Postgrid Clock Routing for High Performance Microprocessor Designs

Haitong Tian, Wai-Chung Tang, Evangeline F. Y. Young, and C. N. Sze

***Abstract*—Designing a high-quality clock network is very important in very large-scale integrated designs today, as it is the clock network that synchronizes all the elements of a chip, and it is also a major source of power dissipation of a system. Early study by Pham *et al.* in 2006 shows that about 18.1% of the total clock capacitance was due to this postgrid clock routing (i.e., lower mesh wires plus clock twig wires). In this paper, we proposed a partition-based path expansion algorithm to solve this postgrid clock routing problem effectively. Experimental results on industrial test cases show that our algorithm can improve over the latest work by Shelar on this problem significantly by reducing the wire capacitance by 24.6% and the wirelength by 23.6%.**

***Index Terms*—Clock routing, microprocessor design, postgrid.**

I. Introduction

For most chips today, data transfer between different function units is synchronized by a single global clock signals. Routing the clock signals under some stringent delay/slew constraints is one of the most important objectives in designing a chip. For high-performance microprocessors, a global clock grid [6]–[10] followed by postgrid routing is used to distribute clock signals to all elements of the chip. Due to the high complexity of microprocessor design, a subset of routing tracks has to be reserved for this postgrid clock routing. Early studies showed that most of the clock power was due to three major categories of capacitances—clock load, clock twig, and clock mesh wires, and clock grid buffers. The postgrid clock routing wires (i.e., lower mesh wires and clock twig wires) comprises 18.1% of the total capacitance dissipation [7]. Traditionally, this step is done manually and iteratively to satisfy the constraints, resulting in a long time to market. This motivates the research of a fast algorithm to resolve this clock routing problem effectively.

There have been many clock routing algorithms in the literature, such as H-tree [1], geometric matching algorithm [4], exact zero skew algorithm [12], and deferred merge embedding [2]. These algorithms are used in the local clock routing areas, and there are no tracks reserved in this clock routing problem. In the postgrid clock routing areas, the routing tracks on different metal layers are given and routing can only be done where tracks are available. Therefore, it is impossible to directly employ these traditional methods on this new routing problem. There is a recent work addressing the same problem by Shelar [9], [10] and a tree growing (TG) algorithm is proposed to solve the problem with delay and slew constraints. In our previous work, a path expansion algorithm was proposed to effectively solve this problem [11], and this paper serves as an extension to our previous one.

In the following, problem definition is given in Section II while our approach is presented in Section III. Finally, experimental results, comparisons, and discussions are shown in Section IV, followed by a conclusion in Section V.

Manuscript received June 9, 2011; revised August 15, 2011; accepted September 7, 2011. Date of current version January 20, 2012. This paper was recommended by Associate Editor C.-K. Koh.

H. Tian is with the Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL 61801 USA (e-mail: htian3@illinois.edu).

W.-C. Tang and E. F. Y. Young are with the Department of Computer Science and Engineering, Chinese University of Hong Kong, Shatin, Hong Kong (e-mail: wctang@cse.cuhk.edu.hk; fyyoung@cse.cuhk.edu.hk).

C. N. Sze is with the IBM Austin Research Laboratory, Austin, TX 78758 USA (e-mail: csze@us.ibm.com).

Digital Object Identifier 10.1109/TCAD.2011.2170688

II. Problem Definition

In this postgrid 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 $P = \{P_1, P_2, \ldots, P_n\}$ 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 postgrid clock routing problem is to connect all the ports to the sources[1] by making use of the reserved tracks and different wire types so as to satisfy the constraints on maximum delay bound D[2] and to minimize the total wire capacitance.

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 5 ps 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. We can estimate the slew of signals using $\sqrt{(2.2RC)^2 + (S_i)^2}$ according to [5], where R and C denote the resistance and capacitance of the wire segment, respectively, and S_i denotes the input slew. In addition, similar to [10], we do not consider buffer insertion in this postgrid 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 postgrid clock routing problem.

III. Our Approach

A graph G is used to model the virtual grid of reserved routing tracks. 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.

We devise a partition-based delay-driven path expansion algorithm to solve this problem. At the very beginning, we partition the ports into smaller clusters according to some "boundary lines" of the chip. For each cluster, the ports are

[1]These sources are vias to the source grid on the topmost metal layer.

[2]If the clock skew on the grid is nonzero, we can set the delay constraint D to be the original delay constraint minus the clock skew.

simultaneously propagated in selected directions. To make our illustration more clear, we define a new term *path* in our approach as follows: a path is a routing between an intermediate node (a via node or a source node) and a port along the reserved tracks. During the expansion process, we always select the path with the smallest Elmore delay (note that it is the total delay from the last node of the path to the first node of the path) to be further processed. This path expansion step are repeated until all the ports are connected, or no more ports can be connected without violating the delay constraint. These are the basic steps of our partition-based delay-driven path expansion algorithm. It is invoked repeatedly with a preprocessing step to connect up some critical ports first. Finally, some postprocessing techniques are performed to further reduce the total wire capacitance. A flow of our approach is illustrated in Fig. 1.

A. Port Partitioning

In our experiments, we have found that ports rarely connect to a source grid far from itself in the clock network. With this observation, we propose a technique to cluster the ports into several small clusters to speed up the algorithm. The middle lines of two successive source grids are used as the baselines to split the chip into smaller partitions. All the ports lie in the same partition are grouped into one cluster. After grouping the clusters, we sequentially employ the path expansion algorithm on those clusters one after another to connect all the ports.

B. Delay-Driven Path Expansion Algorithm

In the delay-driven path expansion algorithm, we propagate from all the ports in current cluster 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.

In each step, we pop up a path p from the heap, which has the smallest Elmore delay among all the paths in H. If p is not connected to any source yet, it is expanded vertically up if a via[3] exists at the endpoint last(p) of p or otherwise 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. For these new paths, we will compute their Elmore delays. Those new paths with Elmore delay smaller than the delay limit D are inserted into the heap H.

However, if the path p has reached a source, we first check against the delay constraint. If no violation occurs, p is taken 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 are considered in a nondecreasing order of their Elmore delays. For each of these paths q, we check whether connecting q to p lead to any delay violation at port(q) or at any port in the current clock tree. If violation occurs, we just neglect q and consider the next path. Otherwise, q will be connected 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 are regarded as "sources" for later expansions, and all the paths originating from the corresponding port are removed from H.

[3]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.

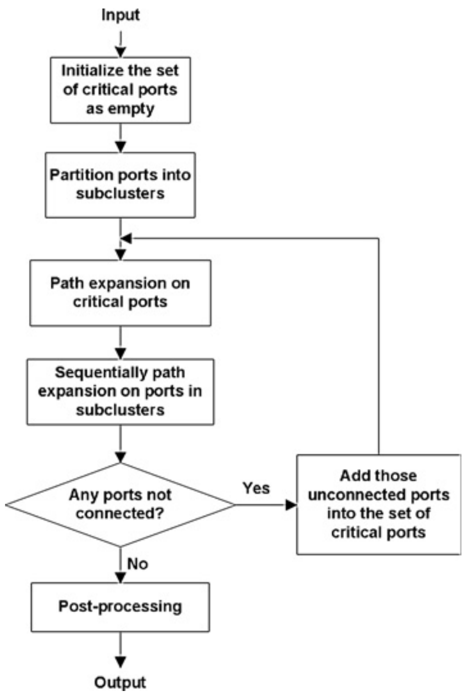
Input
Initialize the set of critical ports as empty
Partition ports into subclusters
Path expansion on critical ports
Sequentially path expansion on ports in subclusters
Any ports not connected?
Yes
Add those unconnected ports into the set of critical ports
No
Post-processing
Output

Fig. 1. Overall flow of our approach.

After a path p is taken into the routing solution, all the paths that intersect with p are also processed. Denote the partial tree constructed by path p as T_p. First of all, we initialize a set chain(p) with all the paths in H that intersect with p. The paths in chain(p) are sorted in a nondecreasing order according their Elmore delays. We then do the following steps recursively until the set chain(p) becomes empty. First, we pop up a path p_1 from chain(p) that has the smallest Elmore delay and check whether connecting p_1 to T_p violate the delay constraint for port(p_1) as well as for all the ports in T_p. If yes, p_1 is neglected and the next path in chain(p) is tried. Otherwise, p_1 is added into T_p and all the paths originating from port(p_1) are removed from H. Furthermore, all the paths in H that intersect with p_1 are added into chain(p) recursively.

C. Preprocessing to Critical Port Connection

The path expansion algorithm does not guarantee connecting all the ports to the sources successfully, especially when the user-specified delay constraint is too stringent. If there are critical ports (far away from sources or with very large port capacitance) which are harder to satisfy the requirement, it is better to generate smaller trees for them first before handling others. Therefore, our postgrid clock routing algorithm involves iterations of the path expansion algorithm and will identify critical ports that fail to be connected to a source in the previous iteration.

We create a set of critical ports P_c which is initialized as empty ϕ. We first invoke the path expansion algorithm on the set of ports in P_c before employing the algorithm on the remaining ports $P - P_c$. This gives the critical ports a higher priority to be routed to the sources. Note 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 are added to P_c. Priorities also exist in P_c in which a higher priority is given to those most recently added ports. We repeat these steps until all the ports are connected or the number of iterations exceeds a user-defined limit K.[4]

D. Postprocessing for Capacitance Reduction

For all test cases, there are two types of wires on each layer with capacitance and resistance tradeoffs.[5] 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 layers. In our path expansion algorithm, we first use type-one wire on all layers to optimize delay as much as possible. A postprocessing step is then performed to reduce the total wire capacitance as long as the delay constraint is maintained by replacing the wire types. Two techniques, wire replacement and topology refinement, are invoked in this postprocessing step.

1) *Wire Replacement:* This refinement process is employed on all 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 pool P_x in which they are sorted in a nondecreasing order of their Elmore delays, and the port P_l with the largest delay in the tree is recorded. The ports in P_x are sequentially processed. Without loss of generality, let us assume that the currently processing port is P_i, and the 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 first check whether any violation occurs if $e(P_i)$ is replaced by the second type of wire. If not, we replace it with the second type of wire and set $P_i = P_j$. This step is repeated until the delay 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 is processed after all other ports in the tree have been explored. In our implementation, this process is repeated three times, as we find that running more iterations of this wire replacement process brings little or no capacitance reduction.

2) *Topology Refinement:* In the path expansion algorithm, we expand a path p upward if the end node of p is a via connecting the upper layer. Besides, chain paths are greedily processed as long as the delay bound is maintained. Thus, there are chances to bring down the capacitance by changing the topology of the initially clock network. To achieve this, we employ a topology refinement step on all terminal ports as follows. First, all the ports that are terminal nodes in the trees are sorted in a nonincreasing order of their Elmore delays in a port pool P_y. These ports are sequentially processed in the algorithm. By using this order, we can work on the port with longer delays first before processing other relatively easier ones. For any port P_i being processed, we 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 is invoked at P_i which is different from the previous path expansion algorithm that: a) only the second type of wire is used during the path expansion; b) paths are expanded in all possible directions; and c) the path with the minimum wire capacitance (instead of the minimum wire delay) are always selected and processed first. New paths with wire capacitance less than C_b are inserted into the heap. Once a path reaches a source or a tree (note that all trees are connected to sources now), we check whether delay violation occurs if the new path is taken. This new path is taken if no violation occurs. Otherwise, we 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 simply restore the original path p_i. The above steps are repeated three times in our implementation.

E. 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 connection to the source will have a delay exceeding the limit D. To handle these special cases, we extend our algorithm to first connect those *problematic* ports by a nontree structure to bring down the delay to be within the limit D.

To identify problematic ports, we sequentially expand each port toward all directions to find a path connecting to a source with the smallest Elmore delay. This Elmore delay is the lower bound delay one can achieve using any tree structures. If it is larger than the delay limit D, the port is classified as a problematic port. This step is used to identify all problematic ports. Consider a particular problematic port P_e, after we make a shortest path connection p_1 for port P_e, we will do the following steps to create a nontree structure. We expand from the first node n_l of p_1 in the opposite direction to find another 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. All the crosslinks between p_1 and p_2 (note that crosslinks can only exist at locations with reserved tracks) are recorded and examined. The computational model in [3] is used to calculate the delays when crosslinks exist. All the crosslinks that can reduce the delay of P_e are taken into the routing solution one by one until the delay constraint is met, or when all the crosslinks are exhausted. If the delay is still violated after adding all the crosslinks, 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. Note that we also allow other ports to connect to the nontree structures, as long as the delay constraint is not violated and the computational model in [3] is used to check the delay constraints.

[4]In this case, the algorithm fails to converge to a feasible solution. Note that this may happen when the delay constraint is too stringent.

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

TABLE I

COMPARISONS WITH TG

Test Cases	No. Sinks	k	Capacitance (pf)				WL (mm)				Delay (ps)	Runtime (s)		
			TG x_1	Ours1 x_2	Improve $\frac{x_1-x_2}{x_1}$ %	Ours	TG y_1	Ours1 y_2	Improve $\frac{y_1-y_2}{y_1}$ %	Ours		TG	Ours1	Ours
test1	300	5	3.3	2.6 (2.8)	20.9 (16.0)	2.3	12.6	10.0 (10.6)	20.1 (15.5)	10.6	0.45	0.02	0.24	0.20
test2	1846	96	13.7	9.7 (10.8)	29.2 (21.1)	5.0	42.9	32.3 (35.6)	24.8 (17.2)	33.7	1.15	0.10	1.39	1.54
test3	836	33	8.1	5.1 (5.8)	36.7 (28.2)	4.2	32.2	20.6 (23.1)	36.7 (28.4)	22.4	0.80	1.35	2.66	2.47
test4	502	26	5.3	4.0 (4.5)	23.8 (14.5)	1.6	12.4	9.5 (11.0)	22.8 (10.8)	9.9	1.35	0.03	1.51	1.62
test5	137	11	1.4	1.1 (1.2)	21.0 (16.3)	0.5	3.4	2.7 (3.1)	19.4 (10.6)	2.8	1.10	0.01	0.08	0.09
test6	724	16	7.9	5.7 (6.1)	27.0 (21.8)	2.5	18.8	14.2 (15.5)	24.6 (17.5)	14.6	1.25	0.05	0.48	0.57
test7	981	11	9.9	7.5 (8.2)	23.8 (17.2)	3.0	23.2	17.9 (19.9)	22.9 (14.0)	17.9	1.45	0.05	0.61	0.76
test8	538	16	5.9	4.4 (4.8)	24.6 (17.9)	1.8	14.1	10.8 (12.3)	23.8 (13.1)	10.8	1.80	0.04	0.37	0.45
test9	1915	21	19.9	14.3 (15.7)	28.3 (21.3)	5.2	46.1	33.1 (37.1)	28.0 (19.4)	32.6	2.75	0.13	2.04	2.44
test10	1134	21	10.7	8.6 (9.3)	19.6 (12.5)	3.2	25.8	20.1 (22.1)	22.2 (14.6)	19.6	1.90	0.09	7.15	7.44
test11	724	11	6.6	4.9 (5.4)	25.0 (18.0)	1.8	13.5	10.5 (11.4)	23.3 (15.5)	10.4	1.05	0.04	1.76	1.87
test12	225	11	2.5	2.0 (2.1)	20.1 (13.8)	0.8	6.3	4.9 (5.4)	21.9 (13.7)	4.9	1.30	0.01	0.14	0.17
test13	859	16	9.5	7.2 (7.6)	24.1 (19.3)	3.2	24.1	18.8 (20.4)	22.2 (15.3)	19.0	1.10	0.06	0.62	0.75
test14	366	11	3.9	3.1 (3.3)	20.7 (15.9)	1.3	9.5	7.8 (8.5)	18.3 (10.7)	8.0	0.95	0.04	0.25	0.29
Average	792		7.7	5.7 (6.3)	24.6 (18.1)	2.6	20.4	15.2 (16.9)	23.6 (15.5)	15.5		0.14	1.38	1.48

Both TG and Ours1 use just type one wire on every layer.
"Ours" represents our regular approach of possibly using both types of wire on each layer.
The figures inside the brackets denote the results before the postprocessing techniques.
k denotes the number of partitions in the test case.

IV. Experimental Results

The path expansion algorithm is implemented in C++ and all the experiments are carried out on a Linux machine with 4 GB RAM and a Pentium 4 microprocessor running at 3.2 GHz. We also implemented the TG approach in [10] using C++ for comparisons. In the experiments, we assume that the slew of the source signals is 10 ps. The first three test cases (test1–test3) are provided by industry. The remaining 11 test cases are modified from the benchmarks used in the ISPD 2010 clock tree synthesis contest, with the grid set according to the conventions used in the first three industrial test cases, and with tracks added regularly across the whole chip area.

A. Comparisons with the TG Approach

Since the approach in [10] considers only one type of wire on each layer, we compare the results of our approach using just the first type of wire on every layer (without the wire replacement step and use only type one wire in all steps) with the results of [10] using the first type of wire on every layer. In these experiments, we first get the lowest achievable delays of TG empirically on all the test cases and use these delays as our delay bounds. The results are shown in Table I. Columns 3 and 7 show the total wire capacitance and the total wirelength (WL) of TG. The results of our approach are shown in columns 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 WL compared with TG, respectively (without postprocessing, the figures are 18.1% and 15.5%, respectively). The runtimes 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 runtimes of our approach are within seconds. On average, the major path expansion algorithm, the topology refinement step, and the wire replacement step take 44%, 31%, and 25% of the total runtime, respectively.

TABLE II

NONTREE ALGORITHM

Test Cases	C (pf)	WL (mm)	D (x ps)	T (s)	k k	$D_{\min}$ (y ps)	Improve ($\frac{y-x}{y}$%)
ntest1	2.4	11.1	0.45	0.1	3	0.68	33.8
ntest2	6.5	36.7	0.45	1.0	3	0.71	36.6
ntest3	5.1	25.2	0.60	3.3	3	0.51	−18.5
ntest4	2.0	11.3	1.00	1.0	3	1.26	20.8
ntest5	0.7	3.2	1.03	0.1	3	1.29	20.3
ntest6	3.5	17.6	0.66	0.5	3	1.25	47.0
ntest7	3.3	20.0	1.35	0.8	3	2.02	33.3
ntest8	2.1	12.5	1.30	0.2	3	1.98	34.4
ntest9	6.2	38.0	2.00	3.5	3	2.42	17.4
ntest10	3.6	22.1	1.80	3.1	3	2.33	22.6
ntest11	2.3	12.1	0.80	3.2	3	1.24	35.4
ntest12	0.9	5.5	1.70	0.1	3	1.65	−3.0
ntest13	3.5	20.7	1.25	0.4	3	1.35	7.2
ntest14	1.8	9.5	0.58	0.3	3	0.98	40.8
Ave.	3.1	17.5		1.3	3		23.4

k denotes the number of problematic ports in the test case.

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 I. We can see that our approach can make good use of the availability of different wire types to further reduce the capacitance.

B. Runtime Improvement with Acceleration Techniques

To demonstrate the effectiveness of our proposed partition-based path expansion algorithm, we also compare its results with the algorithm without the partition technique. The results are shown in Table III.

We can see that our proposed partition-based acceleration technique can further improve the runtime by 26.1% on average while maintaining approximately the same solution quality. The largest run time improvement is over 48% while the average improvement is 26.1%. These results clearly show the effectiveness of this technique.

TABLE III
RUNTIME COMPARISONS

Test Cases	Capacitance (pf)		Runtime (s)			Delay (ps)
	PE*	PE	PE* x_1	PE x_2	Improvement $\frac{x_1-x_2}{x_1}\%$	
test1	2.61	2.61	0.27	0.24	9.8	0.45
test2	9.68	9.67	2.72	1.39	48.9	1.15
test3	5.16	5.12	3.01	2.67	11.5	0.80
test4	4.01	4.00	2.89	1.51	48.0	1.35
test5	1.09	1.09	0.09	0.08	11.8	1.10
test6	5.73	5.73	0.68	0.48	29.2	1.25
test7	7.50	7.51	1.00	0.61	38.8	1.45
test8	4.45	4.45	0.50	0.37	25.2	1.80
test9	14.28	14.26	3.27	2.04	37.7	2.75
test10	8.60	8.59	6.92	7.15	−3.4	1.90
test11	4.93	4.92	2.95	1.76	40.5	1.05
test12	1.98	1.98	0.17	0.14	16.0	1.30
test13	7.20	7.19	0.93	0.62	33.0	1.10
test14	3.10	3.10	0.30	0.25	17.8	0.95
Ave.	5.74	5.73	1.84	1.38	26.1	

PE* denotes the original algorithm without partitioning technique.

C. Results of the Nontree Extension

To validate the effectiveness of our proposed nontree 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 are shown in Table II.

Total capacitance, total WL, delay limits, runtime, and the number of problematic ports are shown in columns 2–5, respectively. The delay limit D is got empirically for all test cases. The second last column $D_{\min}$ in Table II 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 that our nontree approach can further reduce the delay by 23.4% on average. This clearly demonstrates the effectiveness of our proposed nontree algorithm.

D. Reducing Routing Resources

We want to see what happens when routing resources are reduced in the test cases. We reduce half of the routing tracks on metal layer 4 in test2 and run our algorithm on the new test case. The delay limit of this test case is 1.15 ps. Our algorithm can satisfy the delay limit at the expense of a higher capacitance usage. The capacitance usage is 10.6 pf while it is 9.7 pf for the original test case. In general, if the test case has less routing resources, our approach can construct a clock network at the expense of higher capacitance usage.

E. Simulation Results

We validate our results using HSPICE simulation. The slew of the input signals is set to be 10 ps. The slew of a circuit is estimated using $\sqrt{(2.2RC)^2 + (S_i)^2}$ according to [5], where RC is replaced by the largest Elmore delay of the circuit. Detailed results are not shown due to page limit. The delay and slew we calculated are very close to the simulation results. For both tree and nontree structures, the correlation coefficient is over 99% between the simulated delay and calculated delay while it is over 96% between the simulated slew and calculated slew. This verifies the correctness of our method.

V. CONCLUSION

In this paper, we presented an efficient algorithm using the heap data structure to construct a postgrid clock network on reserved multilayer metal tracks. We have compared our approach with the state-of-the-art algorithm and showed that our algorithm can significantly improve over the previous work with a 24.6% reduction in wire capacitance and 23.6% reduction in WL on average while maintaining very practical runtimes. We have also extended the algorithm to allow nontree 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 and improve grid-to-port delay in real postgrid clock networks.

REFERENCES

[1] H. Bakoglu, J. Walker, and J. Meindl, "A symmetric clock-distribution tree and optimized high-speed interconnection for reduced clock skew in ULSI and WSI circuits," in *Proc. IEEE Int. Conf. Comput. Des.*, Oct. 1986, pp. 118–122.

[2] K. Boese and A. Kahng, "Zero-skew clock routing trees with minimum wirelength," in *Proc. 5th Annu. IEEE Int. ASIC Conf. Exhibit.*, Sep. 1992, pp. 17–21.

[3] P. Chan and K. Karplus, "Computing signal delay in general RC networks by tree/link partitioning," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 9, no. 8, pp. 898–902, Aug. 1990.

[4] A. Kahng, J. Cong, and G. Robins, "High-performance clock routing based on recursive geometric matching," in *Proc. 28th Des. Autom. Conf.*, 1991, pp. 322–327.

[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 Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 23, no. 4, pp. 509–516, Apr. 2004.

[6] M. Mori, H. Chen, B. Yao, and C. Cheng, "A multiple level network approach for clock skew minimization with process variations," in *Proc. Asia South Pacific Des. Autom. Conf.*, 2004, pp. 263–268.

[7] D. Pham, T. Aipperspach, D. Boerstler, M. Bolliger, R. Chaudhry, D. Cox, P. Harvey, P. M. Harvey, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Pham, J. Pille, S. Posluszny, M. Riley, D. L. Stasiak, M. Suzuoki, O. Takahashi, J. Warnock, S. Weitzel, D. Wendel, and K. Yazawa, "Overview of the architecture, circuit design, and physical implementation of a first-generation cell processor," *IEEE J. Solid-State Circuits*, vol. 41, no. 1, pp. 179–196, Jan. 2006.

[8] P. J. Restle, T. G. McNamara, D. A. Webber, P. J. Camporese, K. F. Eng, K. A. Jenkins, D. H. Allen, M. J. Rohn, M. P. Quaranta, D. W. Boerstler, C. J. Alpert, C. A. Carter, R. N. Bailey, J. G. Petrovick, B. L. Krauter, and B. D. McCredie, "A clock distribution network for microprocessors," *IEEE J. Solid-State Circuits*, vol. 36, no. 5, pp. 792–799, May 2001.

[9] R. Shelar, "An algorithm for routing with capacitance/distance constraints for clock distribution in microprocessors," in *Proc. Int. Symp. Phys. Des.*, 2009, pp. 141–148.

[10] R. Shelar, "Routing with constraints for post-grid clock distribution in microprocessors," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 29, no. 2, pp. 245–249, Feb. 2010.

[11] H. Tian, W. Tang, E. Young, and C. Sze, "Grid-to-ports clock routing for high performance microprocessor designs," in *Proc. Int. Symp. Phys. Des.*, 2011, pp. 21–28.

[12] R.-S. Tsay, "An exact zero-skew clock routing algorithm," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 12, no. 2, pp. 242–249, Feb. 1993.