which means that interconnect resistivity is not modelled. While appropriate for very large fanout nets (e.g., over 1000 sinks), it essentially becomes equivalent to length-based buffering [12]. Length-based buffering [12] tries to achieve a similar result of slew buffering in spirit. However, we show that it can be area inefficient especially in handling multi-fanout nets.

This work proposes a new buffering formulation: find the minimum area (or cost) buffering solution such that slew constraints are satisfied. In this formulation, one does not need to know required arrival time at sinks, so it can be used earlier in the design flow than traditional buffering. It can be done totally independently of timing analysis, i.e., incremental timing is not required between buffering of individual nets. Based on the new formulation, the general slew buffering problem is shown to be NP-Complete. Despite the difficulty of the problem, some highly efficient and practical algorithms

Shiyan Hu, Jiang Hu and Weiping Shi are with Department of Electrical and Computer Engineering, Texas A&M University, College Station, Texas 77843, Email: {hushiyan, jianghu, zhuoli, wshi}@ece.tamu.edu.

Charles J. Alpert, Shrirang Karandikar, Zhuo Li and C. N. Sze are with IBM Austin Research Laboratory 11501 Burnet Road, Austin, Texas 78758, Email: {alpert, shrirang, csze}@us.ibm.com.

are proposed in this paper:

1) For a single buffer type, an optimal linear time solution is achievable by greedy algorithm under the assumption that the input slew to each buffer is fixed.
2) For multiple buffer types, a very efficient optimal slew buffering algorithm is designed under the assumption that the input slew to each buffer is fixed. Experiments show that compared to slew constrained timing buffering, about $90\times$ speedup is achieved while still saving area.
3) If the input slew to each buffer is not fixed, the dynamic programming cannot be easily applied since the upstream knowledge is needed to compute the input slew. We propose a maximum matching based new algorithm to handle this difficult case. Experimental results demonstrate that up to 21.9% buffer area can be further saved.
4) When buffer positions can be freely chosen, slew buffering may allow more efficient buffer usage. A continuous slew buffering algorithm incorporating adaptive buffer selection idea is proposed for this purpose. It handles 1000 nets in only 30 seconds and often extra 5% buffer area saving can be obtained.
5) Buffering with blockage is handled in this paper, which makes the algorithms ready for practical use.

Although there is a close relationship between slew buffering and timing buffering, the two buffering algorithms are actually very different. For example, in slew buffering, inserting one buffer may only generate one new non-dominated solution. However, in timing buffering, numerous new non-dominated solutions can be introduced. Refer to Section IV-B.2 for details.

The rest of the paper is organized as follows: Section II formulates the slew buffering problem. Section III presents the NP-Completeness proof for the general slew buffering problem. Section IV describes the proposed slew buffering algorithms. Section V describes two related buffering algorithms for comparison. Section VI presents the experimental results with analysis. A summary of work is given in Section VII.

II. Preliminaries

The input to the slew buffering problem includes a routing tree $T = (V, E)$, where $V = \{s_0\} \cup V_s \cup V_n$, and $E \subseteq V \times V$. For simplicity, the routing tree is assumed to be a binary tree in this paper. Trees in other topologies can be converted to a binary tree (see, e.g., [7]). Vertex s_0 is the *source* vertex, V_s is the set of *sink* vertices and V_n is the set of *internal* vertices. Each sink vertex $s \in V_s$ is associated with sink capacitance C_s. Each edge $e \in E$ is associated with lumped resistance R_e and capacitance C_e. A buffer library B contains different types of buffers. Each type of buffer b has a cost W_b, which can be measured by area or any other metric, depending on the optimization objective. Without loss of generality, we assume that the driver at source s_0 is also in B. A function $f : V_n \rightarrow 2^B$ specifies the types of buffers allowed at each internal vertex. That is, for each vertex v, $f(v)$, which is a subset of 2^B, specifies the buffer types allowed at v.

The *slew rate* of a signal refers to the rising or falling time of a signal switching. A commonly used definition of slew is the 10/90 slew and it is adopted in this paper, where 10/90 slew refers to the time difference between when the waveform crosses the 90% point and the 10% point. Some other definitions, such as 20/80 or 30/70 slew, are also used in practice when the waveform has slowly rising or falling tail. The slew model employed in this work is chosen for its simplicity and is essentially equivalent to the Elmore model for delay. More accurate wire and gate delay models may be used if more accuracy is desired. Given that the motivation for the proposed buffering formulation lies in the requirement to efficiently buffer a large number of nets, this slew model is appropriate.

The slew model can be explained using a generic example which is a path p from node v_i (upstream) to v_j (downstream) in a buffered tree. There is a buffer (or the driver) b_u at v_i, and there is no buffer between v_i and v_j. The slew rate $S(v_j)$ at v_j depends on both the output slew $S_{b_u,out}(v_i)$ at buffer b_u and the slew degradation $S_w(p)$ along path p (or wire slew), and is given by [13]:

$$S(v_j) = \sqrt{S_{b_u,out}(v_i)^2 + S_w(p)^2}. \tag{1}$$

The slew degradation $S_w(p)$ can be computed with Bakoglu's metric [14] as

$$S_w(p) = \ln 9 \cdot D(p), \tag{2}$$

where $D(p)$ is the Elmore delay from v_i to v_j.

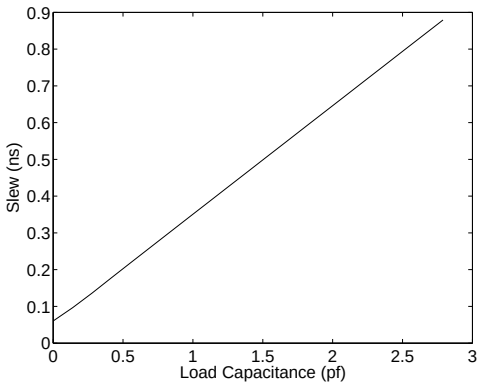

Fig. 1. The slew-capacitance curve of an inverter.

The output slew of a buffer, such as b_u at v_i, depends on the input slew at this buffer and the load capacitance seen from the output of the buffer. Usually, the dependence is described as a 2-D lookup table. In addition to handling the general case of arbitrary input slew, our work includes fast algorithms assuming a fixed input slew which is normally a conservative estimation (the slew constraint). This assumption allows us to process large volume of nets quickly with small solution degradation. For fixed input slew, the output slew of buffer b at vertex v is then given by

$$S_{b,out}(v) = R_b \cdot C(v) + K_b, \tag{3}$$

where $C(v)$ is the downstream capacitance at v, R_b and K_b are empirical fitting parameters. This is similar to empirically

derived K-factor equations [15]. We call R_b the slew resistance and K_b the intrinsic slew of buffer b. Figure 1 shows a slew curve of one inverter generated by EinsTimer [16]. The linear order model is quite reasonable as seen from Figure 1.

A buffer assignment γ is a mapping $\gamma : V_n \rightarrow B \cup \{\overline{b}\}$ where $\overline{b}$ denotes that no buffer is inserted. The cost of a solution γ is $W(\gamma) = \sum_{b \in \gamma} W_b$. With the above notations, the basic slew buffering problem can be formulated as follows.

Discrete Slew Constrained Minimum Cost Buffer Insertion Problem: Given a binary routing tree $T = (V, E)$, possible buffer positions, and a buffer library B, to compute a buffer assignment γ such that the total cost $W(\gamma)$ is minimized such that the input slew at each buffer or sink is no greater than a constant α.

Note that the continuous slew buffering problem is also considered in this paper where buffer positions can be freely chosen in a routing tree. A first glance at the above closed form model might suggest close relationship between timing buffering and slew buffering, however, they actually significantly differ. A detailed analysis is presented in Section IV-B.2. Before closing this section, we note the following computational complexity result:

Theorem 1: The minimum cost slew buffering problem is NP-Complete, if the size of the buffer library is not constant and the cost of each buffer can be an arbitrary integer.

Refer to Section III for the proof. Since the size of the buffer library is bounded and the buffer area is not an arbitrary value in reality, our algorithms perform very well in practice.

III. Complexity of Slew Buffering Problem

Proof of Theorem 1:

The problem is clearly in NP. We reduce from the minimum cost timing buffering problem with unbounded buffer library size[1] to show that the minimum cost slew buffering problem with unbounded buffer library size is NP-Complete. Let Q, R, C, W denote the required arrival time (RAT), resistance, capacitance and cost, respectively. It is shown in [6] that computing a timing buffering for the tree in Figure 2 with RAT at driver $Q_{s_0} \geq 0$ and the total buffer cost at most $M = N + \sum_{i=1}^{n} N^i$ is NP-Complete. Driver resistance is set to $R_{s_0} = N^n$, sink capacitance and sink RAT are listed in Table I, and the buffer library information is shown in Table II, where N is a sufficiently large positive integer, $x_1, x_2, \ldots, x_{2n}$ are positive integers such that $\sum_1^{2n} x_i = 2N$, and there are n sinks and $2n$ buffer types in the buffer library.

We set intrinsic slew and intrinsic delay to zero, and slew resistance equal to driving resistance for each buffer type and driver. Furthermore, every edge in the tree has zero wire capacitance and zero wire resistance. It is then easy to check that the slew rate is equal to the delay in value. For example, delay and slew rate at v_1 are both $R_{s_0} \cdot C_{b_1}$ assuming that b_1 is placed at v_1.

Given an instance of minimum cost timing buffering problem, we construct an instance of minimum cost slew buffering problem as follows. We reuse the routing tree in Figure 2 except that each sink s_i is changed to a sink s_i', where

[1]That is, the number of buffer types is not constant.

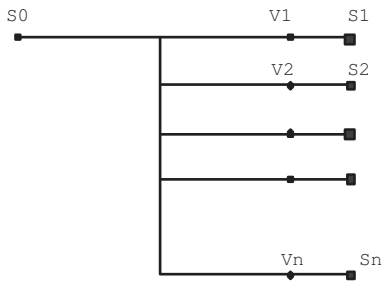

Fig. 2. Underlying routing tree and buffer positions [6].

TABLE I
C, Q VALUES FOR SINKS [6].

Sink s_i	Sink capacitance C_{s_i}	Sink RAT Q_{s_i}
s_1	N^{n+2}	$N^{n+1} + N^{n+2}$
s_2	N^{n+1}	$N^{n+1} + N^{n+2}$
...	...	...
s_n	N^3	$N^{n+1} + N^{n+2}$

$C_{s_i'} = \frac{C_{s_i}}{N}$. A critical fact used in [6] is that every buffer position v_i must be inserted with a buffer, and this buffer must be either b_{2i-1} or b_{2i}.

We claim that there is a solution for the instance of slew buffering problem with slew constraint $\alpha = N^{n+1}$ and the total buffer cost at most M if and only if there is a solution for the instance of minimum cost timing buffering problem with $Q_{s_0} \geq 0$ and with the same cost bound.

We begin with the "only if" direction. Since slew constraint is set to N^{n+1}, it follows that the delay between s_0 and v_i and delay between v_i and s_i' is no more than N^{n+1} each. Since $C_{s_i'} = \frac{C_{s_i}}{N}$, delay between v_i and s_i is bounded above by N^{n+2}. Noting that $Q_{s_i} = N^{n+2} + N^{n+1}$, we have $Q_{s_0} \geq 0$.

For the "if" direction, since we must insert one of b_{2i-1} and b_{2i} at every v_i, delay between v_i and s_i is N^{n+2}, and thus the slew rate at s_i' is N^{n+1}. Since total delay from s_0 to any sink s_i is no larger than $N^{n+1}+N^{n+2}$, one sees that delay between s_0 and any v_i is bounded above by N^{n+1}. Therefore, in the buffered tree, slew rate at any buffer position/sink is bounded above by α, which completes the proof. □

IV. Slew Constrained Minimum Cost Buffering Algorithms

A. Overview of Classic Timing-Driven Buffering

To understand the context of the presented algorithms and to define notation, this section begins with a brief overview of van Ginneken/Lillis [3], [4] algorithm. The algorithm proceeds bottom-up from the leaf nodes toward the driver along a given routing tree. A set of candidate solutions is kept updated during the process. Each solution is associated with a three-tuple (C, W, Q), where C denotes the downstream capacitance at the current node, W denotes the cost (i.e., area) of the solution and Q refers to the required arrival time (RAT).

Suppose that a solution γ_v at position v must "propagate" to an upstream position u and there is no branching point in between. If no buffer is placed at u, then only wire delay

TABLE II
C, R, W VALUES FOR EACH BUFFER TYPE [6].

Buffer b_i	Driving resistance R_{b_i}	Input capacitance C_{b_i}	Cost (Area) W_{b_i}
b_1	1	x_1	$x_2 + N^n$
b_2	1	x_2	$x_1 + N^n$
b_3	N	x_3	$x_4 + N^{n-1}$
b_4	N	x_4	$x_3 + N^{n-1}$
...	...	...	...
b_{2n-1}	N^{n-1}	x_{2n-1}	$x_{2n} + N$
b_{2n}	N^{n-1}	x_{2n}	$x_{2n-1} + N$

needs to be considered. Therefore, the new solution γ_u can be computed as

$$\begin{array}{rcl} C(\gamma_u) & = & C(\gamma_v) + C_e, \\ W(\gamma_u) & = & W(\gamma_v), \\ Q(\gamma_u) & = & Q(\gamma_v) - D_e, \end{array} \tag{4}$$

where $e = (u, v)$ and $D_e = R_e(\frac{C_e}{2} + C(\gamma_v))$. Otherwise, suppose that we add a buffer b_i at u. γ_u can be then computed as

$$\begin{array}{rcl} C(\gamma_u) & = & C_{b_i}, \\ W(\gamma_u) & = & W(\gamma_v) + W_{b_i}, \\ Q(\gamma_u) & = & Q(\gamma_v) - D_{b_i} - D_e \end{array} \tag{5}$$

after buffer insertion. In Eqn. (5), D_{b_i} refers to the buffer delay and is computed as $D_{b_i} = R'_{b_i} \cdot C(u) + K'_{b_i}$, where R'_{b_i} is the driving resistance of b_i but not the slew resistance of b_i, and K'_{b_i} is the intrinsic buffer delay.

An important concept in van Ginneken/Lillis algorithm are *non-dominated solutions*. For any two solutions γ_1, γ_2 *at the same node*, γ_1 *dominates* γ_2 if $C(\gamma_1) \leq C(\gamma_2)$, $W(\gamma_1) \leq W(\gamma_2)$ and $Q(\gamma_1) \geq Q(\gamma_2)$. Whenever a solution becomes dominated, it is removed from the solution set. Therefore, only solutions excel in at least one aspect of downstream capacitance, buffer cost and RAT can survive.

For handling branch merging, suppose that we have obtained all the non-dominated solutions of left branch T_l and right branch T_r at a branching point v_t[1]. Denote the left-branch solution set and the right-branch solution set by Γ_l and Γ_r, respectively. The merging process is performed as follows. For each solution $\gamma_l \in \Gamma_l$ and each solution $\gamma_r \in \Gamma_r$, generate a new solution γ' according to:

$$\begin{array}{rcl} C(\gamma') & = & C(\gamma_l) + C(\gamma_r), \\ W(\gamma') & = & W(\gamma_l) + W(\gamma_r), \\ Q(\gamma') & = & \min\{Q(\gamma_l), Q(\gamma_r)\}. \end{array} \tag{6}$$

At a high level, van Ginneken/Lillis algorithm builds the solution set in a bottom-up fashion. Assume that we have computed all feasible non-dominated solutions at a buffer position v. For the immediately upstream buffer position u (without passing any branching point), we first propagate all solutions up there through performing wire insertion of (u, v) to each solution. The propagated solutions resemble the choices when no buffer is inserted at u. Subsequently, for each propagated solution, we compute a new solution for inserting each buffer. The new solution is inserted into the solution set as long as it is not dominated by any existing one. The solution set is meanwhile updated to prune the solutions being dominated by the newcomer. At a merging point, we carry out the process just described to generate the new solution set. In this way, we keep climbing up the routing tree until the driver is met. After pruning solutions violating the timing constraint at driver, we select the best solution as the one with the smallest cost.

[1]For two branches, we arbitrarily assign them to be left branch and right branch.

B. Discrete Slew Buffering Assuming Fixed Input Slew

1) Algorithm: Our algorithms share the same dynamic programming framework as timing buffering [3], [4] in appearance, but have critical underlying differences which will be analyzed in Section IV-B.2 and Section IV-B.3.

In the dynamic programming framework, a set of candidate solutions are propagated from the sinks toward the source along the given tree. Each solution γ is characterized by a three-tuple (C, W, S), where C denotes the downstream capacitance at the current node, W denotes the cost of the solution and S is the accumulated slew degradation S_w defined in Eqn. (2). At a sink node, the corresponding solution has C equal to the sink capacitance, $W = 0$ and $S = 0$. The solution propagation is accomplished by the following operations.

Consider to propagate solutions from a node v to its parent node u through edge $e = (u, v)$. A solution γ_v at v becomes solution γ_u at u, which can be computed as $C(\gamma_u) = C(\gamma_v) + C_e, W(\gamma_u) = W(\gamma_v)$ and $S(\gamma_u) = S(\gamma_v) + \ln 9 \cdot D_e$ where $D_e = R_e(\frac{C_e}{2} + C(\gamma_v))$.

In addition to keeping the unbuffered solution γ_u, a buffer b_i can be inserted at u to generate a buffered solution $\gamma_{u,buf}$ which can be then computed as $C(\gamma_{u,buf}) = C_{b_i}, W(\gamma_{u,buf}) = W(\gamma_v) + W_{b_i}$ and $S(\gamma_{u,buf}) = 0$.

When two sets of solutions are propagated through left child branch and right child branch to reach a branching node, they are merged. Denote the left-branch solution set and the right-branch solution set by Γ_l and Γ_r, respectively. For each solution $\gamma_l \in \Gamma_l$ and each solution $\gamma_r \in \Gamma_r$, the corresponding merged solution γ' can be obtained according to: $C(\gamma') = C(\gamma_l) + C(\gamma_r), W(\gamma') = W(\gamma_l) + W(\gamma_r)$ and $S(\gamma') = \max\{S(\gamma_l), S(\gamma_r)\}$. To ensure that the worst case in the two branches still satisfies slew constraint, we take the maximum slew degradation for the merged solution.

For any two solutions γ_1, γ_2 *at the same node*, γ_1 *dominates* γ_2 if $C(\gamma_1) \leq C(\gamma_2)$, $W(\gamma_1) \leq W(\gamma_2)$ and $S(\gamma_1) \leq S(\gamma_2)$. Whenever a solution becomes dominated, it is pruned from the solution set without further propagation. A solution γ can also be pruned when it is infeasible, i.e., either its accumulated slew degradation $S(\gamma)$ or the slew rate of any downstream buffer in γ is greater than the slew constraint α.

2) Critical Differences from Timing Buffering: When a buffer b_i is inserted into a solution γ, $S(\gamma)$ is set to zero and $C(\gamma)$ is set to $C(b_i)$. This means that inserting one buffer may bring *only one new solution*, namely, the one with the smallest area, W. However, in minimum cost timing buffering, a buffer insertion may result in many non-dominated (C, W, Q) tuples

with the same C value, where Q denotes the require arrival time.

Consequently, in slew buffering, at each buffer position *along a single branch*, at most $|B|$ new solutions can be generated through buffer insertion since C, S are the same after inserting each buffer. In contrast, buffer insertion in the same situation may introduce many new solutions in timing buffering. This sheds light on why slew buffering can be much more efficiently computed.

Another important fact is that the slew constraint is in some sense close to length constraint. In slew buffering, solutions can soon become infeasible if we do not add a buffer into it and thus many solutions, which are only propagated through wire insertion, are often removed soon. An extreme case demonstrating this point is that in standard timing buffering, the solutions with no buffer inserted can always live until being pruned by driver given a loose timing constraint. This may not happen in slew buffering: this kind of solutions soon become infeasible as long as the slew constraint is not too loose.

Due to these special characteristics of the slew buffering problem, a *linear time* optimal algorithm for buffering with a single buffer type is possible. In timing buffering, it is not known how to design a *polynomial time* algorithm in this case. Refer to Section IV-D for the details. From these facts, the basic differences between these two somewhat related buffering problems are clear.

3) Implementation Experiences: This section presents a fast algorithm for the slew buffering problem. Except for special efforts for handling S, the new algorithm works as [4]. Refer to Figure 3 for the pseudocode of the proposed algorithm. For consistency, we insert a dummy buffer b_0 to a position when no buffer is to be inserted there.

The bolded part in Figure 3 shows the difference between the slew buffering algorithm and [4]. First, S, which represents accumulated slew degradation on wire, is a newly introduced term and thus does not exist in van Ginneken/Lillis' algorithm. Second, the SolutionSetUpdate procedure shown in Figure 4 significantly differs: a new solution is first checked for feasibility; if the slew constraint is satisfied, the domination check/elimination procedure for the solution set will be carried out.

We are to elaborate some implementation details in domination check as well as domination elimination. In the algorithm, the solution set is stored using a linked list where elements are in no particular order. The straightforward linear search is carried out into the solution list by each newcomer for domination checking and meanwhile, the solution list is updated for domination elimination. This simple implementation gives excellent performance due to the critical fact that size of solution set here is always small. We usually have less than 20 non-dominated solutions at driver in each routing tree, and the typical total runtime over 1000 nets is less than 20 seconds. Refer to Section VI for the details.

Therefore, in contrast to using range search tree to prune the dominated solutions as in [4], the simple linked list implementation works very well here. We believe that the simplicity of implementation for slew buffering with fixed buffer input slew will make it widely used in practice.

Algorithm: Slew buffering w/ fixed input slew.

Input: T: routing tree, B: buffer library, α: slew constraint
Output: minimum cost buffer assignment γ satisfying α

1. for each sink s, build a solution set $\{\gamma_s\}$, where $W(\gamma_s) = 0$, $\boldsymbol{S(\gamma_s) = 0}$, and $C(\gamma_s) = C_s$
2. for each branching point/driver v_t in the order given by a postorder traversal of T, let T' be the two branches T_1, T_2 of v_t and Γ' be the solution set corresponding to T', do
3. for each wire e in T', in a bottom-up order, do
4. for each $\gamma \in \Gamma'$ corresponding to T', do
5. $C(\gamma) = C(\gamma) + C_e$
6. **set $\boldsymbol{S(\gamma) = S(\gamma) + \ln 9 \cdot D_e}$**
7. **SolutionSetUpdate $\boldsymbol{(\gamma, \Gamma', b_0, \alpha)}$**
8. if the current position allows buffer insertion, then
9. for each buffer type $b_i \in B$, do
10. for each $\gamma \in \Gamma'$, generate a new solution γ'
11. set $C(\gamma') = C_{b_i}$
12. set $W(\gamma') = W(\gamma') + W_{b_i}$
13. **set $\boldsymbol{S(\gamma') = 0}$**
14. **SolutionSetUpdate $\boldsymbol{(\gamma', \Gamma', b_i, \alpha)}$**
15. // merge Γ_1 and Γ_2 to Γ_{v_t}
16. set $\Gamma_{v_t} = \emptyset$
17. for each $\gamma_1 \in \Gamma_1$ and $\gamma_2 \in \Gamma_2$, generate a new solution γ'
18. set $C(\gamma') = C(\gamma_1) + C(\gamma_2)$
19. set $W(\gamma') = W(\gamma_1) + W(\gamma_2)$
20. **set $\boldsymbol{S(\gamma') = \max\{S(\gamma_1), S(\gamma_2)\}}$**
21. **SolutionSetUpdate $\boldsymbol{(\gamma', \Gamma_{v_t}, b_0, \alpha)}$**
22. eliminate infeasible solutions at driver and return γ with the smallest cost

Fig. 3. Slew constrained minimum cost buffering algorithm with fixed buffer input slew.

Procedure: SolutionSetUpdate w/ fixed input slew

Input: γ': a candidate solution, Γ: a solution set, b: a buffer type, α: a slew constraint
Output: an updated solution set Γ

1. // check whether γ' violates the slew constraint
2. if $b = b_0$, then
3. return Γ if $S(\gamma') > \alpha$
4. else
5. return Γ if $\sqrt{S(\gamma')^2 + (R_b \cdot C(\gamma') + K_b)^2} > \alpha$
6. // domination check and domination elimination
7. for each solution $\gamma \in \Gamma$, do
8. if $C(\gamma) \le C(\gamma')$, $W(\gamma) \le W(\gamma')$ and $S(\gamma) \le S(\gamma')$,
9. return Γ
10. if $C(\gamma') \le C(\gamma)$, $W(\gamma') \le W(\gamma)$ and $S(\gamma') \le S(\gamma)$,
11. remove γ from Γ
12. insert γ' into Γ and return Γ

Fig. 4. Procedure of updating solution set for slew buffering with fixed buffer input slew.

One would wonder the effect of introducing the range search tree into the slew buffering algorithm. As such, the slew buffering algorithm combined with range search tree pruning [4] is also tested. Unfortunately, the slew buffering algorithm is slowed down. This phenomenon is due to the considerable amount of inherent overhead in maintaining the balanced binary search tree through e.g., rotation for each insertion/deletion in the data structure. Refer to Section VI for the details.

Recall that at each buffer position, we introduce $|B|$ new solutions by buffer insertion. Thus, a branch having n buffer positions will introduce at most $n|B|$ *new* solutions. Consider to merge two branches each of which has $n_1|B|$ and $n_2|B|$

solutions, respectively, where n_1 and n_2 denote the number of buffer positions in each branch. After merging, the number of solutions is bounded by $n_1 n_2 |B|^2$. Suppose that another branch merging produces $n_3 n_4 |B|^2$ solutions. Further suppose that these resulting solutions are merged and $n_1 n_2 n_3 n_4 |B|^4$ solutions are obtained. Let n denote the number of buffer positions, m denote the number of sinks, and n_i denote the number of buffer positions at branch i. By the above process, one can see that the total number of solutions is bounded by $(\frac{n}{m})^m |B|^m$, since $n_1 n_2 \cdots n_m \leq (\frac{n}{m})^m$ and $n_1 + n_2 + \ldots + n_m = n$.

Since we can have at most $O((\frac{n|B|}{m})^m)$ solutions at any position, a domination check at a position is performed through a traversal of the linked list consisting of $O((\frac{n|B|}{m})^m)$ solutions and thus needs $O((\frac{n|B|}{m})^m)$ time per traversal. At a buffer position, $|B|$ new solutions are introduced due to buffer insertions. The domination checks need $|B|$ traversals of the solution set, which takes $O(|B|(\frac{n|B|}{m})^m)$ time. The most time-consuming step is branch merging where we at most perform $O((\frac{n|B|}{m})^m)$ domination checks since it is the upper bound for the number of solutions at any position. Thus, a branch merging needs $O(((\frac{n|B|}{m})^m)^2)$ time. Since there are n buffer positions, the proposed algorithm returns the optimal solution in $O(n|B| \cdot (\frac{n|B|}{m})^{2m})$ time. Theoretically (though impractically), when $|B| = \Theta(n), m = \Theta(n)$, the algorithm will run in exponential time. This surprises no one given the NP-Completeness nature of the slew buffering problem.

C. Discrete Buffering without Input Slew Assumptions

1) Basic modifications: In Section IV-B.1, the output slew of a buffer (computed by Eqn. (3)) does not depend on the input slew. This is valid since slew resistance R_{b_i} is obtained by assuming the input slew for each buffer to be fixed at the slew constraint. Certainly, improvement in buffer area is desired if this assumption is eliminated. As such, a more complicated dynamic programming algorithm which handles *non-fixed input slew* is proposed as follows.

Our idea is to approximate continuous-valued input slew by different small-sized slew bins. That is, the input slew at each buffer position is discretized into different input slew bins, each of which covers a range of slew rate. Clearly, better results can be obtained with finer input slew bins. Denote by l the number of input slew bins.

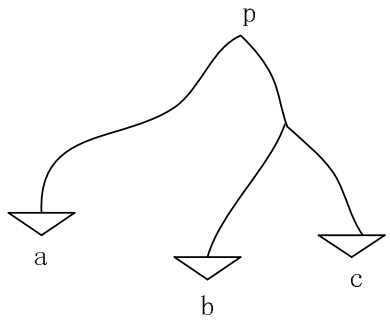

Fig. 5. An example of handling non-fixed input slew.

Suppose that a buffer is to be inserted at position p and there are three *immediate downstream* buffers in a solution γ as shown in Figure 5. As the result upstream from p is not yet known, the input slew to the buffer can be in any slew bin.

As such, in addition to C, W, S, each solution is augmented with new tuples L, U, which specify the lower bound and upper bound of the input slew to these immediate downstream buffers, respectively. In other words, the input slew is required to fall in $[L, U)$. Suppose that viewing at p, we have $n(\gamma)$ immediate downstream buffers, each of which is associated with a lower bound L_i and an upper bound U_i. Accordingly, there is an S_i representing accumulated slew degradation viewing at each immediate downstream buffer. For example, a solution at p in Figure 5 has (S_1, L_1, U_1) for the buffer inserted at a, (S_2, L_2, U_2) for b, and (S_3, L_3, U_3) for c. Therefore, each solution is characterized by (C, W, S_i, L_i, U_i), $1 \leq i \leq n$ if the solution has n immediate downstream buffers.

When a buffer is inserted at p, *at most* l new solutions are generated. They are with the same C, W, S values but with different L, U values. We say "at most" since whether a buffer with a certain input slew bin can be inserted at p needs to be *validated*. For a buffer b to be inserted with the input slew bin g, denote by $[\underline{S_g}, \overline{S_g})$ the slew range of g. The buffer insertion is valid if for each immediate downstream buffer i (viewing at p, $1 \leq i \leq n(\gamma)$) in γ,

$$L_i(\gamma) \leq \sqrt{S_{b,out}(p, g, C(\gamma))^2 + S_i(\gamma)^2} \leq U_i(\gamma), \qquad (7)$$

where $S_{b,out}(p, g, C(\gamma))$ is the output slew of the buffer b at p with g as its input slew bin and $C(\gamma)$ as its downstream capacitance, and a lookup table is used to obtain its value. Upon validation, the buffer b is inserted to γ, the number of immediate downstream $n(\gamma)$ is set to one, $S_1(\gamma)$ is set to zero, and $L_1(\gamma) = \underline{S_g}$ and $U_1(\gamma) = \overline{S_g}$.

It is often valid for a buffer with numerous input slew bins to be inserted to the same solution γ. For efficiency reason, those new solutions are merged after buffer insertion. That is, after buffer insertion, two solutions γ_1 and γ_2 are merged to form γ' if $C(\gamma_1) = C(\gamma_2), W(\gamma_1) = W(\gamma_2)$ and $U_1(\gamma_1) = L_1(\gamma_2)$, where C, W, S of γ' remain unchanged while $L_1(\gamma') = L_1(\gamma_1)$ and $U_1(\gamma') = U_1(\gamma_2)$.

Note that in branch merging, the parameter values (S, L, U) of all immediate downstream buffers for a left-branch solution γ_1 and a right-branch solution γ_2 are stored together and $n(\gamma') = n(\gamma_1) + n(\gamma_2)$.

2) Reduction to maximum bipartite matching: The definition of domination needs to be accordingly modified. For two solutions with the same number of immediate downstream buffers, domination is defined *solely* on C, W, S_i, L_i, U_i. In particular, the i-th buffer in γ_1 and that in γ_2 may refer to different immediate downstream buffers. This allows a fairly effective solution pruning procedure.

Given two solutions γ_1 and γ_2, we are to decide whether there is a pairing of immediate downstream buffers of γ_1 and γ_2, respectively, such that $S_{\pi_1(j)}(\gamma_1) \leq S_{\pi_2(j)}(\gamma_2)$, $L_{\pi_1(j)}(\gamma_1) \leq L_{\pi_2(j)}(\gamma_2)$ and $U_{\pi_1(j)}(\gamma_1) \geq U_{\pi_2(j)}(\gamma_2)$ for each pair j where $1 \leq j \leq n(\gamma_1) = n(\gamma_2)$, and $\pi(\cdot)$ denotes the permutation of indices of immediate downstream buffers. If this is the case, together with $C(\gamma_1) \leq C(\gamma_2)$, $W(\gamma_1) \leq W(\gamma_2)$, we conclude that γ_1 dominates γ_2.

An example would be helpful to illustrate the above definition. Assume that γ_1, γ_2 both have three immediate downstream buffers. Suppose that (S_i, L_i, U_i) for γ_1 are $(3, 10, 60), (5, 30, 65), (3, 20, 50)$, and for γ_2 are $(5, 25, 35), (6, 50, 55), (10, 15, 35)$. γ_1 dominates γ_2 on (S, L, U) since $(3, 10, 60)$ dominates $(10, 15, 35)$, $(5, 30, 65)$ dominates $(6, 50, 55)$, and $(3, 20, 50)$ dominates $(5, 25, 35)$.

Given two solutions, we need to answer whether such pairing exists. The straightforward computation is inefficient since L, U may heavily overlap. As such, we reduce it to the maximum bipartite matching problem for an efficient solution. To check whether γ_1 dominates γ_2, for each $(S_i(\gamma_1), L_i(\gamma_1), U_i(\gamma_1))$ in γ_1, a set of tuples, denoted by $\psi_i(\gamma_1)$, consisting of all $(S_j(\gamma_2), L_j(\gamma_2), U_j(\gamma_2))$ in γ_2 is computed such that the former three-tuple dominates each of the latter three-tuples. A graph $G = (V, E)$ is constructed as follows. Represent each three-tuple by a vertex. A vertex corresponding to the i-th tuple in γ_1 links to the vertices corresponding to $\psi_i(\gamma_1)$. A bipartite graph is formed in this way since there are no links between nodes representing tuples in the same solution. For these two groups of vertices, the task is to answer whether there is a node-wise pairing (each from different groups) of cardinality $n(\gamma_1)$.

This is a maximum matching problem, which is to compute an edge set E' of maximum cardinality from E such that each vertex in V is incident to at most one edge of E'. Domination (on S, L, U) follows if E' is of cardinality of $n(\gamma_1)$. The best bipartite matching algorithm runs in $O(\sqrt{|V|}|E| \log(|V|^2/|E|)/\log|V|)$ time [17]. In this paper, an efficient practical implementation [18] based on the scaling push-relabel approach, is adopted. Refer to Figure 6 and Figure 7 for the algorithms for slew buffering without fixed input slew assumption.

We are to present the complexity analysis of the slew buffering algorithm with non-fixed input slew. Instead of $|B|$ new solutions in the fixed-input slew case, at most $l|B|$ new solutions can be generated at each buffer position due to buffer insertions in slew buffering algorithm with non-fixed input slew. Thus, the total number of solutions is always bounded above by $O((\frac{nl|B|}{m})^m)$ where n is the number of buffer positions and m is the number of sinks. The complexity analysis goes the same as in Section IV-B.3 except that one additional step, which is due to using maximum bipartite matching in domination check, needs to be considered. A solution can have at most m (S, L, U) tuples since there are only m sinks. Therefore, in our maximum matching problem, $|V| \leq m$ and $|E| \leq m^2$. Plugging these numbers into $O(\sqrt{|V|}|E| \log(|V|^2/|E|)/\log|V|)$, we have that the bipartite matching algorithm in [17] runs in $O(m^{2.5})$ time. A domination check needs to perform a traversal of the solution set and for each traversed solution, the above $O(m^{2.5})$ algorithm is carried out. Thus, a domination check needs $O((\frac{nl|B|}{m})^m \cdot m^{2.5})$ time. As in Section IV-B.3, the most time-consuming step is branch merging where $O((\frac{nl|B|}{m})^m)$ domination checks may happen. It is then easy to see that the total runtime is bounded above by $O(nl|B| \cdot (\frac{nl|B|}{m})^{2m} \cdot m^{2.5})$.

Algorithm: Slew buffering w/ non-fixed input slew.

Input: T, B, α, input slew bins
Output: minimum cost buffer assignment γ satisfying α

1. build solutions at each sink for each input slew bin
2. for each branching point/driver v_t in the order given by a postorder traversal of T, let T' be two branches T_1, T_2 of v_t and Γ' be the solution set, do
3. for each wire e in T', in a bottom-up order, do
4. for each $\gamma \in \Gamma'$ corresponding to T', do
5. $C(\gamma) = C(\gamma) + C_e$
6. set $S(\gamma) = S(\gamma) + \ln 9 \cdot D_e$
7. SolutionSetUpdate $(\gamma, \Gamma', b_0, \alpha)$
8. if the current position allows buffer insertion, then
9. for each $\gamma \in \Gamma'$,
10. for validated buffer type b_i and input slew bin g,
11. set $C(\gamma') = C_{b_i}$
12. set $W(\gamma') = W(\gamma') + W_{b_i}$
13. set $S(\gamma') = 0$
14. set $L_1(\gamma) = S_g$
15. set $U_1(\gamma) = \overline{S_g}$
16. SolutionSetUpdate $(\gamma', \Gamma', b_i, \alpha)$
17. // merge Γ_1 and Γ_2 to Γ_{v_t}
18. set $\Gamma_{v_t} = \emptyset$
19. for each $\gamma_1 \in \Gamma_1$ and $\gamma_2 \in \Gamma_2$, generate γ'
20. set $C(\gamma') = C(\gamma_1) + C(\gamma_2)$
21. set $W(\gamma') = W(\gamma_1) + W(\gamma_2)$
22. set $S(\gamma') = \max\{S(\gamma_1), S(\gamma_2)\}$
23. set L, U of γ' to be the union of L, U of γ_1, γ_2
24. SolutionSetUpdate $(\gamma', \Gamma_{v_t}, b_0, \alpha)$
25. eliminate infeasible solutions at driver and return γ with the smallest cost

Fig. 6. Slew constrained minimum cost buffering algorithm with non-fixed buffer input slew.

D. Continuous Slew Buffering

What we have considered so far is the discrete slew buffering problem. It is expected that the total buffer area can be reduced if buffer positions are freely chosen in the routing tree. The following continuous slew buffering algorithm settles this problem. We begin with a simple case:

Theorem 2: For a single buffer type, the optimal slew buffering can be computed in linear time under the assumption that the input slew to each buffer is fixed.

Proof: In essence, the algorithm is only propagating a single candidate up to the source. To insert buffers along a single path, we place a buffer as far (i.e., upstream) as possible from the previously inserted buffer such that the slew constraint is still satisfied. When proceeding to a branching point, a buffer is also placed as upstream as possible while the slew constraint must be satisfied for both branches. It is easy to see that given n buffer positions and sinks, this greedy algorithm returns the optimal solution in $O(n)$ time. □

Note that the above greedy algorithm can work in either discrete or continuous case. We now generalize this idea to handle multiple buffer types. As before, we place a buffer as upstream as possible from the previously inserted buffer such that the slew constraint is satisfied. The major difficulty is, of course, every type of buffers can be inserted at a position. Within a single branch, after a new solution is generated (i.e., a buffer is inserted), it is placed into a priority queue, which is decreasingly ordered by the distance from the current buffer

Procedure: SolutionSetUpdate w/ non-fixed input slew

Input: γ': a candidate solution, Γ: a solution set, b: a buffer type, α: a slew constraint
Output: an updated solution set Γ

1. // check whether γ' violates the slew constraint
2. if $b = b_0$, then
3. return Γ if $S(\gamma') > \alpha$
4. else
5. return Γ if $\sqrt{S(\gamma')^2 + (R_b \cdot C(\gamma') + K_b)^2} > \alpha$
6. // domination check and domination elimination
7. for each solution $\gamma \in \Gamma$, do
8. if γ dominates γ' based on (C, W, S, L, U),
9. return Γ
10. if γ' dominates γ based on (C, W, S, L, U),
11. remove γ from Γ
12. insert γ' into Γ and return Γ

Fig. 7. Procedure for updating solution set for slew buffering with non-fixed buffer input slew.

position to the root. The first element in the queue is then extracted as the next solution to be processed. For this solution, all types of buffers are inserted (each of which is placed as upstream as possible) and thus $|B|$ new solutions are generated and placed into the queue. As before, dominated solutions are pruned. For any two solutions γ_1, γ_2 where γ_1 resides at a position no lower than γ_2, γ_1 *dominates* γ_2 if $C(\gamma_1) \leq C(\gamma_2)$, $W(\gamma_1) \leq W(\gamma_2)$ and $S(\gamma_1) \leq S(\gamma_2)$.

The above exponential algorithm is found to be inefficient by our experiment. As such, an approximation algorithm through adaptively selecting candidate buffers is proposed. All buffers with area less than a threshold, called *filtered buffers*, are first increasingly sorted according to their slew resistance. For a slew constraint α, the first $\lceil c \cdot (e^{\alpha} - 1) \cdot |B| \rceil$ buffers (note that all $|B|$ buffers will be chosen when the value exceeds the number of filtered buffers) are selected to form the library for buffer insertion, where c is a constant and is experimentally determined to be 0.2. One can see that the number of buffer types to be investigated increases exponentially with the slew constraint. The idea behind this selection criterion reads as follows. Roughly speaking, for tight slew constraint, many buffers are needed and there will be many non-dominated solutions. Thus, our computation may only focus on a small number of buffers in order to reduce the size of the solution set. That is, we tradeoff solution quality for runtime. For loose slew constraint, a buffer will be inserted with a large gap from the previously inserted buffer and thus the solution set might not be very large. We can therefore choose more buffers (exponentially more in our case) to obtain high-quality solutions. Varying c, one can achieve different tradeoff between solution quality and runtime. Refer to Figure 8 and Figure 9 for the algorithms for continuous slew buffering with fixed input slew assumption. By combining the techniques in Section IV-C, we can easily obtain the algorithms for continuous slew buffering with non-fixed input slew. Refer to Figure 10 and Figure 11 for the algorithms for continuous slew buffering with non-fixed input slew. Finally we present some discussions about bounding the time complexity of the continuous slew buffering algorithm. Existing buffering algorithms often bound their time complexity using the number of candidate buffer positions. This is difficult in our case as candidate buffer positions are not well defined in our continuous buffering problem. Thus, we need to express our time bound using the smallest distance between any of two buffers such that the slew constraint is barely hold. However, bounding the time complexity in this way does not provide much insight in comparing our continuous slew buffering algorithm and other existing algorithms.

Algorithm: Continuous slew buffering w/ fixed input slew.

Input: T, B, α, input slew bins
Output: minimum cost buffer assignment γ satisfying α

1. building solutions at each sink
2. for each branching point/driver v_t in the order given by a postorder traversal of T, let T' be two branches T_1, T_2 of v_t and Γ' be the solution set, do
3. for each $\gamma \in \Gamma'$, do
4. for each selected buffer type b_i,
5. if b_i is necessary up to the branching point,
6. insert b_i into γ as upstream as possible to obtain γ'
7. else
8. propagate γ to v_t by adding wire to obtain γ'
9. SolutionSetUpdate $(\gamma', \Gamma', b_i, \alpha)$
10. // merge Γ_1 and Γ_2 to Γ_{v_t}
11. set $\Gamma_{v_t} = \emptyset$
12. for each $\gamma_1 \in \Gamma_1$ and $\gamma_2 \in \Gamma_2$, generate γ'
13. set $C(\gamma') = C(\gamma_1) + C(\gamma_2)$
14. set $W(\gamma') = W(\gamma_1) + W(\gamma_2)$
15. set $S(\gamma') = \max\{S(\gamma_1), S(\gamma_2)\}$
16. SolutionSetUpdate $(\gamma', \Gamma_{v_t}, b_0, \alpha)$
17. eliminate infeasible solutions at driver and return γ with the smallest cost

Fig. 8. Continuous slew constrained minimum cost buffering algorithm with fixed buffer input slew.

Procedure: SolutionSetUpdate (continuous, fixed input)

Input: γ': a candidate solution, Γ: a solution set, b: a buffer type, α: a slew constraint
Output: an updated solution set Γ

1. // check whether γ' violates the slew constraint
2. if $b = b_0$, then
3. return Γ if $S(\gamma') > \alpha$
4. else
5. return Γ if $\sqrt{S(\gamma')^2 + (R_b \cdot C(\gamma') + K_b)^2} > \alpha$
6. // domination check and domination elimination
7. for each solution $\gamma \in \Gamma$, do
8. if γ is at the same position or upstream to γ', and $C(\gamma) \leq C(\gamma')$, $W(\gamma) \leq W(\gamma')$ $S(\gamma) \leq S(\gamma')$,
9. return Γ
10. if γ is at the same position or downstream to γ' and $C(\gamma') \leq C(\gamma)$, $W(\gamma') \leq W(\gamma)$ and $S(\gamma') \leq S(\gamma)$,
11. remove γ from Γ
12. insert γ' into Γ and return Γ

Fig. 9. Procedure of updating solution set for continuous slew buffering with fixed buffer input slew.

E. Buffer Blockage Avoidance

In real circuits, some large area chunks may contain *buffer blockage*, which are macro or IP blocks allowing wire routing but not buffer insertion inside them. As such, a routing tree to be buffered might be re-routed to avoid blockage. For this purpose, we adopt a simultaneous buffer insertion and

Continuous slew buffering w/ non-fixed input slew.

Input: T, B, α, input slew bins
Output: minimum cost buffer assignment γ satisfying α

1. building solutions at each sink
2. for each branching point/driver v_t in the order given by a postorder traversal of T, let T' be two branches T_1, T_2 of v_t and Γ' be the solution set, do
3. for each $\gamma \in \Gamma'$, do
4. for each selected and validated buffer type b_i, and for each input slew bin g,
5. if b_i is necessary up to the branching point,
6. insert b_i into γ as upstream as possible to obtain γ'
7. set $C(\gamma') = C_{b_i}$
8. set $W(\gamma') = W(\gamma') + W_{b_i}$
9. set $S(\gamma') = 0$
10. set $L_1(\gamma) = S_g$
11. set $U_1(\gamma) = \overline{S_g}$
12. SolutionSetUpdate $(\gamma', \Gamma', b_i, \alpha)$
13. else
14. propagate γ to v_t by adding wire to obtain γ'
15. SolutionSetUpdate $(\gamma', \Gamma', b_i, \alpha)$
16. // merge Γ_1 and Γ_2 to Γ_{v_t}
17. set $\Gamma_{v_t} = \emptyset$
18. for each $\gamma_1 \in \Gamma_1$ and $\gamma_2 \in \Gamma_2$, generate γ'
19. set $C(\gamma') = C(\gamma_1) + C(\gamma_2)$
20. set $W(\gamma') = W(\gamma_1) + W(\gamma_2)$
21. set $S(\gamma') = \max\{S(\gamma_1), S(\gamma_2)\}$
22. set L, U of γ' to be the union of L, U of γ_1, γ_2
23. SolutionSetUpdate $(\gamma', \Gamma_{v_t}, b_0, \alpha)$
24. eliminate infeasible solutions at driver and return γ with the smallest cost

Fig. 10. Continuous slew constrained minimum cost buffering algorithm with non-fixed buffer input slew.

blockage avoidance approach in [19] which introduces very small additional wire in rerouting while keeps the solution quality. For completeness, we include some details of the approach in [19] here.

It is easy to re-route a path to avoid blockage if it contains no Steiner nodes. Otherwise, we start from the most downstream node inside the blockage, and move each node in turn such that each local move introduces smallest additional wire length. As in [19], we pay attention to the case where no adjustment on topology is needed even if there are Steiner nodes inside the blockage. Suppose that node v moves to v' for blockage avoidance. Downstream solutions need to be propagated to both v and v'. Of course, no buffer can be inserted at v. This propagation process continues and eventually, all solutions are merged at the first upstream node (during the bottom-up computation process) outside the blockage. Propagation to both nodes may result in efficient buffer usage. For example, if the original optimal solution for the problem without blockage does not insert any buffer into any blockage, it will still be returned. According to [19], the time complexity for this approach is bounded by $O(ng|B|h^2 + mk)$, where n denotes the number of buffer positions, m denotes the number of sinks, g denotes the maximal candidate solution set size, h denotes the maximal expanded Steiner node set, and k denotes the number of rectangular blockages.

V. Discussion of related approaches

A. Minimum cost slew constrained timing buffering

We refer to van Ginneken/Lillis' algorithm as *VGL* and the discrete slew buffering algorithm with fixed input slew as *SB*. In order to make a meaningful comparison between them, we first modify VGL to handle a slew constraint, without modifying its delay objective function. The new slew constrained VGL is referred to as *VGL+S*. In this way, we can investigate the difference between simply handling the slew constraint to optimize delay versus handling the slew constraint to optimize cost. For this, the three-tuple (C, W, Q) is augmented to (C, W, Q, S), where Q denotes the required arrival time. Note that domination in timing buffering is defined on C, W, Q but not on S, while S is only responsible for eliminating infeasible solutions. In contrast, domination in slew buffering is defined on C, W, S but not on Q. Therefore, VGL+S algorithm may delete optimal solutions based on timing information while our new algorithm, with domination defined on C, W, S can find the minimum cost solution satisfying slew constraint.

Let us look at a simple example illustrating the difference between timing buffering and slew buffering. Consider merging two solutions sets corresponding to two branches. Suppose that we have two solutions (represented by (C, W, Q, S)) $(3, 30, 70, 20), (5, 50, 80, 10)$ for the left branch, and two solutions $(3, 30, 85, 15), (5, 50, 92, 8)$ for the right branch. Assume that the slew constraint is 50. By timing buffering, we have two non-dominated solutions which are $(6, 60, 70, 20)$ and $(8, 80, 80, 15)$. However, by slew buffering, we have another non-dominated solutions (represented by (C, W, S)) which is $(10, 100, 10)$. Clearly in this case, solutions with a sharper slew rate are not deleted.

The experiments in the next section report the timing-driven buffering solution as the smallest cost (area) solution at the driver, thereby slack at the driver plays no role. In this way, the impact of the actual change in optimization strategy for area instead of delay is considered.

B. Capacitance-based buffering

We also compare slew buffering with another closely related buffering - capacitance-based buffering (CBB) [12], [11]. Roughly speaking, in capacitance-based buffering, downstream capacitance of a buffer cannot exceed the maximum capacitance it can drive, where a single typical buffer is used.

The capacitance-based buffering can be certainly computed in bottom-up fashion. Consider inserting two typical buffers at consecutive nodes v_j (upstream) and v_k (downstream), respectively, and the wirelength in between is the maximum possible value subject to the slew constraint. If the downstream capacitance load at v_j is $C(v_j)$, the capacitance constraint β is set to $\rho C(v_j)$, where ρ is a constant in $(0, 1)$. Note that β is a global constraint in CBB and has a value corresponding to each slew constraint.

VI. Experimental Results

A. Experiment Setup

For convenience, all algorithms in comparison are listed below together with their abbreviations.

Procedure: SolutionSetUpdate (continuous, non-fixed)
Input: γ': a candidate solution, Γ: a solution set, b: a buffer type, α: a slew constraint
Output: an updated solution set Γ
1. // check whether γ' violates the slew constraint
2. if $b = b_0$, then
3. return Γ if $S(\gamma') > \alpha$
4. else
5. return Γ if $\sqrt{S(\gamma')^2 + (R_b \cdot C(\gamma') + K_b)^2} > \alpha$
6. // domination check and domination elimination
7. for each solution $\gamma \in \Gamma$, do
8. if γ is at the same position or upstream to γ', and γ dominates γ' based on (C, W, S, L, U),
9. return Γ
10. if γ is at the same position or downstream to γ' and γ' dominates γ based on (C, W, S, L, U),
11. remove γ from Γ
12. insert γ' into Γ and return Γ

Fig. 11. Procedure of updating solution set for continuous slew buffering with non-fixed buffer input slew.

- SB: discrete slew buffering algorithm with fixed input slew, where input slew is equal to the slew constraint.
- SB+NI: discrete slew buffering with non-fixed input slew.
- C-SB: continuous slew buffering with fixed input slew.
- C-SB+NI: continuous slew buffering with non-fixed input slew.
- SB+B: discrete slew buffering w/ fixed input slew and blockage.
- VGL: van Ginneken/Lillis' min-cost timing buffering algorithm.
- VGL+S: slew constrained VGL.
- VGL+S+PSP: VGL with pre-buffer slack pruning technique [6].
- VGL+S+B: VGL+S with blockage.
- CBB: capacitance-based buffering algorithm.
- CWB: slew constrained buffering with pruning based on (C, W).

All algorithms are implemented in C++ and are tested on a Pentium IV computer with a 3.2GHz CPU and 1G memory. Our test cases are extracted from an industrial ASIC chip, which consist of 1000 nets with more than 50 thousand nodes including sinks, branching nodes and buffer positions. Among them, 757 nets have ≤ 5 sinks and all the remaining nets have ≤ 20 sinks. The sink capacitances range from $2.5fF$ to $200fF$. The wire resistance is $0.56\Omega/\mu m$ and the wire capacitance is $0.48fF/\mu m$. Another set of 100 large-degree nets are used to perform experiments to demonstrate the scalability of the algorithm, where the number of sinks ranges from 105 to 948.

The buffer library consists of 48 buffers, in which 23 are non-inverting buffers and 25 are inverting buffers. Normalized buffer areas range from 5 to 34, slew resistances range from $0.18ns/pF$ to $29.3ns/pF$, and input capacitances range from $2.1fF$ to $76.0fF$.

B. Comparison with Timing Buffering

We first compare SB with VGL+S, and results are summarized in Table III. Here "area saving" refers to the percentage difference in area, "speed up" refers to the percentage difference in CPU time (seconds), and the slew constraint is given in nanoseconds. Note that in SB, the buffer input slew is set to the slew constraint. In VGL+S, range search tree pruning is implemented as in [4]. We make the following observations:

- The number of buffers decreases and the area decreases for both algorithms as the slew constraint loosens. This makes sense since a looser constraint means that buffers can be spaced further apart.
- SB is more efficient in area. For example, with a 1.0 ns slew constraint, the area savings is 5.6% compared to VGL+S.
- The slew buffering algorithm SB is much more efficient. Despite considering all 48 buffers in the library, it runs in just a few seconds on 1000 nets. Furthermore, it runs over 88 times faster than the timing buffering algorithm for slew constraint $\alpha = 1.0$. The main reason for this fact is that there is a significantly smaller set of non-dominated solutions in slew buffering than in timing buffering. For example, when $\alpha = 1.0$, we have only 13 solutions per net in the slew buffering, while the number is 299 in the slew constrained timing buffering. This is caused by the fact that slew gets to be reset to zero whenever a buffer is inserted, while delay has to be propagated up the entire tree. In practice, the runtime is virtually linear.
- For slew constraint equal to $1ns$, we present a log-log (log (number of buffer positions) v.s. log (CPU time)) plot in Figure 12 where the best linear fit to the data points is also shown. The slope of the linear fit is 1.02. Therefore, the runtime of SB almost linearly depends on the number of buffer positions.
- Comparing slack at driver (c.f. slack degradation ratio in Table III), one sees that slew buffering achieves significant improvement in runtime with only slight sacrifice in slack. Note that slack here refers to the sum of slacks over all nets.

It is worth mentioning that the range search tree pruning technique, when incorporated into SB, slows down the algorithm as indicated by our experiment. For example, when the slew constraint is 1.0, SB with range search tree returns the solution in 49.8 seconds compared to 6.2 seconds by the one without it. This fact is due to the considerable amount of inherent overhead in maintaining the balanced range search tree data structure.

It is interesting to investigate the following CWB buffering algorithm. In CWB, the pruning condition is based only on (C, W) but not on Q or S. However, S is still maintained throughout solution propagation for checking whether slew constraint is violated. Compared to VGL+S and SB, CWB should certainly run faster since fewer solutions need to be maintained in solution propagation. It is interested to investigate the solution quality degradation by CWB. The results are summarized in Table IV. Comparing Table IV and Table III, one can see that CWB is worse than VGL+S in area. This makes sense by noting the following facts. It is true that both VGL+S and CWB may prune solutions which are actually superior in slew. However, VGL+S maintains much

TABLE III

COMPARISON OF DISCRETE SLEW BUFFERING (SB) AND SLEW CONSTRAINED TIMING BUFFERING (VGL+S). #S@DR: AVERAGE NUMBER OF NON-DOMINATED SOLUTIONS AT DRIVER. SLACK IS IN ns.

	Discrete Slew Buffering (SB)					Slew Constrained Timing Buffering (VGL+S)					Ratio		
Slew constraint (ns)	Area	# Buf	Slack	#S@Dr	CPU (s)	Area	# Buf	Slack	#S@Dr	CPU (s)	Area Saving	Speed up	Slack Degrad.
0.3	44980	7794	8715	71	19.1	46551	9605	8760	271	346.5	3.5%	18.1	0.5%
0.4	30963	6069	8697	51	15.0	32133	7600	8749	247	351.8	3.8%	23.4	0.6%
0.5	22960	5108	8511	35	11.7	24235	6858	8613	246	408.1	5.6%	34.9	1.2%
0.6	18380	4114	8472	27	9.5	19438	5504	8650	254	417.8	5.8%	43.9	2.1%
0.7	15531	3551	8420	22	8.3	16445	4565	8581	269	463.2	5.8%	55.8	1.9%
0.8	13340	3216	8387	18	7.5	14218	4300	8542	278	487.1	6.6%	65.0	1.8%
0.9	11578	2972	8332	14	6.9	12243	3749	8510	292	532.9	5.7%	77.2	2.1%
1.0	10316	2712	8305	13	6.2	10897	3340	8481	299	548.9	5.6%	88.5	2.1%

more solutions than CWB and there are some correlations between delay Q and slew S, thus, VGL+S should have larger potential to keep those solutions which are superior in slew. As a result, VGL+S outperforms CWB from about 1% to 5% in buffer area.

TABLE IV

SLEW CONSTRAINED BUFFERING WITH PRUNING BASED ON (C, W), CWB. #S@DR: THE NUMBER OF NON-DOMINATED SOLUTIONS AT DRIVER. AREA SAVING IS OBTAINED COMPARING TO SB.

	Discrete Slew Buffering based on (C, W)				Ratio
Slew constraint (ns)	Area	# Buf	#S@Dr	CPU (s)	Area Sav.
0.3	47993	10265	48	16.0	6.7%
0.4	33848	7793	43	14.2	9.3%
0.5	25210	7058	31	10.9	9.8%
0.6	20019	5591	23	8.5	8.9%
0.7	16730	4582	19	7.9	7.7%
0.8	14283	4398	15	7.1	7.1%
0.9	12358	3850	12	6.5	6.7%
1.0	10985	3379	11	5.9	6.5%

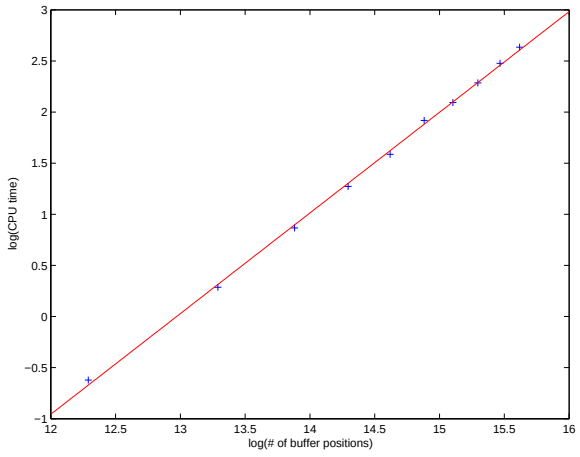

Fig. 12. +: log (number of buffer positions) v.s. log (CPU time) for slew buffering with slew constraint $1.0ns$. Line: best linear fit.

It is known that van Ginneken/Lillis' algorithm is not the most efficient buffering algorithm to handle buffer cost minimization. Several improvements exist. In this paper, we incorporate the pre-buffer slack pruning (PSP) technique proposed in [6] into VGL+S to investigate the performance of SB when compared to one of the state-of-the-art buffering approaches. The results are summarized in Table V. As one can see from Table V, SB runs much faster than VGL+S+PSP.

TABLE V

THE COMPARISON OF SB AND VGL+S+PSP (VGL+S INCORPORATED WITH PRE-BUFFER SLACK PRUNING [6]). SPEED UP REFERS TO THE RUNTIME DIFFERENCE BETWEEN SB AND VGL+S+PSP.

Slew constraint	CPU (s) of VGL+S+PSP	Speed up
0.3	269.3	14.1
0.4	303.0	20.2
0.5	338.1	28.9
0.6	365.7	38.5
0.7	374.3	45.1
0.8	433.5	57.8
0.9	478.2	69.3
1.0	487.9	78.7

To investigate the scalability of the proposed slew buffering algorithm, experiments on a set of 100 large-degree nets are performed. The number of sinks for these testcases ranges from 105 to 948. As before, we compare SB to VGL+S+PSP and the results are summarized in Table VI. One sees that SB can still run much faster than VGL+S+PSP while saving areas.

C. Slew Buffering with Non-Fixed Input Slew

Results of SB+NI are summarized in Table VII. Area saving here refers to comparison to discrete slew buffering with fixed input slew, i.e., SB. We observe the following:

- SB+NI can save up to 21.9% area over SB. In SB+NI, the number of input slew bins to each buffer is 21. For each slew bin, downstream capacitance is also discretized into 21 capacitance bins in the lookup table. With very tight slew constraint, SB+NI saves much more area over SB. It is the case since the actual input slew is significantly smaller than the pre-set upper bound.
- SB+NI becomes slower with tighter slew constraint since the size of the solution set becomes much larger as more buffers are inserted.
- For speedup, we apply the adaptive buffer selection technique as described in Section IV-D for continuous slew buffering to SB+NI. From Table VII, one can see that SB+NI is significantly accelerated while the solution quality is moderately degraded.

TABLE VI

COMPARISON OF DISCRETE SLEW BUFFERING (SB) AND SLEW CONSTRAINED TIMING BUFFERING (VGL+SB+PSP) ON 100 LARGE-DEGREE NETS. SLACK IS IN ns.

	Discrete Slew Buffering (SB)					Slew Constrained Timing Buffering w/ PSP					Ratio		
Slew constraint	Area	# Buf	Slack	#S@Dr	CPU (s)	Area	# Buf	Slack	#S@Dr	CPU (s)	Area Saving	Speed up	Slack Degrad.
0.3	73873	12082	2102	532	373.2	75532	14293	2130	1692	2127.3	2.2%	5.7	1.3%
0.4	61082	11839	2091	438	285.3	62323	13992	2151	1750	2230.0	2.0%	7.8	2.9%
0.5	43992	9870	2130	359	203.9	45340	12840	2172	1779	2292.2	3.1%	11.2	2.0%
0.6	38918	8378	2117	310	149.0	40670	12058	2185	1812	2495.1	4.5%	16.7	3.2%
0.7	32538	7189	2055	257	101.7	34132	11582	2138	1830	2553.5	4.9%	25.1	4.0%
0.8	25127	6032	2050	215	85.5	26615	8172	2125	1855	2628.2	5.9%	30.7	3.6%
0.9	23295	5578	2043	190	77.6	24879	7220	2118	1873	2858.7	6.8%	36.8	3.7%
1.0	20152	4902	2032	157	73.0	21582	6175	2111	1891	3029.2	7.1%	41.5	3.9%

TABLE VII

RESULTS OF SLEW BUFFERING WITH NON-FIXED INPUT SLEW. AREA SAVING IS OBTAINED BY COMPARING TO SB. SLACK DEGRADATION IS OBTAINED BY COMPARING TO VGL+S.

	Discrete Slew Buffering w/ Non-Fixed Input Slew (SB+NI)					SB+NI w/ Adaptive Buffer Selection				
Slew constraint (ns)	Area	# Buf	CPU (s)	Slack Degrad.	Area Saving	Area	# Buf	CPU (s)	Slack Degrad.	Area Saving
0.3	35148	7114	992.1	17.4%	21.9%	37717	7075	19.2	11.7%	16.2%
0.4	25018	5666	931.7	12.3%	19.2%	31661	5000	27.0	6.6%	-2.3%
0.5	19797	4326	762.8	8.0%	13.8%	23841	4175	39.8	1.6%	-3.8%
0.6	16528	3772	569.3	4.6%	10.1%	18921	5406	54.9	1.3%	-2.9%
0.7	13995	3463	473.6	5.1%	9.9%	14941	4958	69.3	2.1%	3.8%
0.8	12129	3145	397.4	4.6%	9.1%	12178	3214	78.2	3.5%	8.7%
0.9	10667	2854	365.2	4.6%	7.9%	10736	2952	81.3	4.3%	7.3%
1.0	9629	2488	337.3	3.9%	6.7%	9663	2525	85.0	2.4%	6.3%

D. Continuous Slew Buffering

Results of C-SB and C-SB+NI are summarized in Table VIII. Area saving here refers to comparison to SB. We observe the following:

- In slew buffering, tighter constraint causes excessive buffer insertion. If the candidate buffer positions are not pre-set carefully in discrete slew buffering, we may often have to insert buffers in an inefficient way. Continuous slew buffering (C-SB) significantly alleviates this problem and results in up to 15% improvement in buffer area.
- C-SB runs very fast due to our adaptive procedure for buffer selection (see Section IV-D). If C-SB is carried out without buffer selection procedure, the algorithm becomes very slow. For example, we obtain a solution with buffer area 9703 in 1722.8 seconds for $\alpha = 1.0$. Compared to C-SB with buffer selection, it is only 0.5% better in buffer area, however, it is about $75\times$ slower.
- We also implement the continuous slew buffering without fixed input slew assumption (C-SB+NI). As is evident from Table VIII, one can further save several percentage area over C-SB while the runtime is still acceptable.

E. Handling Blockage

The experimental results on discrete slew buffering with blockage (SB+B) and the slew constrained timing buffering with the same blockage (VGL+S+B) are included in this paper. Note that SB+B holds the fixed input assumption. We randomly place 20 rectangular blockages with total area summed to 30% that of the smallest bounding box of each net. Results are shown in Table IX. Since blockages are introduced, solution quality of both slew buffering and timing buffering becomes worse than before, i.e., area saving is negative. However, slew buffering still outperforms timing buffering in terms of both runtime and buffer area.

It is interesting to see that timing buffering tends to have more computation overhead than slew buffering when buffer blockages are handled. We would like to interpret this phenomenon as follows. VGL+S is very sensitive to buffer positions in terms of runtime since a new buffer position may lead to many new non-dominated solutions (see Section IV-B.2). However, in slew buffering, a new buffer position can only lead to $|B|$ new solutions as discussed in Section IV-B.2. In fact, the slew buffering algorithm runs almost linear in the number of buffer positions as indicated by Figure 12. Thus, slew buffering is less sensitive to the buffer blockage insertion in terms of runtime.

F. Comparison with Capacitance-Based Buffering

Finally, we compare SB with CBB [12], [11]. As in practice, a typical buffer is selected for running CBB. As such, we calculate for each buffer b_i the longest wire length l_i it can drive such that the slew constraint is satisfied. The typical buffer is the one with maximum $l_i/A(b_i)$ value, where $A(b_i)$ is the buffer area of b_i. The data in Table X (c.f. Table III) demonstrate that SB significantly outperforms CBB in our context: the total area of solutions by CBB is usually more than double that of SB. The area of capacitance based buffering is much worse because only a single buffer is used, capacitance based buffering ignores resistive effect, and multi-pin nets are not well handled in CBB. Note that CBB runs in less time since only a single buffer is used in computation.

TABLE VIII

RESULTS OF CONTINUOUS SLEW BUFFERING. AREA SAVING IS OBTAINED BY COMPARING TO SB. SLACK DEGRADATION IS OBTAINED BY COMPARING TO VGL+S.

	Continuous Slew Buffering (C-SB)					Continuous Slew Buffering w/ Non-Fixed Input Slew (C-SB+NI)				
Slew constraint (ns)	Area	# Buf	CPU (s)	Slack Degrad.	Area Saving	Area	# Buf	CPU (s)	Slack Degrad.	Area Saving
0.3	37840	6383	2.5	9.8%	15.8%	37627	6149	19.7	15.4%	16.4%
0.4	27905	4717	2.7	5.7%	9.9%	24927	4980	285.7	8.8%	19.5%
0.5	21043	4202	9.3	5.9%	8.3%	19281	3851	220.6	7.4%	16.0%
0.6	16880	4735	40.9	6.5%	8.9%	15831	4622	710.5	5.7%	13.9%
0.7	14525	3420	33.1	5.2%	6.9%	13017	3190	700.3	5.2%	16.2%
0.8	12472	3198	29.2	5.9%	6.5%	10813	2886	610.7	6.6%	18.9%
0.9	10872	2883	25.8	5.1%	6.1%	9582	2461	532.8	6.2%	17.2%
1.0	9754	2630	22.9	5.7%	5.4%	8768	2277	442.1	5.5%	15.0%

TABLE IX

HANDLING BLOCKAGE. EACH NET HAS 30% BLOCKAGE AREA. AREA SAVING IS OBTAINED BY COMPARING TO SB.

	Slew Buffering w/ Blockage				Timing Buffering w/ Blockage			
Slew constraint (ns)	Area	#Buf	CPU (s)	Area Saving	Area	#Buf	CPU (s)	Area Saving
0.3	51347	8502	22.3	-12.4%	52121	10792	423.7	-13.7%
0.4	35467	6792	19.3	-12.7%	36214	8302	447.7	-14.5%
0.5	26180	5893	16.2	-12.3%	26635	7693	470.3	-13.8%
0.6	20863	4721	13.2	-11.9%	21201	5887	542.3	-13.3%
0.7	17831	4082	10.0	-12.9%	18380	4952	610.1	-15.5%
0.8	15073	3529	9.2	-11.5%	15731	4598	662.3	-15.2%
0.9	13202	3290	8.7	-12.3%	13767	4110	710.5	-15.9%
1.0	11845	3021	7.9	-12.9%	12237	3775	759.0	-15.7%

TABLE X

CAPACITANCE-BASED BUFFERING (CBB). ONLY A SINGLE TYPICAL BUFFER IS USED. AREA SAVING IS OBTAINED BY COMPARING TO SB.

Slew (ns)	Area	# Buf	CPU (s)	Area Saving
0.3	86572	12357	1.1	-48.0%
0.4	62101	8864	1.3	-50.1%
0.5	54324	7754	1.5	-57.7%
0.6	49825	7112	1.5	-63.1%
0.7	42526	6070	1.4	-63.5%
0.8	36956	5275	1.3	-63.9%
0.9	31611	4512	1.0	-63.4%
1.0	26447	3775	0.9	-61.0%

VII. CONCLUSION

This work proposes a new buffering formulation motivated by the need to efficiently buffer huge numbers of nets under slew constraints. We show that one can optimize for area and satisfy a slew constraint efficiently, despite the problem being NP-hard.

The slew buffering problem is intensively studied in this paper. Three new algorithms are proposed, namely, a slew buffering algorithm with the assumption of fixed input slew, a more sophisticated algorithm without this assumption, and a very efficient continuous slew buffering algorithm. Experimental results demonstrate that new algorithms run one to two orders of magnitude faster than the widely-used timing buffering algorithm and meanwhile they can obtain significant amount of area saving. Future work seeks to incorporate our results into a physical synthesis flow. It is also interesting to design a new slew buffering method which may obtain buffering solutions with quality close to the slew buffering without input slew assumptions and with runtime close to the slew buffering with the fixed input slew assumption.

VIII. ACKNOWLEDGEMENT

The authors are grateful to the anonymous reviewers for their very insightful and helpful comments.

REFERENCES

[1] P. Saxena and N. Menezes and P. Cocchini and D.A. Kirkpatrick, "Repeater scaling and its impact on CAD," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 4, pp. 451–463, 2004.

[2] P.J. Osler, "Placement driven synthesis case studies on two sets of two chips: hierarchical and flat," *in Proceedings of the ACM International Symposium on Physical Design*, pp. 190–197, 2004.

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

[4] J. Lillis and C.-K. Cheng and T.-T.Y. Lin, "Optimal wire sizing and buffer insertion for low power and a generalized delay model," *IEEE Journal of Solid State Circuits*, vol. 31, no. 3, pp. 437–447, 1996.

[5] C.J. Alpert and A. Devgan and S.T. Quay, "Buffer insertion for noise and delay optimization," *in Proceedings of the Design Automation Conference*, pp. 362–367, 1998.

[6] W. Shi and Z. Li and C. Alpert, "Complexity analysis and speedup techniques for optimal buffer insertion with minimum cost," *in Proceedings of the Asia and South Pacific Design Automation Conference*, pp. 609–614, 2004.

[7] W. Shi and Z. Li, "A fast algorithm for optimal buffer insertion," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 6, pp. 879–891, 2005.

[8] Z. Li and W. Shi, "An $O(bn^2)$ time algorithm for optimal buffer insertion with b buffer types," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 3, pp. 484–489, 2006.

[9] N. Menezes and C.-P. Chen, "Spec-based repeater insertion and wire sizing for on-chip interconnect," *in Proceedings of the IEEE International Conference on VLSI Design*, pp. 476–483, 1999.

[10] C.J. Alpert and A. Devgan and S.T. Quay, "Buffer insertion with accurate gate and interconnect delay computation," *in Proceedings of the Design Automation Conference*, pp. 479–484, 1999.

[11] C.J. Alpert and A.B. Kahng and B. Liu and I. Mandoiu and A. Zelikovsky, "Minimum-buffered routing of non-critical nets for slew rate and reliability control," *in Proceedings of International Conference on Computer Aided Design*, pp. 408–415, 2001.

[12] C.J. Alpert and J. Hu and S.S. Sapatnekar and P.G. Villarrubia, "A practical methodology for early buffer and wire resource allocation," *in Proceedings of the Design Automation Conference*, pp. 189–194, 2001.
[13] C.V. Kashyap and C.J. Alpert and F. Liu and A. Devgan, "Closed form expressions for extending step delay and slew metrics to ramp inputs," *in Proceedings of the International Symposium on Physical Design*, pp. 24–31, 2003.
[14] H.B. Bakoglu, *Circuits, Interconnects, and Packaging for VLSI*. Addison-Wesley Publishing Company, 1990.
[15] N.H. Weste and K. Eshraghian, *Principles of CMOS VLSI Design*. Addison Wesley, 1993, pp. 221–223.
[16] EinsTimer Users Guide and Language Reference, *IBM Microelectronics Division, Hopewell Junction, NY*, 1995.
[17] T. Feder and R. Motwani, "Clique partitions, graph compression, and speeding-up algorithms," *in Proceedings of the ACM Symposium on Theory of Computing*, pp. 123–133, 1991.
[18] A.V. Goldberg, "An efficient implementation of a scaling minimum-cost flow algorithm," *Journal of Algorithms*, pp. 1–29, 1997.
[19] J. Hu and C.J. Alpert and S.T. Quay and G. Gandham, "Buffer insertion with adaptive blockage avoidance," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 492–498, 2003.