A Fully Polynomial Time Approximation Scheme for Timing Driven Minimum Cost Buffer Insertion

Shiyan Hu
Dept. of Electrical and Computer Engineering
Michigan Technological University
Houghton, Michigan 49931
shiyan@mtu.edu

Zhuo Li and Charles J. Alpert
IBM Austin Research Laboratory
11501 Burnet Road
Austin, Texas 78758
{lizhuo, alpert}@us.ibm.com

ABSTRACT

As VLSI technology enters the nanoscale regime, interconnect delay has become the bottleneck of the circuit timing. As one of the most powerful techniques for interconnect optimization, buffer insertion is indispensable in the physical synthesis flow. Buffering is known to be NP-complete and existing works either explore dynamic programming to compute optimal solution in the worst-case exponential time or design efficient heuristics without performance guarantee. Even if buffer insertion is one of the most studied problems in physical design, whether there is an efficient algorithm with provably good performance still remains unknown.

This work settles this open problem. In the paper, the first fully polynomial time approximation scheme for the timing driven minimum cost buffer insertion problem is designed. The new algorithm can approximate the optimal buffering solution within a factor of $1+\epsilon$ running in $O(m^2n^2b/\epsilon^3 + n^3b^2/\epsilon)$ time for any $0 < \epsilon < 1$, where n is the number of candidate buffer locations, m is the number of sinks in the tree, and b is the number of buffers in the buffer library. In addition to its theoretical guarantee, our experiments on 1000 industrial nets demonstrate that compared to the commonly-used dynamic programming algorithm, the new algorithm well approximates the optimal solution, with only 0.57% additional buffers and 4.6× speedup. This clearly demonstrates the practical value of the new algorithm.

Categories and Subject Descriptors

B.7.2 [**Integrated Circuits**]: Design Aids - Placement and Routing; J.6 [**Computer-aided Engineering**]: Computer-aided Design

General Terms

Algorithms, Performance, Design

Keywords

Buffer Insertion, Fully Polynomial Time Approximation Scheme, NP-complete, Cost Minimization, Dynamic Programming

DAC'09, July 26-31, 2009, San Francisco, California, USA

1. INTRODUCTION

As VLSI technology enters the nanoscale regime, interconnect delay has become the bottleneck of the circuit timing since devices scale much faster than interconnects. As one of the most effective interconnect timing optimization engines, buffer insertion is indispensable in the physical synthesis flow [1, 2, 3]. It is demonstrated in [4] that in two recent IBM ASIC designs, over one-fourth gates are buffers.

Buffer insertion is one of the most studied problems in physical design. Existing works include [5, 6, 7, 8] on exploring dynamic programming techniques with advanced data structures to compute optimal timing driven buffering solutions. Since buffers themselves are a drain on power, it is highly desirable to use as little buffering resources as possible in buffer insertion. Buffering with cost (power or area) minimization has been considered in [6]. It proposes a dynamic programming algorithm which runs in pseudo-polynomial time. This surprises no one as the minimum cost timing driven buffering problem is NP-complete [9]. Derived from this classic problem, buffering techniques have been developed for various scenarios. For example, there are works [6, 10, 11, 12] handling slew, noise and/or variations, and works exploring the interaction of buffering with routing [13], placement [14], and floorplanning [15, 16].

Despite the fact that many buffering techniques have been developed, the underlying buffering problem is still less studied especially in theory. The dynamic programming can compute the optimal solution but not runs in polynomial time, while heuristic algorithms can run fast but without any performance guarantee. Whether there is any efficient algorithm with provably good performance still remains unknown. This work aims to settle this open problem and advance the understanding of minimum cost timing driven buffering problem from a theoretical point of view. Yet, our new algorithm is highly practical.

In this paper, we propose a fully polynomial time approximation scheme (FPTAS) for the NP-complete timing driven minimum cost buffering problem. In our context, a fully polynomial time approximation scheme refers to a buffering algorithm which is able to compute a solution with the cost at most $1+\epsilon$ times worse than the cost of the optimal buffering solution for any $\epsilon > 0$. It runs in time polynomial in the input size of the problem instance and $1/\epsilon$. Given an NP-complete problem, an FPTAS is generally regarded as an ultimate solution in theory. The main contribution of this paper is summarized as follows.

- An FPTAS algorithm is proposed to approximate the optimal buffering solution within a factor of $1+\epsilon$ in

$O(m^2n^2b/\epsilon^3 + n^3b^2/\epsilon)$ time for any $0 < \epsilon < 1$ and in $O(m^2n^2b/\epsilon + mn^2b + n^3b^2)$ time for any $\epsilon \geq 1$, where n is the number of candidate buffer locations in the tree, m is the number of sinks in the tree, and b is the number of buffers in the buffer library.

- This work presents the first provably good approximation algorithm on the timing-driven minimum cost buffering problem.
- The proposed FPTAS is motivated from the algorithms in [17, 18] for the layer assignment problem. Nevertheless, our FPTAS features novel techniques such as double-ϵ oracle based solution search and timing-cost approximate dynamic programming algorithm. The latter runs in $O(\frac{m^2n}{\epsilon_1\epsilon_2} + \frac{mn^2b}{\epsilon_1} + \frac{m^2n^2}{\epsilon_1^2\epsilon_2} + \frac{mn^2b}{\epsilon_1\epsilon_2} + \frac{n^3b^2}{\epsilon_1})$ time to compute a buffering solution with the cost at most $(1+\epsilon_1)W^*$ and the timing at most $(1+\epsilon_2)T$ where W^* refers to the optimal cost and T refers to the timing constraint.
- The new FPTAS algorithm is highly practical. Experimental results on 1000 industrial nets using a buffer library consisting of 48 buffer types demonstrate that the FPTAS approximates the optimal solution by only 0.57% additional buffers with 4.6× speedup compared to the dynamic programming algorithm.

2. PRELIMINARIES

A routing tree $\mathcal{T} = (V, E)$ is given as an input to the minimum cost timing buffering problem, where $V = \{s_0\} \cup V_s \cup V_n$, and $E \subseteq V \times V$. As in many previous works [9, 11], the routing tree is assumed to be binary in this paper. Trees in other topologies can be easily converted to a binary tree [7]. Vertex s_0 is the root/driver of $\mathcal{T}$, V_s is the set of sink vertices, and V_n is the set of candidate buffer locations. For each sink $s \in V_s$, there is a sink capacitance $C(s)$ and a required arrival time (RAT). The driver s_0 has an arrival time, denoted by $AT(s_0)$. A net satisfies the timing constraint if the arrival time is no greater than the required arrival time at driver. By subtracting $AT(s_0)$ and each sink RAT by $AT(s_0)$, one can modify the net such that $AT(s_0) = 0$ and all RAT are positive. Note that this will not impact buffering solutions. Define the timing constraint T to be the maximum RAT after the above modification. A buffer library B containing all buffer types which can be assigned to candidate buffer locations is also given. Note that B includes both non-inverting buffers and inverting buffers.

As is commonly used in physical synthesis, the Elmore delay model is adopted. The Elmore delay on an edge $e = (v_i, v_j)$ is computed by $D(e) = R(e)\left(\frac{C(e)}{2} + C(v_j)\right)$, where $C(e), R(e), C(v_j)$ refer to the edge capacitance, the edge resistance and the downstream capacitance viewing at v_j, respectively. For a buffer b placed at vertex v_j, its buffer delay is computed by $D(b) = R(b) \cdot C(v_j) + K(b)$, where $R(b)$ and $K(b)$ refer to the driving resistance and the intrinsic delay of buffer b, respectively. Each buffer b has also an input capacitance $C(b)$ and a cost $w(b)$. In this paper, the buffer area is used as buffer cost to illustrate our new algorithm. A buffer assignment γ is a mapping $\gamma : V_n \rightarrow B \cup \{\overline{b}\}$ where $\overline{b}$ denotes the case with no buffer inserted. The total cost of a buffering solution γ for the tree $\mathcal{T}$ is defined as the sum of the costs over all inserted buffers. The timing driven minimum cost buffering problem, known as NP-complete [9], can be formulated as follows.

Timing Constrained Minimum Cost Buffering: Given a binary routing tree with n candidate buffer locations and a buffer library with b buffer types, to compute a buffer assignment solution such that the timing constraint is satisfied and the total buffer cost is minimized.

3. ALGORITHMIC FLOW

Our FPTAS algorithm for timing constrained minimum cost buffering problem is motivated from [17, 18] for a layer assignment problem. Let W^* denote the cost of the optimal buffering solution. At a high level, the FPTAS works in the following intuitive framework. It first makes a guess x on W^* and uses a procedure called *oracle* to check whether such a guess is good, i.e., sufficiently close to W^* or not. If this is the case, return x. Otherwise, make a new guess on W^*. This procedure is iterated until a good guess is made.

Certainly, there are two algorithmic design challenges in the above framework. First, one does not know the optimal cost W^*, which is our target, then how to decide whether $x \geq W^*$ for any x? Moreover, one needs to perform this check efficiently. Second, if the current guess is not good, how to find a possibly better guess? The first difficulty needs to a salient design of the oracle and the second difficulty needs the usage of an efficient oracle based solution search. These two key components will be described in this paper.

4. DOUBLE-ϵ ORACLE SEARCH

4.1 The Oracle

Given any positive number x, the oracle can efficiently decide whether $W^* \geq x$, where W^* is the total buffer cost of the optimal buffering solution. In fact, the decision is answered approximately depending on ϵ, meaning that the answer is either $W^* \geq x$ or $W^* < (1+\epsilon)x$.

A key component in the oracle is a *polynomial time timing-cost approximate dynamic programming* algorithm which will be described in Section 5. This algorithm has three critical properties. First, the algorithm can be performed efficiently, in time polynomial in $\overline{W} = n/\epsilon$. Second, it will either return a buffering solution with the cost no greater than a cost budget value $\overline{W}$, or conclude that this cost budget $\overline{W}$ is too low to find any buffering solution (approximately) satisfying the timing constraint. Third, the dynamic programming algorithm will return a solution with timing slightly larger than the timing constraint T with controlled error. Precisely, the timing of the solution is bounded by $(1+\epsilon)T$ where ϵ is the target approximation ratio. This is acceptable especially considering that in early stage of physical synthesis flow (or when chip is in the prototype stage), the timing constraint is often set according to the designer' experience and thus in general the timing constraint is not stringent. In addition, varying ϵ, our algorithm can provide more design flexility (in terms of timing and cost tradeoff) to the circuit designers. Even in the late physical design stage where timing constraint is stringent, in practice, one can still rip-up and rebuffer the nets with timing violations using [6] to compute the optimal buffering solutions. We call this procedure *timing recovery*. Even if FPTAS with timing recovery is not guaranteed to run in polynomial time, in practice it would still run much faster than using [6] alone. This is the case since there are very few nets which violate the timing constraints by our FPTAS as indicated in the experiments.

Given a timing-cost approximate dynamic programming algorithm, we are ready to present the oracle. First, for any

positive number ϵ, each buffer cost w is scaled by the factor of $\frac{x\epsilon}{n}$ followed by down-rounding, i.e., w becomes $\lfloor \frac{wn}{x\epsilon} \rfloor$. The dynamic programming is performed to the scaled and rounded buffering problem with the cost budget set to n/ϵ. There are two possible decision results in an oracle query.

1. By dynamic programming, a buffering solution with timing no greater than $(1+\epsilon)T$ is found for the total buffer cost $\leq n/\epsilon$. This means that using the unscaled and unrounded costs, the cost of the obtained buffering solution will be smaller than $\frac{n}{\epsilon} \cdot \frac{x\epsilon}{n} + x\epsilon = (1+\epsilon)x$. This is the case since the rounding error in cost is at most $x\epsilon$ by noting that the rounding error is at most $\frac{x\epsilon}{n}$ at each buffer and there are only n candidate buffer locations. Therefore, there is a buffering solution with cost smaller than $(1+\epsilon)x$ and the timing at most $(1+\epsilon)T$ in the original buffering problem. We conclude that $W^* < (1+\epsilon)x$.
2. By dynamic programming, there is no buffering solution with cost $w = n/\epsilon$ which can satisfy even the relaxed timing constraint $(1+\epsilon)T$. This means that the original unscaled buffering problem does not have a buffering solution within the cost $\frac{n}{\epsilon} \cdot \frac{x\epsilon}{n} = x$ satisfying $(1+\epsilon)T$ and thus T. We conclude that $W^* \geq x$ in the original buffering problem.

Since both timing and cost are rounded, our FPTAS actually computes the $(1+\epsilon)$ approximation to the buffering problem with the longest path delay bounded by $(1+\epsilon)T$. According to Lemma 2 in Section 5, the proposed dynamic programming algorithm runs in $O(\frac{m^2n}{\epsilon_1\epsilon_2} + \frac{mn^2b}{\epsilon_1} + \frac{m^2n^2}{\epsilon_1^2\epsilon_2} + \frac{mn^2b}{\epsilon_1\epsilon_2} + \frac{n^3b^2}{\epsilon_1})$ time which gives the time for an oracle query.

4.2 The Double-ϵ Oracle Based Solution Search

After obtaining the oracle, a $(1+\epsilon)$ approximation could be computed as follows. Start with an initial lower bound W_l^* and an initial upper bound W_u^* on the optimal buffering cost W^*. For example, W_u^* could correspond to always using largest buffer at every candidate buffer location and W_l^* could be set to the cost of the single buffer with smallest cost in the buffer library. One just needs to compare the timing of the given unbuffered net to the timing constraint to account for the case where no buffer needs to be inserted. Subsequently, a binary search is performed within these bounds. That is, each time $x = \frac{W_l^* + W_u^*}{2}$ is used to query the oracle and depending on the decision results, W_l^* and W_u^* will be updated accordingly, i.e., update $W_u^* = (1+\epsilon)x$ in Case 1 and update $W_l^* = x$ in Case 2. The complexity of this algorithm certainly depends on the gap between the initial upper and lower bounds.

One can make the time complexity independent of the initial bound values as in [18]. This is accomplished by utilizing the fact that an oracle query takes the time inversely proportional to ϵ. Precisely, a larger ϵ leads to a coarser approximation but runs faster while a smaller ϵ leads to finer approximation but runs slower. This provides the opportunity in varying approximation ratio ϵ adaptively to accelerate the whole procedure of oracle based solution search. Instead of sticking to the target approximation ratio ϵ, one can use larger ϵ initially and gradually reduce it to the target ϵ. When these ϵ form a decreasing geometric sequence (e.g., $\ldots, 27, 9, 3, 1, 1/3, \ldots, \epsilon$), the total asymptotic runtime will be bounded by the last query since the an oracle query takes the time proportional to $1/\epsilon$.

Our oracle based solution search is similar to the one in [18] with an important difference as follows. Since their algorithm only rounds one parameter while our approach rounds two parameters (cost W and timing Q), applying the technique in [18] leads to adaptively setting both rounding parameters. This is not desired for rounding Q since the timing error will not be controlled by ϵ. That is, in first few iterations during oracle based search, the timing of the solutions may be significantly larger than $(1+\epsilon)T$ since approximation ratios there would be big (e.g., the first few approximation ratios in $\ldots, 27, 9, 3, 1, 1/3, \ldots, \epsilon$ are much bigger than the target ϵ). This means that by dynamic programming, one only knows whether the cost budget n/ϵ is sufficient for computing a solution with much larger timing, which may lead to the wrong decision in narrowing down the gap between upper and lower bounds. It motivates us to propose to only change the ϵ corresponding to cost but not the ϵ corresponding to timing. We call it *double-ϵ oracle based solution search*. This is why there are two ϵ_1 and ϵ_2 in the dynamic programming algorithm in Section 5. Namely, ϵ_1 corresponds to the approximation ratio on cost W and ϵ_2 corresponds to the approximation ratio on timing Q. If we fix both of them to ϵ, one cannot reduce the total runtime to be independent of the initial bounds. Our idea is to fix ϵ_2 at ϵ while adaptively changing ϵ_1.

In details, let $W_{u,i}^*$ denote the upper bound and $W_{l,i}^*$ denote the lower bound after i-th iteration in the oracle based solution search. Initially, $W_{u,0}^* = W_u^*$ and $W_{l,0}^* = W_l^*$. A geometric sequence of approximation ratios ϵ_i' converging to ϵ will be used in oracle based solution search for ϵ_1 (but not ϵ_2 since it is fixed at ϵ). Following [18], set

$$\epsilon_i' = \sqrt{\frac{W_{u,i}^*}{W_{l,i}^*}} - 1. \tag{1}$$

In each iteration,

$$x = \sqrt{\frac{W_{u,i}^* \cdot W_{l,i}^*}{1+\epsilon_i'}} \tag{2}$$

is used to query the oracle. It can be proved that after i-th oracle query,

$$\frac{W_{u,i+1}^*}{W_{l,i+1}^*} = \left(\frac{W_{u,i}^*}{W_{l,i}^*}\right)^{3/4}, \tag{3}$$

since for Case 1 $W_{u,i+1}^* = (1+\epsilon_i')x = {W_{u,i}^*}^{3/4} {W_{l,i}^*}^{1/4}$, $W_{l,i+1}^* = W_{l,i}^*$, and for Case 2 $W_{u,i+1}^* = W_{u,i}^*$, $W_{l,i+1}^* = x = {W_{u,i}^*}^{1/4} {W_{l,i}^*}^{3/4}$. This process is iterated until the ratio between upper and lower bound is no greater than 2. Let i_θ be the first i such that $\frac{W_{u,i}^*}{W_{l,i}^*} \leq 2$. According to Lemma 2 in Section 5, an oracle query takes $O(\frac{m^2n}{\epsilon_1\epsilon_2} + \frac{mn^2b}{\epsilon_1} + \frac{m^2n^2}{\epsilon_1^2\epsilon_2} + \frac{mn^2b}{\epsilon_1\epsilon_2} + \frac{n^3b^2}{\epsilon_1})$ time. We first bound the time until i_θ, which is $O(\frac{m^2n}{\epsilon_2} \sum_{1\leq i\leq i_\theta} \frac{1}{\epsilon_i'} + mn^2b \sum_{1\leq i\leq i_\theta} \frac{1}{\epsilon_i'} + \frac{m^2n^2}{\epsilon_2} \sum_{1\leq i\leq i_\theta} \frac{1}{\epsilon_i'^2} + \frac{mn^2b}{\epsilon_2} \sum_{1\leq i\leq i_\theta} \frac{1}{\epsilon_i'} + n^3b^2 \sum_{1\leq i\leq i_\theta} \frac{1}{\epsilon_i'})$. Since $i \leq i_\theta$ and $\frac{W_{u,i}^*}{W_{l,i}^*} > 2$, $\frac{1}{\epsilon_i'} = \frac{1}{\sqrt{\frac{W_{u,i}^*}{W_{l,i}^*}}-1}$ means $\frac{1}{\epsilon_i'^2} \leq (2+\sqrt{2})^2 \cdot \frac{W_{l,i}^*}{W_{u,i}^*}$. Thus,

$$O(\frac{m^2n^2}{\epsilon_2} \sum_{1\leq i\leq i_\theta} \frac{1}{\epsilon_i'^2}) = O(\frac{m^2n^2}{\epsilon_2} \sum_{1\leq i\leq i_\theta} \frac{W_{l,i}^*}{W_{u,i}^*}). \tag{4}$$

It is shown in [18] that

$$\sum_{1\le i\le i_\theta}\frac{W^*_{l,i}}{W^*_{u,i}}=\sum_{1\le i\le i_\theta}(\frac{W^*_{l,i_\theta}}{W^*_{u,i_\theta}})^{(\frac{4}{3})^{i_\theta-i}}=\sum_{0\le j<i_\theta}(\frac{W^*_{l,i_\theta}}{W^*_{u,i_\theta}})^{(\frac{4}{3})^{j}}, \quad (5)$$

(note that j starts from 0), and $\frac{W^*_{l,t}}{W^*_{u,t}}<\frac{1}{2^{3/4}}$ by noting that i_θ is the first i such that $\frac{W^*_{u,i}}{W^*_{l,i}}\le 2$. Subsequently,

$$\sum_{1\le i\le i_\theta}\frac{W^*_{l,i}}{W^*_{u,i}}=\sum_{0\le j<i_\theta}(\frac{W^*_{l,i_\theta}}{W^*_{u,i_\theta}})^{(4/3)^j}<\sum_{0\le j<i_\theta}\frac{1}{2^{3/4}}^{(4/3)^j}. \quad (6)$$

The last term is the sum of a monotonically decreasing geometric sequence which is certainly bounded by $O(1)$. Thus,

$$O(\frac{m^2n^2}{\epsilon_2}\sum_{1\le i\le i_\theta}\frac{1}{\epsilon_i'^2})=O(\frac{m^2n^2}{\epsilon_2}). \quad (7)$$

Similarly, since $\sum_{1\le i\le i_\theta}\sqrt{\frac{W^*_{l,i}}{W^*_{u,i}}}<\sum_{0\le j<i_\theta}0.59^{1/2\cdot(4/3)^j}=O(1)$ [18], we have $O(\frac{mn^2b}{\epsilon_2}\sum_{1\le i\le i_\theta}\frac{1}{\epsilon_i'})=O(\frac{mn^2b}{\epsilon_2})$ and $O(n^3b^2\sum_{1\le i\le i_\theta}\frac{1}{\epsilon_i'})=O(n^3b^2)$. The total runtime for oracle based solution search is bounded by

$$O(\frac{m^2n^2b}{\epsilon}+mn^2b+n^3b^2), \quad (8)$$

since ϵ_2 is fixed at ϵ.

After i_θ oracle queries, the ratio between upper and lower bounds is at most 2. With this better starting point, the timing-cost approximate dynamic programming can be efficiently performed with both ϵ_1 and ϵ_2 set to ϵ. First set x to the current lower bound W^*_{l,i_θ}. Subsequently, scale and round the cost w of each buffer to $\lfloor\frac{wn}{x\epsilon}\rfloor$ where ϵ is the target ϵ. Note that W^* is no greater than $W^*_{u,i_\theta}\le 2W^*_{l,i_\theta}$, which means that there is at least one solution with scaled cost no greater than $2n/\epsilon$. Otherwise, similar to Case 2, there is no buffering solution within cost $\frac{2n}{\epsilon}\cdot\frac{x\epsilon}{n}=2x\ge W^*_{u,i_\theta}$ which is a contradiction. This solution is the $(1+\epsilon)$ approximation which will be returned by our FPTAS. Similar to the argument in Case (1), scaling the result back by the factor of $\frac{x\epsilon}{n}$ forms a lower bound on W^* and the maximum rounding error is $\frac{x\epsilon}{n}\cdot n=x\epsilon=W^*_{l,t}\epsilon\le W^*\epsilon$. Thus, the cost of the obtained buffering solution is at most $(1+\epsilon)W^*$. This last step uses the dynamic programming which takes $O(\frac{m^2n}{\epsilon^2}+\frac{mn^2b}{\epsilon}+\frac{m^2n^2}{\epsilon^3}+\frac{mn^2b}{\epsilon^2}+\frac{n^3b^2}{\epsilon})$ time by noting that both ϵ_1 and ϵ_2 are set to ϵ. The efficient computation is due to the fact that the ratio between the current upper and lower bounds has been reduced to ≤ 2. Together with the time for oracle based solution search in Eqn. (8), the total time is $O(\frac{m^2n}{\epsilon^2}+\frac{mn^2b}{\epsilon}+\frac{m^2n^2}{\epsilon^3}+\frac{mn^2b}{\epsilon^2}+\frac{n^3b^2}{\epsilon}+\frac{m^2n^2b}{\epsilon}+mn^2b+n^3b^2)$ which can be simplified to $O(m^2n^2b/\epsilon^3+n^3b^2/\epsilon)$ for $0<\epsilon<1$ and to $O(m^2n^2b/\epsilon+mn^2b+n^3b^2)$ for $\epsilon\ge 1$. We reach the following theorem.

Theorem 1: A $(1+\epsilon)$ approximation to the timing constrained minimum cost buffering problem can be computed in $O(m^2n^2b/\epsilon^3+n^3b^2/\epsilon)$ time for any $0<\epsilon<1$ and in $O(m^2n^2b/\epsilon+mn^2b+n^3b^2)$ time for $\epsilon\ge 1$, where n is the number of nodes in the tree, m is the number of sinks in the tree, and b is the number of buffers in the buffer library.

5. POLYNOMIAL TIME TIMING-COST APPROXIMATE DYNAMIC PROGRAMMING

5.1 Bound Distinct Cost W and RAT Q

A careful investigation in Lillis' algorithm [6] would reveal that the number of solutions is not polynomially bounded which is why Lillis' algorithm is a pseudo-polynomial algorithm. To design an efficient algorithm, we certainly need to bound the number of solutions during solution propagation. A major innovation in the proposed FPTAS algorithm is a dynamic programming algorithm with polynomially bounded W and Q. As a result, the number of solutions will also be polynomially bounded (since there is only one possible non-dominated solution with each pair of W and Q, namely, the one with the smallest C). First note that we have two ϵ in the algorithm, one being ϵ_1 which refers to the approximation in cost and the other being ϵ_2 which refers to the approximation in timing. ϵ_1 is varying while ϵ_2 is fixed to ϵ in the fast double-ϵ oracle based solution search.

To bound W, recall that in Section 4, one first scales and rounds each buffer cost w to an *integer* as $w=\lfloor\frac{wn}{x\epsilon_1}\rfloor$. After that, the oracle only wants to know whether there is a solution (approximately) satisfying the timing constraint with cost up to n/ϵ_1. Let $\overline{W}=n/\epsilon_1$. The oracle uses this cost bound to perform the dynamic programming. Thus, whenever there is a solution with cost greater than $\overline{W}$, it will be eliminated from the solution set. Consequently, there are at most $\overline{W}+1$ distinct W $(0,1,\ldots,\overline{W})$ at any location during solution propagation.

Our new dynamic programming algorithm is as follows. First, it always works with cost bins (W-bin) since the oracle scales and rounds each buffer cost before performing the dynamic programming algorithm. In dynamic programming, *right before a branch merge* , all the solutions are also discretized into timing bins (Q-bin) and then the solution pruning is performed. This allows us to bound Q as well. Consequently, the number of non-dominated solutions during solution propagation is bounded.

To bound the number of distinct Q, right before each branch merge, for all $Q\ge 0$, round up Q of each branch to the nearest value in $\{0,\epsilon_2T/m,2\epsilon_2T/m,...,T\}$, where m is the number of the sinks. We underestimate the delay by rounding. For example, when $\epsilon_2=0.5$ and $m=2$, $Q=0.7T$ and will be up-rounded to $3\epsilon_2T/m$. The solutions with $Q<0$ will be pruned since the arrival time at driver is 0. Thus, there are at most $\frac{T}{\epsilon_2T/m}+1=m/\epsilon_2+1$ distinct Q after rounding. Together with the fact that there are at most $O(\overline{W})$ distinct W at any point, at most $O(\overline{W}m/\epsilon_2)=O(\frac{mn}{\epsilon_1\epsilon_2})$ non-dominated solutions can be obtained after any branch merge (the number of non-dominated solutions before branch merge will be discussed soon). This is due to the fact that there is only one solution for each pair of Q,W, namely, the one with minimum C. Note that the rounding error in timing at each branching point is at most ϵ_2T/m and thus at most ϵ_2T for the whole tree with $m-1$ branching points. Note that the timing of the obtained solution (in the scaled problem) is at most T but it is with the rounded Q. This means that after rounding Q back, we obtain a buffering solution with timing at most $(1+\epsilon_2)T$ for the original buffering problem.

The time complexity analysis is as follows. After performing a branch merge, there are at most $O(\frac{mn}{\epsilon_1\epsilon_2})$ non-dominated solutions. After that, solutions are propagated

in its upstream branch. An add wire operation does not introduce new solution. After performing buffer insertion operation at a node v, for solutions with new buffers inserted at v, there are only b distinct C (where b is the number of buffer types). Since the number of W is always bounded by $O(\overline{W})$, there are at most $O(\overline{W}b) = O(nb/\epsilon_1)$ non-dominated buffered solutions. This is due to the fact that there is only one solution for each pair of C, W, i.e., the one with maximum Q.

We are to bound the time for computing these $O(\overline{W}b) = O(nb/\epsilon_1)$ non-dominated buffered solutions. Given a solution, buffer insertion at v leads to b possible new solutions. Since there are at most $O(mn/(\epsilon_1\epsilon_2) + n^2b/\epsilon_1)$ non-dominated solutions anywhere (see below), computing non-dominated buffered solutions at node v takes $O(mnb/(\epsilon_1\epsilon_2)+ n^2b^2/\epsilon_1)$ time. This is due to that within a single W bin, (Q, C) based pruning takes linear time in the number of generated solutions which is the same as the pruning without considering W in [5]. Note that cross W-bin pruning is not performed since this will not improve the asymptotic complexity. Together with those $O(\overline{W}m/\epsilon_2)$ unbuffered solutions (which are propagated by add wire from the last branch merge), there are at most $O(\overline{W}m/\epsilon_2 + \overline{W}b)$ non-dominated solutions. When these solutions are propagated along this branch, there are at most $O(\overline{W}m/\epsilon_2 + n\overline{W}b) = O(mn/(\epsilon_1\epsilon_2)+n^2b/\epsilon_1)$ non-dominated solutions at any point before next branch merge since there are at most n candidate buffer locations and Q is not rounded until the branch merge.

Before branch merge, all solutions are first put into cost bins (W-bin) and timing bins (Q-bin). That is, they are placed into the bins with integer costs from 0 to n/ϵ_1. In each cost bin, there are m/ϵ_2+1 timing bins with the timing as $0, \epsilon_2T/m, 2\epsilon_2T/m, ..., T$ by up-rounding each Q. By a linear traversal of all bins, dominated solutions will be pruned. The whole process certainly takes time linear in the number of solutions, i.e., $O(mn/(\epsilon_1\epsilon_2) + n^2b/\epsilon_1)$ time. In solution pruning, since W is always an integer, there are $W + 1$ possible merging results of left and right branch solutions for each W. For example, to obtain merged cost $W = 3$, the four possibly combinations of left-branch solution γ_1 and right-branch solution γ_2 are (1) $W(\gamma_1) = 0, W(\gamma_2) = 3$, (2) $W(\gamma_1) = 1, W(\gamma_2) = 2$, (3) $W(\gamma_1) = 2, W(\gamma_2) = 1$, and (4) $W(\gamma_1) = 3, W(\gamma_2) = 0$. For a fixed combination on W, there are a list of solutions with distinct Q along each branch. For each Q-bin, the time complexity can be easily bounded since all the solutions in each branch are non-dominated and thus for the same W, C are increasingly sorted. One just needs to correspondingly merge two solutions with the same Q.

Table 1: An example for branch merge.

Left branch (Q, W, C)				
C	$W = 0$	$W = 1$	$W = 2 \leftarrow$	$W = 3$
$Q = 0$	10	7	5	2
$Q = \epsilon_2T/m$	20	17	15	12
$Q = 2\epsilon_2T/m$	50	45	40	35
Right branch (Q, W, C)				
C	$W = 0$	$W = 1 \leftarrow$	$W = 2$	$W = 3$
$Q = 0$	15	12	10	5
$Q = \epsilon_2T/m$	25	22	15	10
$Q = 2\epsilon_2T/m$	55	50	45	20

It is helpful to look at an example to illustrate the above analysis. Refer to Table 1. To obtain the merged cost of 3, suppose that we are merging solutions with $W = 2$ in the left branch and solutions with $W = 1$ in the right branch. The corresponding W are shown with arrows. For $Q = 0$ after branch merge, the minimum C is $C = 5 + 12 = 17$. For $Q = \epsilon_2T/m$ after branch merge, the minimum C is $C = 15 + 22 = 37$. For $Q = 2\epsilon_2T/m$ after branch merge, the minimum C is $C = 40 + 50 = 90$. One then also needs to consider the other three merging possibilities for the merged cost to be 3, i.e., (1) $W(\gamma_1) = 0, W(\gamma_2) = 3$, (2) $W(\gamma_1) = 1, W(\gamma_2) = 2$, and (4) $W(\gamma_1) = 3, W(\gamma_2) = 0$. Subsequently, the solution with the minimum C for each pair of W, Q is picked. This process takes $O(m/\epsilon_2 \cdot W)$ time for each merged W since there are only $O(m/\epsilon_2)$ distinct Q. Summing over all W till $\overline{W}$, the time complexity for branch merge and solution pruning in branch merge is $O(m/\epsilon_2 \cdot (\overline{W})^2) = O(mn^2/(\epsilon_1^2\epsilon_2))$. Together with the time for putting the solutions into bins before branch merge, the total runtime is $O(mn/(\epsilon_1\epsilon_2) + n^2b/\epsilon_1 + mn^2/(\epsilon_1^2\epsilon_2))$ for a single branch merge.

5.2 Time Complexity

As mentioned above, a single buffer insertion together with pruning takes $O(mnb/(\epsilon_1\epsilon_2) + n^2b^2/\epsilon_1)$ time. Summing over n candidate buffer locations, total buffer insertion takes $O(mn^2b/(\epsilon_1\epsilon_2) + n^3b^2/\epsilon_1)$ time. This certainly upper bounds the time for add wire. A single branch merge takes $O(mn/(\epsilon_1\epsilon_2) + n^2b/\epsilon_1 + mn^2/(\epsilon_1^2\epsilon_2))$ time. Summing over all $m - 1$ branch merges, $O(m^2n/(\epsilon_1\epsilon_2) + mn^2b/\epsilon_1 + m^2n^2/(\epsilon_1^2\epsilon_2))$ time is needed. Thus, the dynamic programming will run in

$$O(\frac{m^2n}{\epsilon_1\epsilon_2} + \frac{mn^2b}{\epsilon_1} + \frac{m^2n^2}{\epsilon_1^2\epsilon_2} + \frac{mn^2b}{\epsilon_1\epsilon_2} + \frac{n^3b^2}{\epsilon_1}) \quad (9)$$

time to compute a solution with the cost at most $(1 + \epsilon_1)$ optimal cost and with timing at most $(1 + \epsilon_2)T$. We reach the following lemma.

Lemma 2: The timing-cost approximate dynamic programming algorithm can compute a timing driven buffering solution with the cost at most $(1+\epsilon_1)$ optimal cost and with timing at most $(1+\epsilon_2)T$ in $O(\frac{m^2n}{\epsilon_1\epsilon_2} + \frac{mn^2b}{\epsilon_1} + \frac{m^2n^2}{\epsilon_1^2\epsilon_2} + \frac{mn^2b}{\epsilon_1\epsilon_2} + \frac{n^3b^2}{\epsilon_1})$ time for any $\epsilon_1, \epsilon_2 > 0$, where n is the number of candidate buffer locations, m is the number of sinks, and b is the number of buffers in the buffer library.

6. EXPERIMENTAL RESULTS

We compare the proposed FPTAS for the timing driven minimum cost buffering problem to the dynamic programming algorithm [6] which computes optimal buffering solution. The experiments are performed on a set of 1000 nets at various scales extracted from an industrial ASIC chip. The buffer library consists of 48 buffer types including buffers and inverters. The buffer cost is measured by buffer area in this paper. However, other metric can be easily handled in FPTAS.

Refer to Table 2 for the comparison. Cost Ratio and Speedup are computed by comparing to the total buffer cost and the runtime of dynamic programming algorithm. # Vio. specifies the number of nets with timing violations. The results with small $\epsilon < 1$ are shown. This range of ϵ is desired in practice since one always wishes to compute solutions close to the optima. We make the following observations.

- The dynamic programming in [6] computes the optimal solution. Total buffer cost is 3304.09, no net has timing violation, and CPU time is 625.2 seconds.

Table 2: Comparison of the dynamic programming [6] and the FPTAS algorithm on 1000 industrial nets. In dynamic programming solution, total buffer cost is 3304.09, and CPU is 625.2 seconds. # Vio. specifies the number of nets with timing violations. Cost Ratio and Speedup are computed by comparing to dynamic programming.

FPTAS					
ϵ	# Vio.	Total Cost	CPU(s)	Cost Ratio	Speedup
0.01	3	3322.91	135.5	0.57%	4.6×
0.05	9	3390.64	127.6	2.6%	4.9×
0.10	12	3499.78	122.1	5.9%	5.1×
0.20	30	3541.17	117.2	7.2%	5.3×
0.30	53	3578.80	114.5	8.3%	5.5×
0.40	51	3669.12	112.9	11.1%	5.5×
0.50	53	3766.96	110.5	14.0%	5.7×
Average					5.2×

Table 3: The obtained timing on the nets violating the timing constraints for FPTAS with $\epsilon = 0.01$.

FPTAS with $\epsilon = 0.01$			
Net	Timing Constraint	Actual Delay	Timing Violation
1	18.043	18.2140	0.95%
2	4.239	4.2504	0.27%
3	4.054	4.0555	0.04%

- Our FPTAS works very well in practice. Compared to the dynamic programming solutions, there are only slight solution degradations in total buffer costs while on average over 5× speedup is obtained. For example, when the target approximation ratio is set to $\epsilon = 0.01$, the actual approximation ratio (cost ratio) is only 0.57% while 4.6× speedup is achieved. The speedup is so significant since the total number of solutions at driver over 1000 nets is 166,676 for FPTAS with $\epsilon = 0.01$ (for all iterations in performing double-ϵ oracle based solution search) while it is 1,221,604 in the dynamic programming algorithm [6]. It is important to note that the cost ratio is theoretically guaranteed to be no greater than ϵ. In practice, it is much smaller as is shown in Table 2. This clearly demonstrates the effectiveness of our FPTAS algorithm.
- Larger ϵ leads to more speedup while smaller ϵ leads to less solution quality degradation. This is again as guaranteed theoretically.
- Since timing is rounded in our timing-cost approximate dynamic programming, there are timing violations in the obtained buffering solutions. However, it is clear that this happens with very small probability in practice as indicated by our experimental results. When $\epsilon = 0.01$, only 3 out of 1000 nets have timing violations. In addition, the obtained timing is theoretically guaranteed to be within $(1+\epsilon)T$ where T is the timing constraint. For example, the nets with timing violations for FPTAS with $\epsilon = 0.01$ are shown in Table 3. Their actual delays are clearly bounded by the $(1+\epsilon)T = 1.01T$.
- When the timing constraints are stringent, one may need every net to satisfy the timing constraint. For this, the following timing recovery procedure could be performed. Those nets with timing violations can be ripped up and rebuffered using optimal dynamic programming [6]. The overall runtime would still be much better than [6] since few nets need to be rebuffered. For the nets without timing violations, the approximation ratio is bounded by $1+\epsilon$. For the nets with timing violations, the optimal solutions are computed in rebuffering. Thus, the approximation ratio is still bounded by $1+\epsilon$ after timing recovery. Refer to Table 4 for the results. The cost is increased compared to FPTAS without timing recovery since rounding on timing in FPTAS underestimates delay and may make the cost of the obtained buffering solution smaller than the optimal cost (esp. for many of the nets with timing violations) even if rounding on cost increases it. Empirically, FPTAS with $\epsilon = 0.01$ gives the best performance since it has fewest nets which need rebuffering.

Table 4: The results of FPTAS with timing recovery.

FPTAS with Timing Recovery					
ϵ	# Vio.	Total Cost	CPU(s)	Cost Ratio	Speedup
0.01	0	3331.57	143.2	0.83%	4.4×
0.05	0	3415.21	157.7	3.4%	4.0×
0.10	0	3592.76	209.5	8.7%	3.0×
0.20	0	3654.91	223.0	10.6%	2.8×
0.30	0	3702.88	262.2	12.1%	2.4×
0.40	0	3867.12	261.8	17.0%	2.4×
0.50	0	4028.25	265.7	21.9%	2.4×
Average					3.0×

7. REFERENCES

[1] P. Saxena and N. Menezes and P. Cocchini and D.A. Kirkpatrick, "Repeater scaling and its impact on CAD," *TCAD*, vol. 23, no. 4, pp. 451–463, 2004.

[2] J. Cong, "An interconnect-centric design flow for nanometer technologies," *Proceedings of the IEEE*, vol. 89, no. 4, pp. 505–528, 2001.

[3] Z. Li, C. Alpert, S. Hu, T. Muhmud, S. Quay, and P. Villarrubia, "Fast interconnect synthesis with layer assignment," *ISPD*, 2008.

[4] P.J. Osler, "Placement driven synthesis case studies on two sets of two chips: hierarchical and flat," *ISPD*, pp. 190–197, 2004.

[5] 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.

[6] 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.

[7] W. Shi and Z. Li, "A fast algorithm for optimal buffer insertion," *TCAD*, vol. 24, no. 6, pp. 879–891, 2005.

[8] R. Chen and H. Zhou, "A flexible data structure for efficient buffer insertion," *ICCD*, 2004.

[9] W. Shi and Z. Li and C. Alpert, "Complexity analysis and speedup techniques for optimal buffer insertion with minimum cost," *ASPDAC*, pp. 609–614, 2004.

[10] C.J. Alpert and A. Devgan and S.T. Quay, "Buffer insertion for noise and delay optimization," *DAC*, pp. 362–367, 1998.

[11] S. Hu, C. J. Alpert, J. Hu, S. Karandikar, Z. Li, W. Shi, and C. N. Sze, "Fast algorithms for slew constrained minimum cost buffering," *DAC*, 2006.

[12] R. Chen and H. Zhou, "Fast min-cost buffer insertion under process variations," *DAC*, 2007.

[13] H. Zhou, D.F. Wong, I.-M. Liu, and A. Aziz, "Simultaneous routing and buffer insertion with restrictions on buffer locations," *DAC*, 1999.

[14] T.-C. Chen, A. Chakraborty, and D. Z. Pan, "An integrated nonlinear placement framework with congestion and porosity aware buffer planning," *DAC*, 2008.

[15] J. Cong, T. Kong, and D. Z. Pan, "Buffer block planning for interconnect-driven floorplanning," *ICCAD*, 1999.

[16] C.J. Alpert, J. Hu, S.S. Sapatnekar and P. Villarrubia, "A practical methodology for early buffer and wire resource allocation," *DAC*, 2001.

[17] S. Hu, Z. Li, and C.J. Alpert, "A polynomial time approximation scheme for timing constrained minimum cost layer assignment," *ICCAD*, 2008.

[18] S. Hu, Z. Li, and C.J. Alpert, "A faster approximation scheme for timing constrained minimum cost layer assignment," *ISPD*, 2009.