Grid-to-Ports Clock Routing for High Performance Microprocessor Designs

Haitong Tian, Wai-Chung Tang, Evangeline F. Y. Young
Department of Computer Science and Engineering
The Chinese University of Hong Kong
{httian,wctang,fyyoung}@cse.cuhk.edu.hk

C. N. Sze
IBM Austin Research Laboratory
csze@us.ibm.com

ABSTRACT

Clock distribution in VLSI designs is of crucial importance and it is also a major source of power dissipation of a system. For today's high performance microprocessors, clock signals are usually distributed by a global clock grid covering the whole chip, followed by post-grid routing that connects clock loads to the clock grid. Early study [7] shows that about 18.1% of the total clock capacitance dissipation was due to this post-grid clock routing (i.e., lower mesh wires plus clock twig wires). This post-grid clock routing problem is thus an important one but not many previous works have addressed it. In this paper, we try to solve this problem of connecting clock ports to the clock grid through reserved tracks on multiple metal layers, with delay and slew constraints. Note that a set of routing tracks are reserved for this grid-to-ports clock wires in practice because of the conventional modular design style of high-performance microprocessors. We propose a new expansion algorithm based on the heap data structure to solve the problem effectively. Experimental results on industrial test cases show that our algorithm can improve over the latest work on this problem [10] significantly by reducing the capacitance by 24.6% and the wire length by 23.6%. We also validate our results using hspice simulation. Finally, our approach is very efficient and for larger test cases with about 2000 ports, the runtime is in seconds.

Categories and Subject Descriptors

B.7.2 [**Design Aids**]: Placement and Routing

General Terms

Algorithm, Design

Keywords

Clock Routing, Grid, Non-tree, Microprocessor Designs

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profi or commercial advantage and that copies bear this notice and the full citation on the firs page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specifi permission and/or a fee.
ISPD'11, March 27–30, 2011, Santa Barbara, California, USA.
Copyright 2011 ACM 978-1-4503-0550-1/11/03...$10.00.

1. INTRODUCTION

In today's high performance systems, clock signals are distributed through a global clock grid [6–10], followed by post-grid routing that connects clock loads to the grid. Early studies showed that most of the clock power dissipation was due to three major categories of capacitances – clock load, clock twig and clock mesh wires, and clock grid buffers. The post-grid clock routing wires (i.e., lower mesh wires and clock twig wires) comprises 18.1% of the total capacitance dissipation [7]. This post-grid clock routing problem is thus a very important one, although not many previous works have addressed it.

Due to the high complexity of microprocessor design, the clock distribution network is usually synthesized and tuned at the same time when different design teams are working on their logic modules. In this case, the clock distribution between the clock grid and the block-level clock ports is subject to conflict of routing resources for data signals. To resolve this conflict and to facilitate simultaneous work between different design teams, a subset of routing tracks have to be reserved for this post-grid clock routing. As a result, this post-grid clock routing problem assumes a given set of reserved tracks, forming a virtual grid structure. The quality of this routing step is of significant importance as it will affect directly the total power consumption, the clock skews and slews at the input of the ports and finally the quality of the chip. These provide motivations to solve this multi-source multi-port post-grid clock routing problem with an objective to minimize the interconnect capacitance while meeting given delay and slew constraints. Traditionally, this step is done manually and iteratively to satisfy the constraints, resulting in a long time to market, especially when the problem size has increased to thousands of clock ports in the layout region. This also motivates the research of a fast algorithm to resolve this clock routing problem effectively.

This post-grid clock routing problem bears a fundamental difference with those previous works on clock tree construction, since the available routing tracks on different metal layers are given and can be very scarce. Besides, in our problem, there are multiple ports and multiple sources in the layout region. There is one very recent work addressing the same problem by Shelar [9, 10] and he proposed a tree growing algorithm to solve the problem with delay and slew constraints. In his algorithm, the clock network is generated by expanding from the sources step by step, and the frontier edge (detailed in Section 5.1) with the smallest wire capaci-

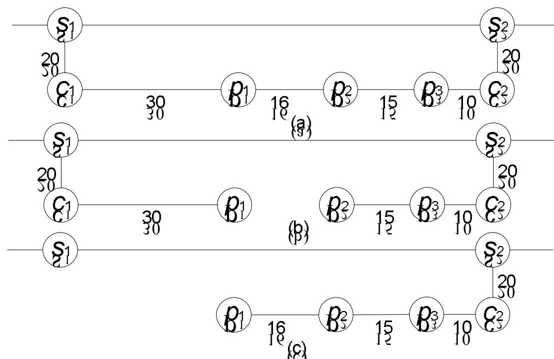

Figure 1: (a) Routing graph with sources s_1 and s_2, ports p_1, p_2 and p_3, and via nodes c_1 and c_2. The numbers near the edges denote the wire capacitances. (b) Routing solution of [10]. (c) Routing solution of our approach, with a 36% reduction in wire capacitance.

tance is chosen every time. Checking against delay and slew constraints is done when a port is reached. However, the routing method in [10] has a couple of intrinsic problems. It uses a top-down tree growing heuristic in which the downstream capacitance information is not available when the trees are being constructed, and it thus can hardly optimize the delay value. In addition, its slew calculation is based on the lumped-RC model instead of the distributed RC model, and this may lead to accuracy and fidelity problems. In Fig. 1, we show a simple example to illustrate the differences between Shelar's appraoch and ours. In this example, three ports p_1, p_2 and p_3 are to be connected to the two sources s_1 and s_2. Fig. 1(b) and Fig. 1(c) show the routing topology obtained by the tree growing approach in [10] and by our approach respectively. In this example, our approach can achieve a 36% reduction in wire capacitance compared with the tree growing approach.

In this paper, we devised an efficient algorithm for this post-grid clock routing problem that can satisfy user given delay and slew bounds while minimizing the total wire capacitance. Note that similar to the formulation in [10], clock skew is optimized in the context of minimizing the maximum delay[1]. We compared our approach with the previous work [10] and can show that with the same delay and slew constraints, our approach can improve over [10] by 24.6% in wire capacitance and by 23.6% in wire length. To further verify the quality of our results, we simulate the constructed clock network using hspice and the simulation results confirm the effectiveness of our approach.

In the following, we will first give a preliminary overview in Section 2 of this hybrid clock network, which motivates this multi-source multi-port post-grid clock routing problem. Problem definition will be given in Section 3 while our approach will be presented in Section 4. Finally, experimental results, comparisons and discussions will be shown in Section 5, followed by a conclusion in Section 6.

[1]Notice that clock skew is upper bounded by the maximum delay. In our problem, this delay bound is set to be very stringent, e.g, within 5ps, which is in reality the limit for the clock skew.

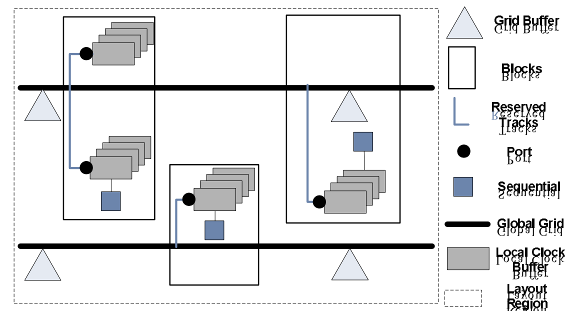

Figure 2: Post-grid Clock Network Distribution

2. POST-GRID CLOCK ROUTING

Clock signals are generated by a phase locked loop (PLL) and reach the global grid through grid buffers. The grid, typically lying on the topmost metal layer, is usually implemented using spines and will distribute clock signals to different regions of the chip. The grid and PLL are usually designed manually. The signals will further be routed through a set of reserved tracks on the lower metal layers to the clock ports of different blocks, and this step is called post-grid clock routing. The block-level ports will be created in such a way to align with the reserved tracks and the clock signals will be further sent to different sequentials inside the blocks. There can be thousands of ports in each layout region in a real post-grid clock routing problem. As shown in Fig. 2, the global grid wires are driven by multiple grid buffers and deliver the clock signals to block-level ports by routing along the reserved tracks on the lower metal layers.

A simple example of this post-grid clock routing problem is shown in Fig. 3(a). In this example, there are five metal layers (from layer 3 to layer 7) with six ports lying on metal layer 3 and the source grid is on metal layer 7. Routing can only be done on those reserved tracks (dashed lines). A sample routing solution is shown in Fig. 3(b). The target is to connect all the ports to the sources without exceeding a very stringent delay bound (which is also an upper bound of the skew), a slew bound and to minimize the total wire capacitance. Note that for this particular instance, our algorithm gets the *optimal* solution as shown in Fig. 3(b).

3. PROBLEM DEFINITION

In this post-grid clock routing problem, we are given (1) a set of reserved tracks (including the source grid which is always on the topmost metal layer) on different metal layers which have alternate routing directions, (2) the locations and capacitances of n ports $P = \{P_1, P_2, ..., P_n\}$ on some lower metal layers, and (3) the types of wires (with different capacitance/resistance tradeoffs) available on each metal layer. We assume that the clock grid on the topmost layer provides zero-skew clock signals. The objective of this post-grid clock routing problem is to connect all the ports to the sources[2] by making use of the reserved tracks and different wire types so as to satisfy the constraints on maximum delay bound D and maximum slew S, and to minimize the total wire capacitance. The delay here is computed accord-

[2]These sources are vias to the source grid on the topmost matal layer.

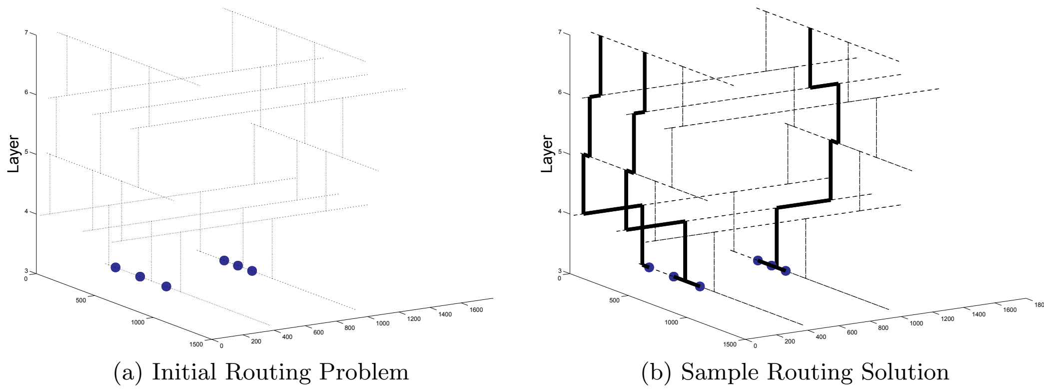

Figure 3: Post-grid Clock Routing Problem

ing to the Elmore delay model due to its simplicity and high fidelity, and the slew is estimated by $\sqrt{(2.2RC)^2 + (S_i)^2}$ according to [5], where R and C denote the resistance and capacitance of the wire segment respectively, and S_i denotes the input slew.

Similar to the previous work [10], we do not optimize the skew directly. This is because the grid-to-ports delay bound (also upper bound the skew) is very stringent and is set to be within 5ps for all the data sets, which is very small compared with the overall circuit skew budget. Therefore, it is not necessary to put the skew as another optimizing objective specifically. In addition, similar to [10], we do not consider buffer insertion in this post-grid clock routing. A very detailed explanation is provided in [10]. In fact, the well-defined grid and reserved tracks make buffer insertion unnecessary for this post-grid clock routing problem.

4. OUR APPROACH

This post-grid clock routing problem can be seen as a multi-source multi-sink[3] tree construction problem with a delay bound, a slew bound, and an objective to minimize the total wire capacitance. We first model the virtual grid of reserved routing tracks by a graph G. The set of vertices contains (1) the block-level clock ports (i.e., the sinks), (2) the possible via positions between reserved tracks on adjacent metal layers, and (3) the clock sources (which are the vias connecting to the source grid). The edges in G represent the wire segments on the reserved tracks connecting ports, vias or sources. Our approach includes a pre-processing step that performs segment merging, finding segment intersections and construction of the graph G and uses some techniques in [3, 4] and it will not be detailed here.

To solve this clock routing problem, we devise a delay-driven *path expansion* algorithm that will propagate from each port in selected directions. A path is a routing between an intermediate node (a via node or a source node) and a block-port along the reserved tracks. In the expansion process, we will always select the path with the smallest Elmore delay (note that it is the total delay of the path) in the current path pool to be further processed. A path p will be *taken* when it reaches a source. Then, all the paths that intersects with p will also be considered and *taken* if no delay nor slew violation occurs. This path expansion step will be repeated until all the ports are connected, or no more ports can be connected without violating the delay and slew constraints. These are the basic steps of our delay-driven path expansion algorithm. It will be invoked repeatedly with a pre-processing step that will connect up some critical ports first. Finally, some post-processing techniques are performed to further reduce the total wire capacitance. A flow of our approach is illustrated in Fig. 4.

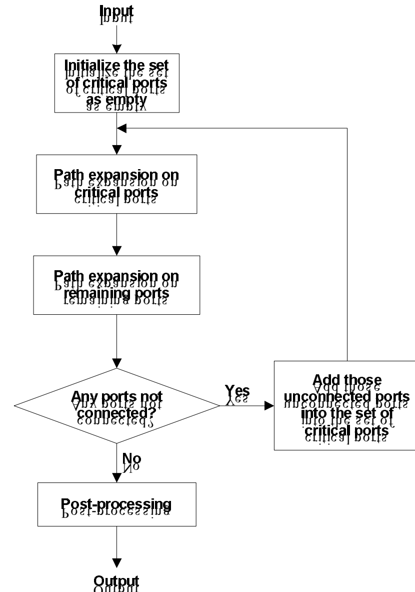

Figure 4: An Overall Flow of Our Approach

4.1 Delay-driven Path Expansion Algorithm

In this delay-driven path expansion algorithm, we will propagate from all the ports simultaneously along the reserved tracks to reach a source. A heap data structure H is used to store all the currently expanding paths sorted according to their Elmore delays. At the beginning, the heap H is initialized with all the ports, which can be regarded as zero length paths with zero delay.

[3]These "sinks" are block-level clock ports in our problem and are different from the "sinks", which are flip-flops or latches, in traditional clock routing problems.

In each step, we will pick a path p from the top of the heap, which has the smallest Elmore delay among all the paths in H. We will then check whether p has reached a source. If not yet, we will expand p vertically up if a via [4] exists at the endpoint $last(p)$ of p or will otherwise expand sideways (horizontally or vertically, depending on the track direction of the metal layer the last node of p is lying on) along the reserved tracks. We will first compute the Elmore delays of these new paths. Those new paths with Elmore delay smaller than the delay limit D will be inserted into the heap H. The path p will then be removed from H.

However, if the path p has reached a source, we will first check against the delay and slew constraints. If no violation occurs, we will take this path p into our routing solution. Suppose that the path p is expanded from a port $port(p)$, all the paths originating from $port(p)$ will be removed from H. Furthermore, we will process every path q where q intersects with p. All these paths will be considered in a non-decreasing order of their Elmore delays. For each of these paths q, we will check whether connecting q to p in the routing solution will violate any constraint at $port(q)$ as well as at any port in the current clock tree under construction. If any violation occurs, we will just neglect q and consider the next candidate. Otherwise, we will take q into the routing solution and connect it to p. We call these paths which do not come to the top of the heap but are processed *chain paths*. Note that once a path is taken into the routing solution, all the nodes on it will be regarded as "sources" for later expansions, and all the paths originating from its port will be removed from H.

Wire length reduction is not directly addressed in our algorithm. But as we always choose a path with the minimum delay to expand and delay is closely related to wire length, paths with shorter wire lengths will have a higher chance to be selected and processed. Therefore, we can expect a reduction in wire length using our approach. A pseudo-code of this path expansion algorithm is shown in Algorithm 1.

4.1.1 Processing of Chain Paths

In the above path expansion algorithm, after a path p is taken into the routing solution, we will process all the paths that intersect with p in the algorithm. First of all, we will initialize a current routing tree T_p as the single path p and initialize a set $chain(p)$ with all the paths in H that intersect with p. The paths in $chain(p)$ are sorted according their Elmore delays in a non-decreasing order. We will then do the following recursively until the set $chain(p)$ becomes empty. First, we will pick and remove a path p_1 from $chain(p)$ that has the smallest Elmore delay. We will then check if connecting p_1 to T_p will violate the delay or slew constraints for $port(p_1)$ as well as for all the existing ports in T_p. If yes, p_1 will be neglected and the next path in $chain(p)$ will be considered. Otherwise, p_1 will be added into T_p and all the paths originating from $port(p_1)$ will be removed from H. Furthermore, all the paths in H that intersect with p_1 will be added into $chain(p)$ recursively.

[4]Note that the capacitance and resistance of the vias are neglected here for simplicity. The same assumption was made in the previous work [10]. However, the via capacitance and resistance can be easily incorporated into our framework by considering them when computing the delay of a path.

Algorithm 1: Path Expansion Algorithm

```
1  begin
2      while H is not empty do
3          p = delete_min(H);
4          if p connects to source and d(p) ≤ D and
           s(p) ≤ S then
5              T_p ← p;
6              clean up H;
7              //remove all paths in H that originate
               //from port p
8              foreach p' intersects with p do
9                  chain(p) ← p';
10             end
11             while chain(p) is not empty do
12                 q = delete_min(chain(p));
13                 if adding q to T_p does not violate D and
                   S constraints then
14                     connect q to T_p;
15                     foreach p' intersects with q do
16                         chain(p) ← p';
17                     end
18                     clean up H;
19                     // remove all paths in H that
20                     // originate from port q
21                 end
22             end
23             Store T_p as one clock tree in the solution;
24         else
25             H ← expansion of p in selected directions;
26         end
27     end
28 end
```

4.2 Pre-processing to Connect Critical ports

The path expansion algorithm does not guarantee connecting all the ports to the sources successfully especially when the user specified constraints are too stringent. If there are critical ports (far away from sources or with very large port capacitance) which are harder to satisfy the requirements, it will be better to generate smaller trees for them first before handling others. Therefore, our post-grid clock routing algorithm involves iterations of the path expansion algorithm and will identify critical ports that fail to be connected to a source in a previous iteration. Those critical ports will be given higher priority to be processed in the next path expansion iteration such that smaller clock trees are more likely to be generated to connect them.

The pseudo-code in Algorithm 2 summarizes the overall flow of our approach. We create a set of critical ports P_c which is initialized as ϕ. We then enter the path expansion iterations in which we first execute the path expansion algorithm on the set of ports in P_c. This gives the critical ports a higher priority to be routed to the sources. We will then execute the path expansion algorithm on the remaining ports $P - P_c$. Notice that these remaining ports may also be connected to the trees constructed for the critical ports. After that, all the ports that cannot be routed to a source in this round will be added to P_c. Priorities also exist in P_c in which a higher priority is given to those most recently added

Algorithm 2: Main Program

1 **begin**
2 $P \leftarrow$ all ports;
3 $P_c \leftarrow \phi$; //critical ports
4 k=0;
5 **repeat**
6 Initialize H as P_c;
7 *path expansion*() with H initialized as P_c;
8 Initialize H as $P - P_c$;
9 *path expansion*() with H initialized as $P - P_c$;
10 $P_c \leftarrow P_c$ + ports that fail to be connected to a source;
11 $k \leftarrow k + 1$;
12 **until** *all sinks are connected or* $k > K$;
13 **if** *all sinks are connected* **then**
14 Post-process;
15 //*wire replacement* and *topology refinement*
16 **else**
17 No solutions under current constraints;
18 **end**
19 **end**

ports. We repeat these steps until all the ports are connected or the number of iterations exceeds a user defined limit K.[5]

4.3 Post-processing to Reduce Capacitance

For all the data sets, there are two types of wires on each layer with capacitance and resistance tradeoffs[6]. The first type has higher capacitance but lower resistance per unit length, while the second type has lower capacitance but higher resistance per unit length. The per unit length delay of type-one wire is less than that of type-two wire on all the layers. In our path expansion algorithm, we will first just use type-one wire on all layers to optimize delay as much as possible. A post-processing step is then performed to reduce the total wire capacitance as long as the delay and slew constraints are maintained by changing the wire types. Two techniques, *wire replacement* and *topology refinement*, are invoked in this post-processing step.

4.3.1 Wire Replacement

This refinement process is done for the trees in the clock network one after another with the following steps. First, all the terminal ports in the current tree are stored in a port pool P_x in which they are sorted in a non-decreasing order of their Elmore delays, and the port P_l with the largest delay in the tree will be recorded. We will then sequentially explore all the ports in P_x. Without loss of generality, lets assume that the currently processing port is P_i, and node P_j is the parent node of P_i in the tree. We use $e(P_i)$ to denote the edge connecting P_i and P_j. We will then check whether any violation occurs if $e(P_i)$ is replaced by the second type of wire. If not, we will replace it with the second type of wire and set $P_i = P_j$. This step is repeated until the delay or slew constraint is violated at any port in the current tree, or when P_i becomes an ancestor of the node P_l (since we do not want to increase the largest delay in this tree). Port P_l will be finally explored after all other ports

Algorithm 3: Wire Replacement

1 **begin**
2 $T_r \leftarrow$ all trees;
3 **while** T_r *is not empty* **do**
4 $T_i \leftarrow$select one tree in T_r;
5 $P_l \leftarrow$ port with the largest Elmore delay in P_x;
6 $P_x \leftarrow$ all terminal ports in T_i except P_l;
7 **while** P_x *is not empty* **do**
8 $P_i \leftarrow$ port node in P_x with the smallest Elmore delay;
9 $P_a \leftarrow$ lowest common ancestor of P_l and P_i;
10 **repeat**
11 Replace $e(P_i)$ using the second type of wire if no violation occurs;
12 $P_i \leftarrow parent(P_i)$;
13 **until** $\exists P_k \in T_i$ *where* $d(P_k) > D$ *or* $P_i = P_a$;
14 $P_x \leftarrow P_x - P_i$;
15 **end**
16 $P_i \leftarrow P_l$, $P_a \leftarrow$ tree root of T_i;
17 repeat steps 10-13;
18 $T_r \leftarrow T_r - T_i$;
19 **end**
20 **end**

Algorithm 4: Topology Refinement

1 **begin**
2 $P_y \leftarrow$ all terminal ports;
3 $sort(P_y)$ in a non-increasing order of their Elmore delays;
4 **while** P_y *is not empty* **do**
5 $P_i \leftarrow$ a port in P_y;
6 *modified path expansion*() on P_i;
7 // Paths expand toward all directions, and the
8 //path with smallest wire capacitance will be
9 //expanded first
10 $P_y \leftarrow P_y - P_i$;
11 **end**
12 **end**

in the tree have been processed. In our implementation, the above process is repeated three times, as we find that for most test cases, running more iterations of this wire replacement process brings little or no capacitance reduction. The pseudo-code in Algorithm 3 details the flow of this wire replacement process.

4.3.2 Topology Refinemen

In the path expansion algorithm, we will expand a path p upwards as long as the end node of p is at a via connecting to the upper layer. Besides, chain paths are greedily processed as long as the delay and slew bounds are maintained. Thus, there are still chances to bring down the capacitance by changing the topology of the initially constructed trees. To achieve this, we will employ a topology refinement step on all the terminal ports as follows. First, we will sort all the ports that are terminal nodes in the trees in a non-increasing order of their Elmore delays in a port pool P_y. These ports will be processed sequentially in the algorithm. For any port P_i being processed, we will first disconnect P_i from the tree it is currently connecting to, and record the total wire capacitance C_b of the removed path p_i. A new path expansion algorithm will then be invoked at P_i which

[5]In this case, the algorithm fails to converge to a feasible solution. Note that this may happen when the delay or slew constraints are too stringent.

[6]Our algorithm can also handle the case that multiple types of wire are available on each layer.

Table 1: Comparisons with TG

Test Cases	No. Sinks	Capacitance (pf)				Wire Length (mm)				Delay (ps)	Runtime (s)		
		TG x_1	Ours1 x_2	Improvement $\frac{x_1-x_2}{x_1}$ %	Ours	TG y_1	Ours1 y_2	Improvement $\frac{y_1-y_2}{y_1}$ %	Ours		TG	Ours1	Ours
test1	300	3.3	2.6 (2.8)	20.9 (16.0)	2.3	12.6	10.0 (10.6)	20.1 (15.6)	10.6	0.45	0.02	0.23	0.20
test2	1846	13.7	9.7 (10.6)	29.2 (22.4)	5.0	42.9	32.3 (34.9)	24.8 (18.6)	34.2	1.15	0.10	2.53	2.68
test3	836	8.1	5.2 (5.8)	36.3 (28.2)	4.2	32.2	20.5 (23.1)	36.3 (28.5)	22.6	0.80	1.35	2.37	2.20
test4	502	5.3	4.0 (4.5)	23.7 (14.6)	1.7	12.4	9.5 (11.0)	23.0 (11.0)	10.6	1.35	0.03	2.81	2.91
test5	137	1.4	1.1 (1.2)	21.1 (15.7)	0.5	3.4	2.7 (3.1)	19.6 (10.5)	3.0	1.10	0.01	0.07	0.09
test6	724	7.9	5.7 (6.2)	27.1 (21.7)	2.5	18.8	14.2 (15.5)	24.7 (17.4)	15.3	1.25	0.05	0.57	0.67
test7	981	9.9	7.5 (8.2)	23.8 (17.2)	3.1	23.2	17.9 (19.9)	23.0 (14.1)	19.5	1.45	0.05	0.87	1.02
test8	538	5.9	4.5 (4.8)	24.5 (18.0)	1.9	14.1	10.8 (12.2)	23.7 (13.3)	11.9	1.80	0.04	0.41	0.50
test9	1915	19.9	14.3 (15.6)	28.3 (21.5)	5.5	46.1	33.2 (37.0)	28.1 (19.7)	35.6	2.75	0.13	2.98	3.38
test10	1134	10.7	8.6 (9.4)	19.4 (12.4)	3.4	25.8	20.2 (22.0)	21.9 (14.8)	21.4	1.90	0.09	6.72	6.88
test11	724	6.6	4.9 (5.3)	24.8 (18.9)	1.9	13.5	10.4 (11.3)	23.1 (16.5)	11.2	1.05	0.04	2.84	3.00
test12	225	2.5	2.0 (2.1)	20.2 (13.8)	0.9	6.3	4.9 (5.4)	22.0 (13.7)	5.4	1.30	0.01	0.13	0.17
test13	859	9.5	7.2 (7.6)	24.0 (19.3)	3.3	24.1	18.8 (20.4)	22.0 (15.4)	20.1	1.10	0.06	0.81	0.95
test14	366	3.9	3.1 (3.3)	20.7 (15.9)	1.4	9.5	7.8 (8.5)	18.4 (10.8)	8.4	0.95	0.04	0.25	0.29
Ave.	792	7.7	5.7 (6.2)	24.6 (18.3)	2.7	20.4	15.2 (16.8)	23.6 (15.7)	16.4		0.14	1.69	1.79

Note 1: Both TG and Ours1 use just type one wire on every layer.
Note 2: "Ours" represents our regular approach of using both types of wire on each layer
Note 3: The figures inside brackets denote the results before the post-processing techniques.

is different from the previous path expansion algorithm that (1) only the second type of wire will be used during the path expansion process, (2) paths will be expanded in all possible directions and (3) the path with the minimum wire capacitance (instead of the minimum wire delay) will be selected and processed first in the expansion process. New paths with wire capacitance less than C_b will be inserted into the heap. Once a path reaches a source or a tree (note that all trees are connected to sources now), we will check whether any violation occurs if the new path is taken. This new path will be taken if no violation occurs. Otherwise, we will continue the modified path expansion algorithm until another path reaches a source or a tree, or when all the paths are exhausted. If all the paths are explored but no path is successfully connected, we will simply restore the original path p_i. The above steps are repeated twice in our implementation. Algorithm 4 shows the flow of this topology refinement process.

4.4 Extension to Handle Large Load Capacitances

In practice, there are cases in which a small number of ports have exceptionally large capacitances that even its shortest direct connection to the nearest source will have a delay exceeding the limit D. To handle these special cases, we have extended our algorithm to first connect those *problematic* ports by a non-tree structure to several sources to bring down the delay to within the limit D. The non-tree structure is constructed by connecting the problematic port to more than one sources by several paths and by adding cross links between those paths.

Consider a particular problematic port P_e, After a path p_1 is taken into the routing solution, we will do the following steps to create a non-tree structure. First, we will expand from node n_l in the opposite direction of the path p_1 to find a nearest source. Let p_2 be the new path. p_2 will be taken into the routing solution if it helps in reducing the delay of P_e. Then, all the crosslinks between p_1 and p_2 (note that crosslinks can only exist at locations with reserved tracks) will be recorded and examined. The computational model in [2] is used to calculate the delays at the ports when crosslinks exist. All the crosslinks that can reduce the delay of P_e will be taken into the routing solution one by one until the delay and slew constraints are met, or when all the crosslinks are exhausted. If the delay and slew constraints are still not met with p_2 and all the crosslinks added, we will set $n_l = parent(n_l)$ and repeat the above steps recursively with one edge up the original path p_1 to find more sources and crosslinks.

After handling all the problematic ports, other ports will be handled as usual according to Algorithm 2. Note that we also allow other ports to connect to the non-tree structures, as long as the delay and slew constraints are not violated.

5. EXPERIMENT RESULTS

The path expansion algorithm proposed in this paper is implemented in C++ and all the experiments are carried out on a Linux machine with 4GB RAM and a Pentium 4 microprocessor running at 3.2GHz. We have also implemented the tree growing approach (TG) in [10] using C++ for comparisons. In the experiments, we assume that the slew of the source signals is 10ps, and the slew bound of the output signals is set to be 15ps. The first three test cases (test1-3) are provided by industry. The remaining eleven test cases are obtained from the circuits used in the ISPD 2010 Clock Network Synthesis Contest [1]. For the ISPD test cases which have no layer information given, five layers of reserved tracks are added according to the track conventions used in test 1-3.

5.1 Comparisons with the Tree Grow (TG) Approach

In the paper [10], Shelar proposed a tree growing algorithm to construct a clock network on reserved tracks in the context of post-grid clock routing . A pool F of frontier

nodes, which is initialized to be all source nodes, is used to store all the current nodes to be expanded. The following steps are performed recursively until all the ports are connected to the sources. First, all unexplored edges adjacent to a frontier node in F are stored in an edge pool E_f and sorted in an ascending order of their edge capacitances. Then, the clock network is built by a greedy edge expansion process, in which all the edges in E_f are sequentially added into the clock network. Furthermore, the frontier node pool F will be updated with the end nodes of the newly added edges. After all the ports are connected to the sources, the final clock network is obtained by deleting redundant edges in the trees. Delay and slew constraints are considered in the algorithm.

Since the approach in [10] considers only one type of wire on each layer, for fair comparison, we compare the result of our approach using just the first type of wire on every layer (i.e., without the wire replacement step and use only type one wire in all the other steps) with the result of [10] using the first type of wire on every layer. In these experiments, we first get the lowest achievable delays obtained by TG empirically on all the test cases and use these delays as our delay bounds. The same slew limit is applied to both methods. The results are shown in Table 1. Column 3 and 7 show the total wire capacitance and the total wire length generated by TG. The results of our approach are shown in column 4 and 8. On average, our approach provides a 24.6% improvement in the total wire capacitance and a 23.6% improvement in the total wire length compared with TG respectively. The running times of both algorithm are shown in the last two columns. As we can see that though our approach is slower, the runtimes are still very practical. For all the test cases, the running times of our approach are within seconds. On average, the major path expansion algorithm, the topology refinement step and the wire replacement step take 74%, 17% and 9% of the total running time respectively. Note that in some cases, the running time of "Ours1" is even larger than that of "Ours" although "Ours1" does not perform the wire replacement step. This is because the inputs to the topology refinement procedure in "Ours" and "Ours1" are different as "Ours1" does not perform wire replacement. There are thus variations in the running times of the topology refinement step.

If we allow both types of wires on each layer, further reduction in wire capacitance can be obtained and the results are shown in column 6, 10 and 14 of Table 1. As we can see from the result, our approach can make good use of the availability of different wire types to further reduce the capacitance. For example, in test2, the wire capacitance can be reduced significantly by 49% (from 9.68pf to 4.92pf) with the wire replacement step.

5.2 Lowest Achievable Delay

Our approach can actually produce solution with better delay than the TG approach. We have run our algorithm on all the test cases to get the smallest achievable delays. The results are shown in Table 3. For almost all the test cases, we can further reduce the delays generated by TG. Take *test3* as an example, we can significantly reduce the delay from 0.80ps to 0.55ps, which shows an advantage of using our method in satisfying stringent user specified delay limits. In practice, designers may not know whether a delay limit is achievable for a circuit. Our approach can help in determining the lowest achievable delay by embedding the algorithm in a binary search loop. This is possible since our approach will take the delay limit as an input constraint.

Table 2: Non-tree Algorithm

Test Cases	C (pf)	WL (mm)	D (x ps)	T(s)	No. P.P.	D_{min} (y ps)	Imp. ($\frac{y-x}{y}$%)
ntest1	2.7	10.5	0.45	0.3	3	0.68	33.8
ntest2	10.1	33.6	0.45	4.7	3	0.71	36.6
ntest3	5.6	22.3	0.60	8.7	3	0.51	-18.5
ntest4	4.2	10.0	1.00	2.1	3	1.26	20.8
ntest5	1.2	2.9	1.03	0.1	3	1.29	20.3
ntest6	6.3	16.3	0.66	1.4	3	1.25	47.0
ntest7	7.6	18.1	1.35	1.6	3	2.02	33.3
ntest8	4.6	11.2	1.30	0.5	3	1.98	34.4
ntest9	14.5	33.9	2.00	25.0	3	2.42	17.4
ntest10	8.7	20.4	1.80	15.9	3	2.33	22.6
ntest11	5.1	11.1	0.80	6.8	3	1.24	35.4
ntest12	2.0	4.9	1.70	0.2	3	1.65	-3.0
ntest13	7.3	19.0	1.25	1.0	3	1.35	7.2
ntest14	3.4	8.8	0.58	0.8	3	0.98	40.8
Ave.	5.94	15.9		4.92	3		23.4

Table 3: Lowest Achievable Delays

Test Cases	Capacitance (pf)	Wire Length (mm)	Delay (ps)	Runtime (s)
test1	2.27	10.6	0.45	0.20
test2	6.08	34.6	0.47	4.69
test3	5.06	24.6	0.55	4.56
test4	1.75	10.2	1.00	2.12
test5	0.55	3.0	0.86	0.12
test6	2.83	15.5	0.83	1.01
test7	3.02	18.0	1.35	1.60
test8	1.86	11.2	1.32	0.50
test9	5.45	33.7	1.95	28.22
test10	3.28	20.1	1.67	22.27
test11	1.92	10.7	0.89	2.87
test12	0.91	5.1	1.12	0.22
test13	3.43	19.9	0.90	1.47
test14	1.48	8.4	0.67	0.40

5.3 Results of the Non-tree Extension

To validate the effectiveness of our proposed non-tree algorithm, we further generate 14 test cases from the original ones (the new test cases have their names starting with an "n"). These new test cases are generated as follows. We first sort the ports according to their minimum Elmore delays, which is the delay when a port is connected to its nearest source directly. Then we increase the capacitances of the first three ports in the list so that their minimum delays increase by at least 50%. Detailed results on these new test cases are shown in Table 2.

Total capacitance, total wire length, delay limits, running time and number of problematic ports are shown in column 2-5 respectively. The delay limits D is got empirically for all test cases. The second last column D_{min} in Table 2 shows the minimum delay of the problematic ports when they are connected to the nearest source *directly*. Therefore, these are the lower bound delays achievable using a tree structure. We can see from the comparsion in the last column that our non-tree approach can reduce further the delay by 23.4% on average. For *ntest3* and *ntest12*, our non-tree algorithm does not help much and it automatically degenerates into the

original path expansion algorithm (the result is thus a set of trees) because of the high density of the ports especially in the surroundings of the problematic ports. For all the other test cases, our proposed non-tree approach can successfully generate a solution in which the maximum port delay is less than the lower bound delay shown in the second last column. This clearly demonstrates the effectiveness of our proposed non-tree algorithm.

5.4 Simulation Results

We further validate our results using hspice simulation. The slew of the input signals are set to be 10ps. Detailed results are shown in Table 4 and Table 5. As we can see from the simulation results, The delay and slew we calculated is very close to the simulation results. For both tables, the correlation coefficient is over 99% between the simulated delay and calculated delay while it is over 94% between the simulated slew and calculated slew. This verifies the effectiveness of our method.

Table 4: Simulation Results for Tree

Test Cases	Calculated Results		Simulation Results	
	Delay (ps)	Slew (ps)	Delay (ps)	Slew (ps)
test1	0.45	10.05	0.45	10.07
test2	1.14	10.32	1.14	10.24
test3	0.80	10.15	0.80	10.15
test4	1.35	10.43	1.34	10.33
test5	1.09	10.29	1.09	10.25
test6	1.25	10.37	1.25	10.32
test7	1.43	10.50	1.43	10.52
test8	1.78	10.76	1.78	10.90
test9	2.75	11.69	2.70	11.43
test10	1.90	10.84	1.90	11.07
test11	1.05	10.26	1.05	10.23
test12	1.28	10.40	1.28	10.31
test13	1.08	10.29	1.08	10.24
test14	0.95	10.22	0.95	10.20

Table 5: Simulation Results for Non-Tree

Test Cases	Calculated Results		Simulation Results	
	Delay (ps)	Slew (ps)	Delay (ps)	Slew (ps)
ntest1	0.45	10.05	0.45	10.07
ntest2	0.45	10.05	0.45	10.07
ntest3	0.60	10.09	0.60	10.11
ntest4	1.00	10.24	1.00	10.21
ntest5	1.03	10.25	1.03	10.22
ntest6	0.66	10.10	0.66	10.13
ntest7	1.35	10.43	1.35	10.39
ntest8	1.29	10.40	1.29	10.33
ntest9	2.00	10.93	2.00	10.70
ntest10	1.80	10.76	1.80	10.97
ntest11	0.80	10.15	0.80	10.16
ntest12	1.70	10.68	1.70	10.81
ntest13	1.25	10.37	1.25	10.30
ntest14	0.58	10.08	0.58	10.10

6. CONCLUSION

In this paper, we present an efficient algorithm using the heap data structure to construct a post-grid clock network on reserved multi-layer metal tracks. We have compared our approach with the state-of-the-art algorithm and show that our algorithm can significantly improve over the previous work with a 24.6% reduction in wire capacitance and 23.6% reduction in wire length on average while maintaining very practical runtimes. We have also extended the algorithm to allow non-tree structures in order to handle the existence of ports with exceptionally large load capacitances and verified our results using hspice simulation. Our algorithm is expected to bring reduced energy consumption, improve grid-to-port delay in real post-grid clock networks.

7. REFERENCES

[1] *ISPD 2010 High Performance Clock Network Synthesis Contest*. http://www.sigda.org/ispd/contests/10/ispd10cns.html.

[2] P. Chan and K. Karplus. Computing signal delay in general rc networks by tree link partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(8):898–902, Aug 1990.

[3] B. Chazelle. Filtering search: A new approach to query-answering. In *24th Annual Symposium on Foundations of Computer Science*, pages 122–132, 1983.

[4] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. In *29th Annual Symposium on Foundations of Computer Science*, pages 590–600, 1988.

[5] C. Kashyap, C. Alpert, F. Liu, and A. Devgan. Closed-form expressions for extending step delay and slew metrics to ramp inputs for rc trees. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(4):509–516, April 2004.

[6] M. Mori, H. Chen, B. Yao, and C.-K. Cheng. A mulitple level network approach for clock skew minimization with process variations. In *Proceedings of Asia and South Pacific Design Automation Conference*, pages 184–187, 2000.

[7] D. Pham, T. Aipperspach, D. Boerstler, M. Bolliger, R. Chaudhry, D. Cox, P. Harvey, P. Harvey, H. Hofstee, C. Johns, et al. Overview of the architecture, circuit design, and physical implementation of a first-generation cell processor. *IEEE Journal of Solid-State Circuits*, 41(1):179–196, Jan. 2006.

[8] P. Restle, T. McNamara, D. Webber, P. Camporese, K. Eng, K. Jenkins, D. Allen, M. Rohn, M. Quaranta, D. Boerstler, et al. A clock distribution network for microprocessors. In *IEEE Journal of Solid-State Circuits*, pages 184–187, 2000.

[9] R. Shelar. An algorithm for routing with capacitance/distance constraints for clock distribution in microprocessors. In *Proceedings of the 2009 international symposium on Physical design*, pages 141–148, 2009.

[10] R. Shelar. Routing with constraints for post-grid clock distribution in microprocessors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(2):245–249, Feb. 2010.