A Faster Approximation Scheme For Timing Driven Minimum Cost Layer Assignment

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
{alpert, lizhuo}@us.ibm.com

ABSTRACT

As VLSI technology moves to the $65nm$ node and beyond, interconnect delay greatly limits the circuit performance. As a critical component in interconnect synthesis, layer assignment manifests enormous potential in drastically reducing wire delay. This is due the fact that wires on thick metals are much less resistive than those on thin metals. Nevertheless, it is not desired to assign all wires to thick metals and the right strategy is to only use minimal thick-metal routing resources for meeting the timing constraints. This timing driven minimum cost layer assignment problem is NP-Complete, and a fast algorithm with provable approximation bound is highly desired.

In this paper, a new fully polynomial time approximation scheme is proposed. It is based on a linear-time dynamic programming algorithm for bounded-cost layer assignment and efficient oracle queries. The proposed algorithm can approximate the optimal layer assignment solution in $O(mn^2/\epsilon)$ time within a factor of $1+\epsilon$ for any $\epsilon > 0$, where n is the tree size and m is the number of routing layers. This significantly improves the previous work. The new algorithm is also highly practical. Our experiments on industrial netlists demonstrate that the new algorithm runs up to 6.5× faster than the optimal dynamic programming with few percent additional wire as guaranteed theoretically. This gives another $> 2\times$ speedup over the previous work.

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

Layer Assignment, Fully Polynomial Time Approximation Scheme, NP-Complete, Oracle, Dynamic Programming

ISPD'09, March 29–April 1, 2009, San Diego, California, USA.

1. INTRODUCTION

As VLSI technology moves to the $65nm$ and $45nm$ nodes, interconnect delay greatly limits the circuit performance due to the fact that devices scale much faster than interconnects. As a consequence, interconnect synthesis becomes a critical tool for achieving timing closure [2]. Early works [3, 4, 5, 6, 7, 8] on interconnect synthesis are only focused on simultaneous buffering and wire sizing while none of them considers layer assignment. The most critical difference between wire sizing and layer assignment is whether wire capacitance and resistance have to be scaled correspondingly.

In wire sizing, up-sizing a wire width by $k\times$ leads to the resistance reduction by $k\times$ and the capacitance increase by (almost) $k\times$ which can only result in limited delay reduction. In addition, up-sizing the wire width almost necessarily hurts the routability. Thus, wire sizing is not frequently used in practical physical synthesis to close timing [2]. In contrast, in layer assignment, resistance in thick metals (in higher layer) dramatically decreases compared to thin metals (in lower layer), while wire capacitance only slightly increases [2]. As a result, the delay in thick metals is much smaller than the delay in thin metals.

Layer assignment provides enormous potential in drastically reducing interconnect delay. More routing layers are available in advanced technology. For example, $65nm$ technologies can have eight metal layers and $45nm$ technologies can have ten metal layers. To take the advantage of this, the physical synthesis tool must perform layer assignment, in particular, on timing critical nets of long length. Further, because the resources for thick metal already exist, the impact to routability by layer assignment is much less significant than that by wire sizing. Nevertheless, one still needs to be conservative on bumping wires up to thick metals since there needs to be some reserved thick-metal wiring resources for the future layer assignments in the interconnect synthesis flow [1, 2]. In addition, as the router will make the final decision on whether to bump up wires to thick metal, assigning too many wires to thick metals may increase the chance for the router to assign thick wires to thin metals. As a result, timing could be severely hurt, which is called "timing surprise" in [1].

Despite the remarkable effect on timing improvement, quite few previous works are focused on layer assignment. The work [2] presents several heuristics for layer assignment in timing closure. Based on it, the work [1] formulates the layer assignment problem as using minimum amount of thick-metal resources to meet a timing target for a buffered routing tree, where except wire layer, neither buffer size nor buffer

location can be changed. This formulation is quite useful in the late stage of a physical synthesis flow such as Placement Driven Synthesis (PDS) flow [10]. At late stage, adding significant ECO change through e.g., rebuffering/repowering are generally prohibited due to physical space constraints and the fact that all placement based optimizations may have been stretched to the maximum degree [1]. Thus, at this stage, only the optimization without impacting placement is desired and layer assignment is an ideal candidate.

A common fact used in [2, 1] is that layer assignment should assign the entire wire between consecutive buffers to the horizontal and vertical layers with the same or similar RC characteristics. In [2, 1] and this work, a wire layer refers to a pair of wire layers (one horizontal and one vertical) with similar parasitics. It is worth noting that wire shaping is effective in improving timing over uniform wire sizing. However, it is shown in [7] that uniform wire sizing is almost as good as non-uniform wire sizing when buffering effect is considered. Thus, as [2, 1], this work does not explore wire shaping for buffer-aware layer assignment.

In [1], the timing constrained minimum cost layer assignment problem is proven to be NP-Complete, and a fully polynomial time approximation scheme running in $O(m \log\log m \cdot n^3/\epsilon^2)$ time is proposed, where n denotes the number of tree nodes and m denotes the number of layers. A polynomial time approximation scheme (PTAS) on the layer assignment problem refers to an algorithm which always returns a solution with cost at most $1+\epsilon$ times the optimal solution for any $\epsilon > 0$ and it runs in a time polynomial in the input size of the problem. The algorithm needs to run in a time proportional to (but not necessarily polynomial in) $1/\epsilon$ unless P=NP. When a PTAS also runs in a time polynomial in $1/\epsilon$, the algorithm becomes a fully polynomial time approximation scheme (FPTAS). Given an NP-Complete problem, an FPTAS is essentially the best algorithm one can have. Note that the FPTAS in [1] is not efficient enough and requires an important assumption that wire cost needs to be polynomially bounded by m.

In this paper, we propose a new fully polynomial time approximation (FPTAS) scheme which significantly improves the one in [1]. The main contribution of this paper is summarized as follows.

- A new FPTAS algorithm is proposed to approximate the optimal solution by a factor of $1+\epsilon$ in $O(mn^2/\epsilon)$ time for any $\epsilon > 0$. This improves the algorithm in [1] which runs in $O(m \log\log m \cdot n^3/\epsilon^2)$ time.
- The proposed algorithm is based on a new linear time dynamic programming algorithm for bounded-cost layer assignment and efficient oracle queries.
- In contrast to [1], the new algorithm does not need the assumption on wire cost.
- The new FPTAS algorithm is highly practical. Experimental results on industrial testcases demonstrate that the new FPTAS algorithm approximates the optimal solution by only few percent additional wire as guaranteed theoretically, with up to 6.5× speedup. This gives another $> 2\times$ speedup over [1].

2. PRELIMINARIES

The problem formulation follows the one in [1]. For completeness, it is included as follows. In the problem, a buffered routing tree $T = (V, E)$ is given where V consists driver, sinks, Steiner nodes and buffer locations, and $E \subseteq V \times V$. Let $n = |V|$. Denote by $V_r(T), V_t(T), V_b(T)$ the driver (or root), the set of sinks (or terminals), and the set of buffers in a tree T, respectively. The buffered tree is assumed to be binary and can be obtained from various buffering techniques such as [8, 12, 13, 14].

A set of m routing layers are given as $L = \{l_1, l_2, \ldots, l_m\}$. In this paper, a routing layer refers to a pair of horizontal layer and vertical layer with the same or similar parasitics [2, 1]. Elmore delay is used which is widely adopted in interconnect synthesis. The delay of an edge e at layer l, denoted by $d(e,l)$, is computed as $d(e,l) = R_e \cdot (C_e/2 + C_l)$ where R_e, C_e, C_l refer to the edge resistance, edge capacitance and load capacitance, respectively. For an edge e on a layer l, denote the wire cost by $w(e,l)$. This wire cost function is a generic one. For example, it could be defined using wire area (in this case, the thick-metal resources will have larger costs, which are as desired) or wire congestion estimation or a combination of them. Note that the previous work [1] requires that the wire cost is polynomially bounded by the number of routing layers m. In contrast, this work does not need this assumption while still able to achieve a much faster FPTAS.

In the routing tree T, associate with driver $V_r(T)$ an arrival time (AT), and each sink in $V_t(T)$ a required arrival time (RAT). After propagating RAT to driver and AT to sinks, a net satisfies the timing constraint if RAT of driver is no earlier than its AT or equivalently AT of each sink is no greater than its RAT.

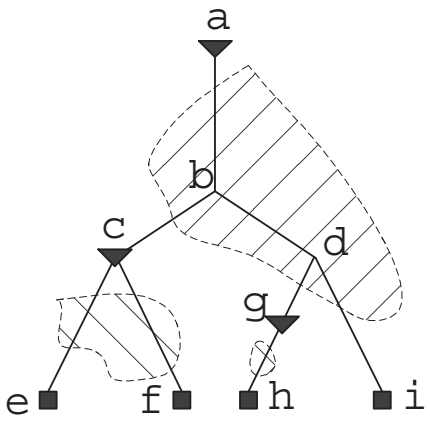

Figure 1: A tree T with three subtree under layer assignment T_a, namely, the subtree formed by node a, c, g, i, by c, e, f and by g, h, respectively.

Let T_a denote *the subtree under layer assignment* of T which is a subtree starting and ending with driver, buffers or sinks and with no other buffer inside [1]. Given any T_a, the root (which is a driver or buffer) of T_a is called *root*, denoted by $V_r(T_a)$, and all other buffers/sinks are called *terminals*, denoted by $V_t(T_a)$. Refer to Figure 1 for a tree with three T_a, namely, the subtree formed by node a, c, g, i, by c, e, f and by g, h, respectively. This work is focused on

layer assignment. Thus, driver/buffers/sinks in T are not to be changed, rather, each T_a will be assigned to different layers for delay-cost tradeoff. In addition, according to [7], all edges in any T_a needs to be assigned to the same routing wire layer which refers to the horizontal wire layer and the vertical wire layer with the same or similar parasitics.

Given a routing layer l and a T_a, compute (in linear time) and store the sum of all edge capacitances and the sum of all wire costs. We also pre-compute and store each root-terminal path delay in T_a for each routing layer l. This can be efficiently performed. For example, for T_a formed by nodes c, e, f in Figure 1, the delay for path $c-e$ is computed as

$$\begin{aligned} D(c,e) &= R(c)\cdot[C_e(c,e)+C_e(c,f)+C(e)+C(f)] \\ &\quad +R_e(c,e)\cdot[C_e(c,e)/2+C(e)], \end{aligned} \tag{1}$$

where $C(\cdot), R(\cdot)$ refer to the input capacitance and driving resistance of driver/buffers/sinks, respectively, and $C_e(\cdot,\cdot)$, $R_e(\cdot,\cdot)$ refer to the capacitance and the resistance of the edge, respectively. One can compute all root-terminal delays of T_a by a breadth-first traversal of T_a in $O(|T_a|)$ time. For example, for T_a formed by a, c, g, i and for each routing layer l, one can compute and store the delay of path $a-c$, of path $a-g$ and of path $a-i$ by a breadth-first traversal of T_a. Note that they are computed before the algorithm is performed. It is a one-time cost which needs totally $O(mn)$ time.

Define *the cost of a layer assignment for the tree* T, also called *tree cost*, as the sum of the costs for all edges in T. As in [1], our problem is to satisfy a timing constraint by layer assignment with minimum tree cost. It can be formulated as follows.

Timing Constrained Minimum Cost Layer Assignment: Given a buffered binary routing tree $T = (V, E)$, a set of routing layers L, and costs of each wire on each layer, to compute a layer assignment solution such that the required arrival time at driver is no earlier than its arrival time and the tree cost is minimized.

This problem is proven to be NP-Complete in [1] by reduction from bipartition problem.

3. THE ALGORITHMIC FLOW

Our new FPTAS shares the algorithmic flow with the previous work [11, 1], however, the underlying core algorithms are very different. The algorithmic flow is first briefly described for completeness and the critical differences are then highlighted. All wire cost values are first scaled by a big real number such that after scaling, all cost values are positive integers. This uniform scaling will not change the optimal layer assignment (but just its cost). Denote by W^* the tree cost (i.e., total wire cost) in the optimal solution of the timing constrained minimum cost layer assignment problem. Given any positive integer x, if we have a procedure which can always tell whether $W^* \geq x$ or not efficiently, it is then not hard to design an algorithm to compute W^*. For example, one can perform a binary search within the upper and lower bounds of the optimal cost W^* by repeatedly calling the procedure. Such a procedure is called the *oracle*. It is somewhat magic since it is able to (approximately) tell whether $W^* \geq x$ or not for any x in polynomial time even if the value of the optimal tree cost W^* is not known.

There are two critical steps in designing an efficient FPTAS for layer assignment: (1) construct an oracle which answers the above query efficiently, and (2) limit the runtime due to oracle queries for efficient computation.

In this paper, *both of these critical steps are significantly improved* compared to [1]. As a result, our algorithm runs in $O(mn^2/\epsilon)$ time while the algorithm in [1] runs in $O(m \log\log m \cdot n^3/\epsilon^2)$ time. The fast oracle query is based on a new linear-time dynamic programming algorithm for cost bounded layer assignment. The oracle query runtime is significantly reduced by adaptively modifying ϵ to effectively tradeoff approximation and runtime. Further, [1] requires that the wire cost is polynomially bounded by m while our algorithm does not need this assumption.

A key component for oracle construction is the approach which can compute the optimal layer assignment solution subject to a given total cost bound $\overline{W}$. That is, the approach can answer the following query: given a cost budget $\overline{W}$, whether there is a solution satisfying the timing constraint with cost no greater than $\overline{W}$. For this, the previous work [1] proposes a dynamic programming algorithm. It computes the solution in multiple iterations and at each iteration, solutions are propagated with cost up to a number which will be incremented after each iteration until it reaches the given cost bound $\overline{W}$. In this work, instead of computing the solution iteration by iteration, we will propagate all solutions with cost no greater than $\overline{W}$ in just one run, which can significantly reduce the runtime.

4. THE LINEAR TIME DYNAMIC PROGRAMMING

4.1 Motivation

The algorithm in [1] is different from Lillis algorithm in [8] in that [1] computes the solutions in multiple iterations and at each iteration, solutions are propagated with cost up to a w and w is incremented until it reaches $\overline{W}$. This algorithm runs in $O(mn\overline{W}^2)$ time, which is not highly efficient due to running multiple iterations. Our idea is to propagate all solutions in just one run while only maintaining those solutions with cost no greater than $\overline{W}$. A direct implementation of this idea leads to an algorithm still running in quadratic time by modifying the algorithm in [8]. The reason is that in [8], when merging two branches, without additional properties on Q, C, W, the merging time is quadratic (in terms of the number of solutions) even if the number of the non-dominated solutions after merging is much less.

On the other hand, a critical observation in layer assignment is that it is possible to limit the number of non-dominated solutions during solution propagation to be linear, by exploring some specific properties in layer assignment problem. This means that if one can always minimize the number of dominated solutions generated, the algorithm can run fast. Our new algorithm achieves this and it runs linearly in m, n and the cost bound $\overline{W}$, significantly faster than the quadratic algorithm in [1].

4.2 The Algorithm

4.2.1 The Framework

In our dynamic programming algorithm, the tree T will be processed in a bottom-up fashion. Layer assignment is first performed to the subtrees T_a directly linking to sinks of T and different layer assignments are computed subject to the wire cost budget $\overline{W}$. That is, the cost of any solution cannot be greater than $\overline{W}$ during propagation. These par-

tial solutions will be updated by processing the immediately upstream subtrees. The process is iterated until the driver is reached. A layer assignment (with cost budget no greater than $\overline{W}$) satisfies the timing constraint if and only if the RAT at driver is no earlier than the AT at driver. If the timing constraint is met, arbitrarily return a solution satisfying the timing constraint. Otherwise, return no solution.

There are two operations in the algorithm, namely, *subtree processing* and *solution update at buffer*. In the algorithm, suppose that there is a subtree under layer assignment T_a where all terminals (buffers) of T_a have been processed and the root (buffer) of T_a is not yet processed. The subtree T_a will be processed as a whole and Q, C, W will be accordingly updated. This operation is called subtree processing. Note that there is no cross-layer assignment in T_a since wire tapering/shaping is not desired considering buffering effect [7, 1]. At each buffer, all the solutions will be updated considering the buffer delay and dominated solutions will be pruned. This operation is called solution update at buffer.

It is helpful to gain some insight by investigating the number of solutions during solution propagation. The cost of each solution could range between 0 and $\overline{W}$. For the ease of illustration (without impacting the asymptotic complexity), assume that the solution cost ranges between 1 and $\overline{W}$. Since the cost is integer and bounded above by $\overline{W}$, one may think that $m|C|\overline{W}$ solutions need to be propagated during dynamic programming, where $|C|$ denotes the maximum number of different C one may have. However, the critical observation is that one can have *at most $m\overline{W}$ non-dominated solutions immediately before any buffer in the tree*. To see this, after processing a subtree T_a, there are only $n\overline{W}$ solutions, which will be described below as our main focus. After a buffer is processed, C are the same in all solutions. For each W, at most one solution will survive, which is the one with largest Q. Clearly, we have at most $\overline{W}$ solutions after solution update at buffer. Since we also duplicate these solutions to all layers, there are actually $m\overline{W}$ solutions.

At each layer, for each terminal of T_a, a set of solutions are presented where the number of solutions is at most $\overline{W}$ since they are the solutions immediately upstream to the buffers. Suppose that there are n' terminals. Let v denote a terminal and u denote the root. We first update the Q value in each solution as follows. Note that in each solution, all wires in T_a need to be in the same layer. For each terminal v,

$$Q(\gamma_{v,u,l}) = Q(\gamma_{v,l}) - d(T_a, v, u, l), \forall v \in V_t(T_a), \quad (2)$$

Where $Q(\gamma_{u,v,l})$ refers to the RAT corresponding to the solution $Q(\gamma_{v,l})$ after subtracting the delay along the path $v - u$. In other word, it is equal to the RAT at the root of T_a due to updating $Q(\gamma_{v,l})$ by considering path $v - u$. This can be computed efficiently as all terminal-root path delays $d(T_a, v, u, l)$, the total tree capacitance C, and the total tree cost w are pre-computed as described in Section 2 (pre-computation takes $O(mn)$ time for all layers).

At this moment, at each layer, if one simply takes the minimum Q of all the possible paths, it will take $\overline{W}^{n'}$ time assuming that there are $\overline{W}$ solutions at each of n' terminals. This is exactly the *merging process*. For example, in the subtree formed by a, c, g, i in Figure 1, if we have $\overline{W}$ solutions at each of c, g, i, one may need $\overline{W}^3$ time in the solution merging. It can be performed much more efficiently.

For the simplicity of the description of the algorithm, we do not yet update C, W at this moment. This is valid due to the following reasons. At the same layer, all solutions have the same C since the buffers are not changed. For example, in Figure 1, for the same layer assignment for wires on the subtree formed by a, g, i, they have the same C which includes the total wire capacitance of T_a and the sum of input capacitances of buffers at a, g, i. We do not need to update W at this moment since the total wire cost for T_a is the same at a layer. It is clear that solutions can have different Q, W values where Q actually refers to $Q(\gamma_{v,u,l})$ as computed above, and W refers to $W(\gamma_{v,l})$ which is not yet updated. Note that W can still be different since it depends on the layer assignment downstream to the terminal v. In implementation, one could update solutions with $W(T_a)$ appropriately. However, this will not impact the asymptotic time complexity of the algorithm due to lack of the knowledge on $W(T_a)$.

After the merging process, Q is computed and $W(T_a)$ and $C(T_a)$ are then added to the solution. If two solutions at the same layer are not dominated after adding $W(T_a)$, they are also not dominated without adding $W(T_a)$, since $W(T_a)$ are added to all solutions.

4.2.2 Linear Time Multi-Way Merging

To reduce the merging time from the straightforward $\overline{W}^{n'}$ to $\overline{W}$ time, the idea is to only generate *non-dominated solutions* but not all solutions since most of them are dominated by others. C is first removed from consideration since all of them are the same for the same layer. Q, C, W becomes Q, W for each layer. Since the solutions are maintained in the increasing order of W, one first performs a solution pruning based on Q, W which takes linear time such that Q are non-increasing after pruning. Note that for the simplicity of description, Q are only assumed to be non-increasing but not strictly decreasing. In fact, if there are no solutions for a cost w, one will put the Q of its previous solution (with the cost $w-1$ or the last solution with the cost smaller than w) there. This may introduce some redundant solutions but will not impact the asymptotic time complexity. It will simplify the description of the algorithm and allow us to focus on the most important part of the algorithm.

It is clear that after merging, there are at most $\overline{W}$ non-dominated solutions at each layer and $m\overline{W}$ non-dominated solutions for all layers immediately downstream the root buffer in T_a. The problem is how to generate the non-dominated solutions without generating all solutions in merging. Such a process is described as follows.

Formally, at each layer, the merging process is to compute the largest Q, among all $n'\overline{W}$ solutions at n' terminals, for each cost $\sum_{1 \le i \le n'} W_i = j$ where $j = 1, 2, \ldots, \overline{W}$. We begin with a special case with $n' = 2$ branches for the ease of illustrating our idea which will be generalized to handle more terminals. At each terminal, Q are non-increasing and W are monotonically increasing. The merging is essentially the same as the linear time Q, C merging in [9].

In merging, two set of solutions $\{a_1, a_2, \ldots, a_k\}$ and $\{b_1, b_2, \ldots, b_k\}$ with $k = \overline{W}$ are given. One has $Q(a_1) \le Q(a_2) \le \ldots, \le Q(a_k)$ and $W(a_i) = i$, and $Q(b_1) \le Q(b_2) \le \ldots, \le Q(b_k)$ and $W(b_i) = i$. Note again that $\le$ but not $<$ is used to simplify the description of the algorithm.

Suppose that after merging, the best Q value with the cost t, denoted by Q_t, is obtained by choosing a_i and b_j. Thus, $Q_t = \min\{Q(a_i), Q(b_j)\}$ and $i + j = t$. We claim that

after merging, the best Q for cost $t+1$, denoted by Q_{t+1}, must be the maximum Q between $\min\{Q(a_{i+1}), Q(b_j)\}$ and $\min\{Q(a_i, b_{j+1})\}$.

For example, in Figure 2, given two solution sets sorted by cost and $k = \overline{W} = 5$. Suppose that the maximum Q solution for the cost 5 is obtained from merging $a_3 = (3,7)$ and $b_2 = (2,8)$. Subsequently, the best Q for the cost 6 must be one of merging $a_4 = (4,5), b_2 = (2,8)$ and $a_3 = (3,7), b_3 = (3,4)$. One can easily see that it is obtained by merging a_4, b_2.

(W,Q)	a	b
1	(1,10)	(1,12)
2	(2,8)	(2,8)←
3	(3,7)←	(3,4)
4	(4,5)	(4,2)
5	(5,2)	(5,1)

Figure 2: An example for merging two sets of solutions.

We prove the claim by contradiction. Without loss of generality, assume that $Q(a_i) \leq Q(b_j)$. Thus, $Q_t = Q(a_i)$.

Case 1: $Q(a_{i+1}) \geq Q(b_{j+1})$. We claim that $Q_{t+1} = \min\{Q(a_{i+1}), Q(b_j)\} = Q(a_{i+1})$ by choosing a_{i+1} and b_j. Suppose to the contrary that the best Q, denoted by Q'_{t+1}, is $> Q_{t+1} = Q(a_{i+1})$. We have $Q'_{t+1} > Q(a_{i+1}), Q(b_{j+1})$. As such, one has to choose one solution from $\{a_1, \ldots, a_i\}$ and one solution from $\{b_1, \ldots, b_j\}$ for $Q'_{t+1} > Q(a_{i+1}), Q(b_{j+1})$. The largest cost is then bounded above by $i + j = t$ which contradicts the fact that the cost needs to be $t+1$.

Case 2: $Q(a_{i+1}) \leq Q(b_{j+1})$. We have two subcases.

1. $Q(a_i) \leq Q(b_{j+1})$. We claim that $Q_{t+1} = \min\{Q(a_i), Q(b_{j+1})\} = Q(a_i)$ by choosing a_i and b_{j+1}. Suppose to the contrary that $Q'_{t+1} > Q_{t+1} = Q(a_i)$ and this is achieved when choosing $a_{i'}$ and $b_{j'}$. Thus, $Q'_{t+1} = \min\{Q(a_{i'}), Q(b_{j'})\}$. Since $i' + j' = t + 1 \geq 3$, at least one of i' and j' is greater than one. Suppose that it is i'. Choosing $a_{i'-1}$ and $b_{j'}$ gives the cost of $i' - 1 + j' = t$ with Q value $Q'_t = \min\{Q(a_{i'-1}), Q(b_{j'})\} > Q(a_i)$ since $Q(a_{i'-1}) \geq Q(a'_i) \geq Q'_{t+1} > Q_{t+1} = Q(a_i)$ and $Q(b_{j'}) \geq Q'_{t+1} > Q_{t+1} = Q(a_i)$. This contradicts the fact that the best Q for the cost t is $Q_t = Q(a_i)$.

2. $Q(a_i) \geq Q(b_{j+1})$. We claim that $Q_{t+1} = \min\{Q(a_i), Q(b_{j+1})\} = Q(b_{j+1})$ by choosing a_i and b_{j+1}. Suppose to the contrary that $Q'_{t+1} > Q_{t+1} = Q(b_{j+1})$. We have $Q'_{t+1} > Q(a_{i+1}), Q(b_{j+1})$ (since $Q(a_{i+1}) \leq Q(b_{j+1})$ in Case 2). Similar to Case 1, this means that one needs to choose one solution from $\{a_1, \ldots, a_i\}$ and one solution from $\{b_1, \ldots, b_j\}$. Thus, the cost is at most $i + j = t$. This contracts the fact that the cost needs to be $t+1$.

Based on the above fact, the merging algorithm is immediate. One can associate a pointer with each set of solutions and the pointer is initialized at the first element in each solution set. Each time, one just needs to tentatively pick an element immediately following each pointer (totally 2 possibilities), compute the Q, find the best element, and move the pointer accordingly. This process certainly takes only $2\overline{W}$ time per layer and $O(m\overline{W})$ time for all layers.

One can generalize the above technique to merge n' terminals to achieve linear-time n'-way merging. Precisely, it takes only $n'\overline{W}$ time per layer and $O(mn'\overline{W})$ time for all layers for the subtree T_a with n' terminals. The proof will be more complicated but the idea is the same. It is omitted due to space limitation. We reach Lemma 1.

Lemma 1: Given n' sets of solutions, each of which consists of $k = \overline{W}$ solutions, i.e., $\{a_1^1, a_2^1, \ldots, a_k^1\}$ and $\{a_1^2, a_2^2, \ldots, a_k^2\}$, $\ldots$, $\{a_1^{n'}, a_2^{n'}, \ldots, a_k^{n'}\}$. Given Q, W functions such that $Q(a_1^s) \leq Q(a_2^s) \leq \ldots, \leq Q(a_k^s)$ and $W(a_i^s) = i$ for $s = 1, 2, \ldots, n'$. One can merge them in $O(n'\overline{W})$ time such that after merging, the largest Q is maintained for each $W = i, i = 1, 2, \ldots, n'$.

Note again that after merging, W, C for all solutions will be updated to include the subtree layer assignment cost and subtree capacitance.

4.2.3 Solution Update At Buffer

A "solution update at buffer" operation can be performed in $O(m\overline{W})$ time. After merging, we have one solution per cost per layer, i.e., totally $m\overline{W}$ solutions. First update Q of all solutions to incorporate the root buffer delay. After this, the new Q, W value represents the required arrival time immediately upstream to the root of T_a. For each W, we then compute the largest Q in all layers and put it in the new solution while eliminating all other solutions. We then set C of each new solution to be the buffer input capacitance. After that, we will obtain a solution set with size at most $\overline{W}$. We then prune the dominated solutions among them which takes linear time. After that, we will duplicate these solutions to all layers, which will be used for the subtree immediate upstream to the root. It is clear that this procedure takes $O(m\overline{W})$ time.

4.2.4 Time Complexity

Processing a subtree T_a takes the time linearly in the number of its terminals, the cost budget $\overline{W}$ and the number of layers m. The solution update at each buffer takes $m\overline{W}$ time. Summing over all terminals, this gives us the total runtime of $O(mn\overline{W})$ time which significantly improve the $O(mn\overline{W}^2)$ time algorithm in [1]. We reach Lemma 2.

Lemma 2: Given a tree with n nodes and m layers, the optimal layer assignment subject to the cost budget $\overline{W}$ can be computed in $O(mn\overline{W})$ time.

5. ORACLE CONSTRUCTION

The oracle is a procedure to call the dynamic programming for the layer assignment with bounded cost. The oracle construction is the same as [1] except the usage of the above new linear-time algorithm. It is included as follows for completeness.

Given any integer x, the oracle can decide whether $x \geq W^*$. Given any positive constant number ϵ (generally assume that $\epsilon < n$), at each layer, each wire cost but not wire delay is scaled by a factor of $\frac{x\epsilon}{n}$ followed by down-rounding. Precisely, for each edge e and layer l, wire cost $w(e,l)$ is scaled to $\lfloor \frac{w(e,l)n}{x\epsilon} \rfloor$.

Subsequently, the above dynamic programming algorithm is performed with the scaled wire cost, the same timing constraint, and a cost budget $\overline{W} = n/\epsilon$. That is, only solutions with cost no greater than n/ϵ can be maintained during solution propagation. There are two cases.

- The dynamic programming returns a solution. This

means that a layer assignment which satisfies the timing constraint is found for cost $w \leq n/\epsilon$ with scaled costs. Scaling the cost back, in the original problem, its cost will be no greater than $\frac{n}{\epsilon} \cdot \frac{x\epsilon}{n} + \frac{x\epsilon}{n}(n-1) < (1+\epsilon)x$. This is due to the fact that rounding error is at most $\frac{x\epsilon}{n}$ at each edge and the whole tree has $n-1$ edges. Thus, in the original problem, there is a solution with cost smaller than $(1+\epsilon)x$ satisfying the timing constraint. It means that $W^* < (1+\epsilon)x$.

- The dynamic programming returns no solution. This means that a layer assignment with the scaled cost up to n/ϵ cannot satisfy the timing constraint. Scaling the cost back, the original problem with the cost of $\frac{n}{\epsilon} \cdot \frac{x\epsilon}{n} = x$ cannot satisfy timing constraint. Thus, $W^* \geq x$.

Since the new dynamic programming runs in $O(mn\overline{W}) = O(mn \cdot n/\epsilon)$ time, a query to the oracle takes $O(mn^2/\epsilon)$ time. This significantly improves the $O(mn^3/\epsilon^2)$ oracle query time in [1].

6. BOUND RATIO INDEPENDENT FPTAS

6.1 Motivation

Using oracle, one can design an efficient algorithm to find the approximate optimal solution for the layer assignment problem. Suppose that a lower bound W_l^* and an upper bound W_u^* on the cost of the optimal layer assignment W^* are given. In fact, they are not hard to obtain. For example, the cost of a layer assignment with layers all setting to the largest-cost (resp. smallest-cost) layers can be the upper (resp. lower) bound. One can then perform a binary search within these bounds to find the approximate optimal solution. Basically, each time set an x to be the average of the upper and the lower bounds and use it to query the oracle. If the oracle says that $W^* < (1+\epsilon)x$, reduce the upper bound to $(1+\epsilon)x$. Otherwise, the oracle says $W^* \geq x$, and thus increase the lower bound to x. A binary search will be terminated in logarithmic iterations of oracle queries. In fact, by performing a logarithmic scale binary search technique, [1] is able to query the oracles with $O(\log\log M)$ iterations where M denotes the ratio between W_u^* and W_l^*. This means that the algorithm is significantly dependent on the initial bound ratio M and note that M could be unbounded.

In this paper, the algorithm will be made independent of bound ratio M. This result is due to the key observation is that one does not need to always query the oracle with the same ϵ. This is originally observed in [15] for improving an FPTAS for the restricted shortest path problem. Recall that an oracle query takes $O(mn^2/\epsilon)$ time. This means that the oracle query with larger ϵ gets a coarser approximation running in shorter time while with smaller ϵ it gets finer approximation running in longer time. If one adaptively and appropriately changes ϵ in oracle queries, it is possible to reduce the runtime significantly. To see this, suppose that the target ϵ is 0.5. One can actually query the oracle using large ϵ at first few iterations, i.e., when the range between the upper and lower bound is large. This coarse approximation will very effectively and efficiently (much faster than using small ϵ) reduce the range. When the range becomes small, smaller ϵ will be used and it will eventually converge to the target ϵ which is 0.5. If one is able to reduce ϵ in a geometric sequence such as $\epsilon = \ldots, 32, 16, 8, 4, 2, 1, 0.5$, it is easy to see that the total runtime will be bounded by $O(2 \cdot mn^2/0.5)$ independent of M, the ratio between the initial upper bound and the initial lower bound. The actual adaptive setting of ϵ is more complicated than this simple motivational example, but the idea is the same.

6.2 Details

Our FPTAS adopts the adaptive ϵ technique proposed in [15] and the details of this technique, after being adapted to our context, are elaborated as follows.

Denote by $W_{u,i}^*$ the upper bound after iteration i and $W_{l,i}^*$ the lower bound after iteration i. $W_{u,0}^* = W_u^*$ and $W_{l,0}^* = W_l^*$. Instead of querying the oracle with ϵ, we query it with (a series of different) ϵ' which forms a geometric sequence converging to ϵ. Precisely, set

$$\epsilon_i' = \sqrt{\frac{W_{u,i}^*}{W_{l,i}^*}} - 1. \tag{3}$$

Set

$$x = \sqrt{\frac{W_{u,i}^* \cdot W_{l,i}^*}{1+\epsilon_i'}}. \tag{4}$$

As described above, there are two scenarios after $(i+1)$-th oracle query with x and ϵ' as set above. (1) $W_{u,i+1}^* = (1+\epsilon')x = {W_{u,i}^*}^{3/4} {W_{l,i}^*}^{1/4}$ and $W_{l,i+1}^* = W_{l,i}^*$ or (2) $W_{u,i+1}^* = W_{u,i}^*$ and $W_{l,i+1}^* = x = {W_{u,i}^*}^{1/4} {W_{l,i}^*}^{3/4}$.

In either case, one has

$$\frac{W_{u,i+1}^*}{W_{l,i+1}^*} = \left(\frac{W_{u,i}^*}{W_{l,i}^*}\right)^{3/4}. \tag{5}$$

The above oracle queries are performed until the ratio between upper and lower bound is no greater than 2, i.e., for some i such that $\frac{W_{u,i}^*}{W_{l,i}^*} \leq 2$. Suppose the first i for this to happen is when $i = t$.

Since one oracle query takes $O(mn^2/\epsilon')$ time, the total runtime is bounded by $O(mn^2 \sum_{1 \leq i \leq t} \frac{1}{\epsilon_i'})$. When $i \leq t$, one has

$$\frac{1}{\epsilon_i'} \leq (2+\sqrt{2})\sqrt{\frac{W_{l,i}^*}{W_{u,i}^*}}. \tag{6}$$

To see this, first note that $i \leq t$ means that $\frac{W_{u,i}^*}{W_{l,i}^*} > 2$. In order for

$$\frac{1}{\epsilon_i'} = \frac{1}{\sqrt{\frac{W_{u,i}^*}{W_{l,i}^*}} - 1} < (2+\sqrt{2})\sqrt{\frac{W_{l,i}^*}{W_{u,i}^*}}, \tag{7}$$

one needs

$$\frac{1}{2+\sqrt{2}} < 1 - \sqrt{\frac{W_{l,i}^*}{W_{u,i}^*}}. \tag{8}$$

This is true when $\sqrt{\frac{W_{l,i}^*}{W_{u,i}^*}} < \sqrt{\frac{1}{2}}$ which is $\sqrt{\frac{W_{u,i}^*}{W_{l,i}^*}} > \sqrt{2}$. Subsequently,

$$O(mn^2 \sum_{1 \leq i \leq t} \frac{1}{\epsilon_i'}) = O(mn^2 \sum_{1 \leq i \leq t} \sqrt{\frac{W_{l,i}^*}{W_{u,i}^*}}). \tag{9}$$

Due to Eqn. (5), one has

$$\frac{W^*_{l,i}}{W^*_{u,i}} = (\frac{W^*_{l,i+1}}{W^*_{u,i+1}})^{4/3}, \quad (10)$$

and thus

$$\frac{W^*_{l,i}}{W^*_{u,i}} = (\frac{W^*_{l,t}}{W^*_{u,t}})^{(4/3)^{t-i}}, \quad (11)$$

for any $1 \leq i \leq t$.

$$\sum_{1\leq i\leq t} \frac{W^*_{l,i}}{W^*_{u,i}} = \sum_{1\leq i\leq t} (\frac{W^*_{l,t}}{W^*_{u,t}})^{(4/3)^{t-i}} = \sum_{0\leq j<t} (\frac{W^*_{l,t}}{W^*_{u,t}})^{(4/3)^{j}}. \quad (12)$$

Since t is the first i for $\frac{W^*_{u,i}}{W^*_{l,i}} \leq 2$, this means that

$$1.68 < \frac{W^*_{u,t}}{W^*_{l,t}} \leq 2, \quad (13)$$

and

$$0.59 > \frac{W^*_{l,t}}{W^*_{u,t}} \geq 0.5, \quad (14)$$

due to that

$$\frac{W^*_{u,t}}{W^*_{l,t}} = (\frac{W^*_{u,t-1}}{W^*_{l,t-1}})^{3/4}. \quad (15)$$

Note that

$$\sum_{1\leq i\leq t} \sqrt{\frac{W^*_{l,i}}{W^*_{u,i}}} = \sum_{0\leq j<t} (\frac{W^*_{l,t}}{W^*_{u,t}})^{1/2\cdot(4/3)^{j}}. \quad (16)$$

We have

$$\sum_{0\leq j<t} (\frac{W^*_{l,t}}{W^*_{u,t}})^{1/2\cdot(4/3)^{j}} < \sum_{0\leq j<t} 0.59^{1/2\cdot(4/3)^{j}}, \quad (17)$$

and

$$\sum_{0\leq j<t} 0.59^{1/2\cdot(4/3)^{j}} < \sum_{0\leq j<t} 0.77^{(4/3)^{j}}. \quad (18)$$

This is to compute the sum of a monotonically decreasing geometric sequence, where the sum is bounded by 6.5 as shown in [15]. Thus, the total runtime is bounded by

$$O(mn^2 \sum_{1\leq i\leq t} \sqrt{\frac{W^*_{l,i}}{W^*_{u,i}}}) = O(mn^2 \cdot 6.5) = O(mn^2). \quad (19)$$

6.3 Computing Approximation

Note that after t oracle queries, the ratio between upper and lower bounds is at most 2. As in [1], one can then use the dynamic programming with following scaling which is similar to the oracle construction.

Set x to the current lower bound, i.e., $x = W^*_{l,t}$, and scale each wire cost $w(e,l)$ to $\lfloor \frac{w(e,l)n}{x\epsilon} \rfloor$. Since the cost of the optimal solution $W^* \leq W^*_{u,t} \leq 2W^*_{l,t}$, there exists a solution with scaled cost no greater than $2n/\epsilon$. This solution will be returned as the $1+\epsilon$ approximation. First, suppose to the contrary that one does not obtain a solution within that cost bound. This means that there is no solution with cost $2n/\epsilon \cdot x\epsilon/n = 2x = 2W^*_{l,t}$ in the original (unscaled) problem, which contradicts the fact that $W^* \leq 2W^*_{l,t}$. Second, this solution is a $1+\epsilon$ approximation. Note that scaling the result back by $x\epsilon/n$ still forms a lower bound on the optimal cost W^*, i.e., $W^*_{scaled} x\epsilon/n \leq W^*$ where W^*_{scaled} denotes the optimal solution with the scaled costs. Since the maximum rounding error is $\frac{x\epsilon}{n} \cdot (n-1) < x\epsilon = W^*_{l,t}\epsilon \leq W^*\epsilon$, one easily sees that the cost of the solution is at most $(1+\epsilon)W^*$. This process takes $O(mn \cdot 2n/\epsilon) = O(mn^2/\epsilon)$ time since the cost budget $\overline{W}$ is set to $2n/\epsilon$.

Since the total oracle queries take $O(mn^2)$ time, the whole algorithm runs in $O(mn^2/\epsilon)$ time. This significantly improves the algorithm in [1] which runs in $O(m \log\log m \cdot n^3/\epsilon^2)$. We reach Theorem 1.

Theorem 1: A $(1+\epsilon)$ approximation to the timing constrained minimum cost layer assignment problem can be computed in $O(mn^2/\epsilon)$ time for any $\epsilon > 0$, where n is the number of nodes in the tree and m is the number of routing layers.

7. EXPERIMENTAL RESULTS

The FPTAS algorithm for the timing constrained minimum cost layer assignment problem is implemented in C++. We compare the new algorithm to the optimal non-polynomial time dynamic programming algorithm in [1] and the previous work [1] which is also an FPTAS algorithm but runs slower. The experiments are performed to a set of 1000 buffered nets extracted from an industrial ASIC chip. As in [1], the wire cost is measured by scaled wire area, and thus thick-metal wires have larger cost. Note that other metric can be easily incorporated in our new FPTAS algorithm.

The comparison of our new FPTAS algorithm and the FPTAS in [1] is summarized in Table 1 where both of them are compared to the dynamic programming algorithm. The actual approximation ratio is computed as the ratio between the wire cost of the obtained solution and the wire cost of the optimal solution minus one. We make the following observations.

- The new FPTAS algorithm always returns the solution within the target approximation ratio ϵ. This is theoretically guaranteed by the nature of ours FPTAS algorithm. In fact, the actual approximation ratio is often much smaller than ϵ. For example, the solution returned by the new FPTAS is only 2.2% off the optimal solution when $\epsilon = 5\%$. The optimal solution is the one obtained by dynamic programming, however, it is not guaranteed to run in polynomial time.

- Compared to the dynamic programming algorithm, the new algorithm runs about 4× faster with 2.2% additional wire and can run up to 6.5× faster.

- Compared to the previous FPTAS algorithm in [1], the new algorithm obtains another > 2× speedup. Note that the solutions are different since the final scaling ratio $W^*_{l,i}$ on nets can be different between two FPTAS algorithms. Our new algorithm obtains better solutions in some cases. The > 2× speedup comes from the linear-time dynamic programming for bounded cost layer assignment, compared to the quadratic-time one used in designing the FPTAS in [1], and the efficient oracle queries.

Table 1: Comparison of the optimal dynamic programming algorithm in [1], the previous FPTAS algorithm in [1], and the new FPTAS algorithm on 1000 nets. For the dynamic programming solution, the optimal total wire cost is 225738 and CPU is 648.5 seconds. Actual Approximation Ratio and Speedup of both FPTAS algorithms are computed by comparing to the dynamic programming.

Target Approx.	FPTAS in [1]				New FPTAS			
Ratio ϵ	Total Cost	CPU(s)	Actual Approx. Ratio	Speedup	Total Cost	CPU(s)	Actual Approx. Ratio	Speedup
5%	231678	340.1	2.6%	1.9×	230755	165.1	2.2%	3.9×
10%	237951	258.2	5.4%	2.5×	241217	138.3	6.9%	4.7×
20%	257384	254.7	14.0%	2.5×	252930	121.4	12.1%	5.3×
30%	281947	238.3	24.9%	2.7×	277109	118.6	22.8%	5.5×
40%	299025	225.8	32.5%	2.9×	304714	105.7	35.0%	6.1×
50%	328819	221.2	45.7%	2.9×	321612	99.5	42.5%	6.5×

- It is also clear from Figure 3 that the runtime of the new FPTAS is inversely proportional to the target approximation ratio ϵ, which is desired.

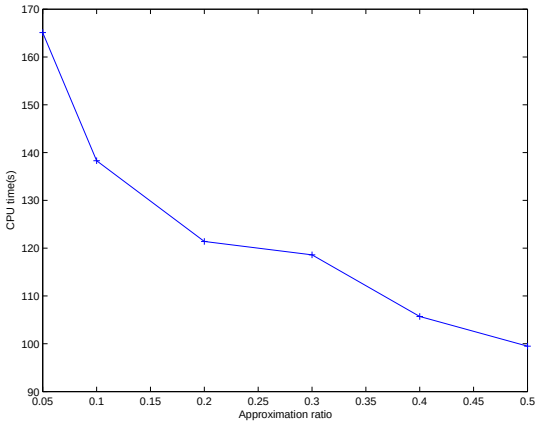

Figure 3: Approximation ratio ϵ v.s. CPU time (s).

8. CONCLUSION

Interconnect synthesis becomes prevalent in physical design tools for achieving timing closure [2]. Timing driven minimum cost layer assignment is a critical component in interconnect synthesis. This problem is NP-Complete and a fast algorithm with provable approximation bound is highly desired.

This work proposes a new fully polynomial time approximation scheme for the problem. Precisely, the new algorithm approximates the optimal layer assignment solution by a factor of $1+\epsilon$ in $O(mn^2/\epsilon)$ for any $\epsilon > 0$, where n is the number of nodes in the tree and m is the number of available routing layers. This significantly improves the previous FPTAS algorithm running in $O(\log\log m \cdot n^3/\epsilon^2)$ time. Further, it does not need any assumption in wire cost as required in [1]. The new algorithm is also highly practical. Our experiments on 1000 industrial testcases demonstrate that compared to dynamic programming, the new algorithm runs about 4× faster with 2.2% additional wire and can run up to 6.5× faster. This gives another $> 2\times$ speedup over the previous FPTAS algorithm on layer assignment.

9. REFERENCES

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

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

[3] J. Fishburn and C. Schevon, "Shaping a distributed-rc line to minimize elmore delay," *IEEE Trans. on Circuits and Systems-I*, vol. 42, no. 12, pp. 1020–1022, 1995.

[4] C.-P. Chen, C. Chu, and D. Wong, "Fast and exact simultaneous gate and wire sizing by lagrangian relaxation," *IEEE Transactions on Computer-Aided Design*, vol. 18, no. 7, pp. 1014–1025, 1999.

[5] J. Cong, K.-S. Leung, and D. Zhou, "Performance-driven interconnect design based on distributed rc delay model," in *DAC*, pp. 606–611, 1993.

[6] J. Cong and K.-S. Leung, "Optimal wire sizing under elmore delay model," *IEEE Transactions on Computer-Aided Design*, vol. 14, no. 3, pp. 321–336, 1995.

[7] C. Alpert, A. Devgan, J. P. Fishburn, and S. T. Quay, "Interconnect synthesis without wire tapering," *IEEE Transactions on Computer-Aided Design*, vol. 20, no. 1, pp. 90–104, 2001.

[8] 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.

[9] 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.

[10] C. Alpert, S. Karandikar, Z. Li, G.-J. Nam, S. Quay, H.Ren, C. Sze, P. Villarrubia, and M. Yildiz, "Techniques for fast physical synthesis," *Proceedings of IEEE*, vol. 95, no. 3, pp. 573–599, 2007.

[11] R. Hassin, "Approximation schemes for the restricted shortest path problem," *Mathematics of Operations Research*, vol. 17, no. 1, pp. 36–42, 1992.

[12] 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," *IEEE Transactions on Computer-Aided Design*, vol. 26, no. 11, pp. 2009–2022, 2007.

[13] 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*, vol. 25, no. 3, pp. 484–489, 2006.

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

[15] F. Ergun and R. Sinha and L. Zhang, "An improved FPTAS for restricted shortest path," *Information Processing Letters*, Vol. 83, No. 5, pp. 287–291, 2002.