MARCH: MAze Routing Under a Concurrent and Hierarchical Scheme for Buses

Jingsong Chen
CSE Department, CUHK
jschen@cse.cuhk.edu.hk

Jinwei Liu
CSE Department, CUHK
jwliu@cse.cuhk.edu.hk

Gengjie Chen
CSE Department, CUHK
gjchen@cse.cuhk.edu.hk

Dan Zheng
CSE Department, CUHK
dzheng@cse.cuhk.edu.hk

Evangeline F. Y. Young
CSE Department, CUHK
fyyoung@cse.cuhk.edu.hk

ABSTRACT

The continuous development of modern VLSI technology has brought new challenges for on-chip interconnections. Different from classic net-by-net routing, bus routing requires all the nets (bits) in the same bus to share similar or even the same topology, besides considering wire length, via count, and other design rules. In this paper, we present MARCH, an efficient maze routing method under a concurrent and hierarchical scheme for buses. In MARCH, to achieve the same topology, all the bits in a bus are routed concurrently like marching in a path. For efficiency, our method is hierarchical, consisting of a coarse-grained topology-aware path planning and a fine-grained track assignment for bits. Additionally, an effective rip-up and reroute scheme is applied to further improve the solution quality. In experimental results, MARCH significantly outperforms the first place at 2018 IC/CAD Contest in both quality and runtime.

1 INTRODUCTION

The continuous development of modern VLSI technology has brought new challenges for on-chip interconnections. In modern designs, there are buses with long wires that can introduce long wire delay. To maintain signal integrity, some post-routing optimizations such as buffer insertions are needed. However, if the bits in the same bus are routed in different topologies, it is very difficult to find places to insert buffers for different bits of the same bus in a regular manner. To resolve this problem, it is preferred to have the same routing topology among all the bits of a bus, which is different from classic net-by-net routing. In spite of reducing the size of the solution space of the routing problem to some extent, this topology constraint also makes it more difficult to efficiently allocate appropriate routing resources to each bus on multiple metal layers. Meanwhile, similar to classic net-by-net routing, solution qualities such as wire length and via count are also important metrics to optimize for bus routing. An effective bus router should provide a solution with high routing quality while maintaining the topology consistency among different bits of the bus for the benefits of signal synchronization.

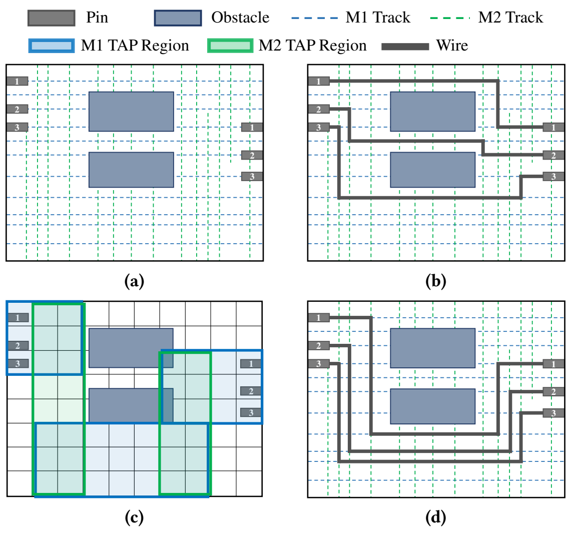


Figure 1: (a) A bus with two pins and three bits to be routed; (b) The bits routed one by one; (c) The Topology-Aware Path planning (TAP) result; (d) The Track Assignment for Bits (TAB) result.

Routing has been well studied by many previous works, including both global routing (e.g. ARCHER [1], NCTU-GR [2], and NTHU-Route [3]) and detailed routing (e.g. TritonRoute [4] and Dr. CU [5]). However, the techniques of these works can hardly be straightforwardly applied in bus routing due to the difficulty of maintaining topology consistency. If the buses are processed bit by bit (e.g. route bit 1, 2 and 3 sequentially as in Figure 1 (b)), the latter bits may lack available track segments to be routed on especially when the routing track configuration is non-uniform and complex. In the worst case, much effort of trial and error is needed until finding a feasible topology. There are some previous works handling related issues for escape routing on printed circuit board (PCB) designs, e.g. pin assignment guaranteeing routability [6], layer assignment to minimize the number of layers used [7], and an ILP-based solution [8] to solve the entire bus planning problem. However, for typical escape routing on PCB designs, it is not required to have the same topology among different bits of the same bus although the bus bits are typically routed together.

To observe the topology constraint, Streak [9] uses a representative bit to generate a set of topology candidates and then applies an ILP to select a good one. All the other bits in the bus try to follow the selected one. However, the selected topology may not be achievable

The work described in this paper was partially supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. CUHK14202218).

DAC '19, June 2–6, 2019, Las Vegas, NV, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6725-7/19/06...$15.00
https://doi.org/10.1145/3316781.3317860

due to the lack of routing resources. To handle this issue, there is a post-refinement stage in Streak where the original bus will be divided into several sub-buses and different sub-buses will have different topologies. Therefore, the techniques in Streak are not useful for this problem since a bus cannot be split into sub-buses of different topologies. Besides Streak, there is very few previous work aiming at routing buses with topology constraint.

In this paper, we present an effective bus routing method named MARCH which can efficiently solve this important problem handling practical issues like minimum spacing and minimum wire width on metal layers with irregular track structures. The objective is to finish routing all the buses, maintaining the same topology for different bits of a bus while optimizing metrics like wire length, wire segment number and compactness of the buses. The main contributions of this paper can be summarized as follows.

- We propose MARCH, which routes all the bits in a bus concurrently, instead of processing bit after bit. Such concurrency directly captures the topology consistency constraint together with other objectives (e.g. wire length) and constraints (e.g. spacing) in a correct-by-construction manner.
- A hierarchical framework is designed for the efficiency of MARCH, consisting of a Topology-Aware Path planning (TAP) and a Track Assignment for Bits (TAB). TAP is efficient as it works on a coarse-grained solution space (see Figure 1 (c)). TAB generates fine-grained routing solution, but it also gains efficiency by searching on the regions provided by TAB only (see Figure 1 (d)).
- We present an effective rip-up and reroute scheme to further improve the routing solution quality.

2 PRELIMINARIES

In the bus routing problem, buses may have multiple pins and have different wire width constraints on different metal layers. In each layer, there are routing obstacles and non-uniform routing tracks with different lengths and wire width constraints.

2.1 Evaluation Metrics

The total cost C_{total} of a bus routing solution is the summation of the failure penalty cost C_{fail}, the spacing penalty cost C_{space}, and the routing cost C_{route} of all the buses.

$$C_{total} = C_{fail} + C_{space} + C_{route} \tag{1}$$

2.1.1 Failure Penalty Cost. A successfully routed bus has to satisfy the following requirements. The routing tree of a bit needs to connect all the pins of the bit. All wires should be on-track and do not violate the width constraint of the track. More importantly, all bits are routed with the same topology satisfying the following requirements. (a) All bits should have the same number of wire segments. In Figure 2 (a), bit 1 has three wire segments, while bit 2 has only one. (b) Wires of different bits should go through the same sequence of layers. The routing in Figure 2 (b) violates this since bit 1 goes through M2, M1, and M2, while bit 2 goes through M2, M3, and M2. (c) Wires of different bits should be routed towards the same directions. In Figure 2 (c), bit 1 is routed right, down, and right, while bit 2 is routed right, up, and right. (d) Within each segment of the topology, the bit order should either be the same as or in reversed order of the bits at the pin locations. Note that on the layer with horizontal (vertical) tracks, the bit order is from bottom to top (left to right). In Figure 2 (d), the bit order of the middle

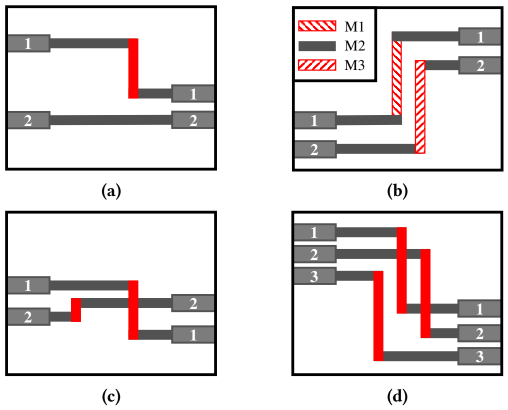


Figure 2: Four types of topology failures

wire segments is neither the same as nor in reversed order of the bits at the pin locations, so the constraint is violated.

Let N_{fail} denote the number of buses failed to be routed, then $C_{fail} = w_{fail} \cdot N_{fail}$.

2.1.2 Spacing Penalty Cost. Any pair of objects (e.g. wires of different bits, obstacles, chip boundary, etc.) on the same layer should not violate their corresponding spacing constraints. Let N_{space} denote the number of spacing violations, then $C_{space} = w_{space} \cdot N_{space}$.

2.1.3 Routing Cost. For successfully routed buses, the routing cost C_{route} can be computed based on three normalized costs: wire length cost C^b_{wire}, segment cost C^b_{seg}, and compactness cost C^b_{com}:

$$C_{route} = \sum_{\text{bus } b} w_{wire} \cdot C^b_{wire} + w_{seg} \cdot C^b_{seg} + w_{com} \cdot C^b_{com} \tag{2}$$

where C^b_{wire} is the average (among all the bits) of the total wire length of a bit divided by the half parameter wire length of the bit, and C^b_{seg} is the segment number of the topology divided by a lower bound [10]. C^b_{com} is the average (among all the wire segments) of the segment width divided by a lower bound [10], where segment width is the distance between the two outermost bits of the segment. A good routing solution should have short wire length, less segments in the topology, and more compacted width in each segment.

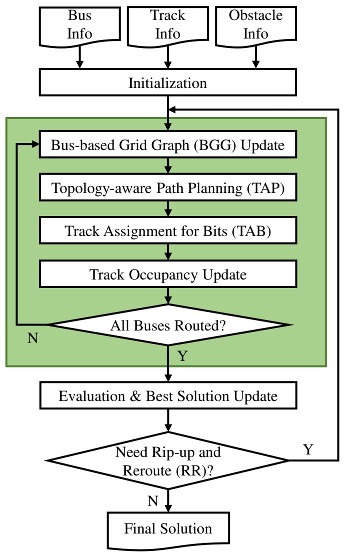
2.2 Problem Formulation

Problem 1 (Bus Routing). Given the pin information and width constraints of the buses, the tracks and their width constraints on each layer, and the obstacles, connect all the pins of each bit for all the buses and minimize the cost C_{total}.

3 ALGORITHMS

The framework of MARCH consists of two levels of loops. The inner loop routes all the buses (see the green box in Figure 3), and the outer loop is a Rip-up and Reroute (RR) scheme that tries to find a better solution. During initialization, with the loaded information of buses, tracks, and obstacles, a Bus-based Grid Graph (BGG) data structure, which will be used during the whole procedure, will first be constructed.

In the inner loop, each bus will go through fours steps: Bus-based Grid Graph (BGG) update, Topology-Aware Path planning (TAP), Track Assignment for Bits (TAB), and track occupancy update. First, BGG will be updated according to the bus to be routed so that it can provide



Bus Info
Track Info
Obstacle Info
Initialization
Bus-based Grid Graph (BGG) Update
Topology-aware Path Planning (TAP)
Track Assignment for Bits (TAB)
Track Occupancy Update
All Buses Routed?
N
Y
Evaluation & Best Solution Update
Need Rip-up and Reroute (RR)?
Y
N
Final Solution

Figure 3: Overall flow of MARCH

accurate information of the routing resources meeting the width constraint of the bus. The pins of the bus will be marked on the BGG. The bits will then be routed concurrently on BGG during TAP. In TAP, a row of grid graph cells (G-cells), named *frontline*, will propagate from the source pins to the sink pins, generating a routing path consisting of a set of rectangular regions (e.g. Figure 1 (c)) called TAP regions. The TAP regions will be used to guide TAB later on.

To check spacing violations, each track maintains its *track occupancy* which records the positions of the segments on the track that cannot be used because of the spacing violation with some neighboring obstacles or routed wires. When one tries to use a certain part of a track, an accurate number of spacing violations incurred can be obtained by checking the track occupancy. The track occupancy is maintained by a binary search tree (BST) for efficiency. After TAB, the track occupancies of all the tracks will be updated according to the track segments used by the previously routed bus.

After routing all the buses, a routing solution will be generated. An evaluator then computes C_{total} according to the metrics in Section 2.1. The best solution will be updated if a lower C_{total} is found. The evaluator results will also determine whether a RR process is needed. If so, history costs computed according to the detected spacing violations will be added to BGG, and the routing procedure will be restarted.

3.1 Bus-based Grid Graph (BGG)

BGG plays an important role in our algorithm since it provides the necessary information for cost estimation during TAP. BGG is a multi-layer grid graph with uniform G-cells. In each layer of BGG, there is an edge connecting adjacent G-cells along the layer's routing direction. The bits can be routed along the edge or switched to a neighboring G-cell on adjacent layers by via.

Each edge stores its own edge capacity and history cost. The edge capacity is computed during BGG update. It approximates the maximum number of tracks that meets the width constraint of the bus to be routed and can be used concurrently without causing any spacing violation. For instance in Figure 4, suppose the bus width is 10 and the spacing constraint is 50. To obey the spacing constraint among

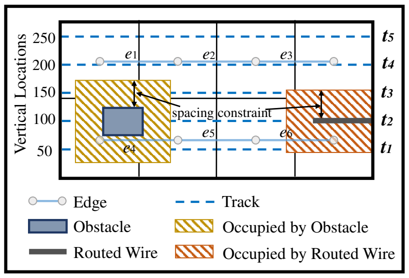


Figure 4: An example of computing edge capacity

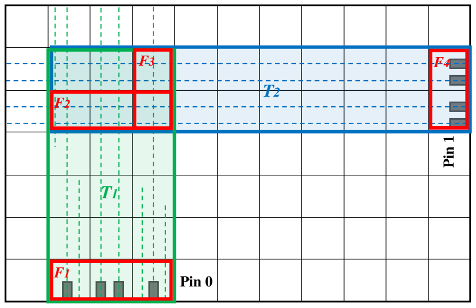


Figure 5: The TAP result of a bus

different bits of the same bus, neighboring tracks cannot be used concurrently for this example. Meanwhile, since the track with the lowest index will be checked first, the tracks t_1, t_3, and t_5 will be used to compute the edge capacity of the BGG in this example. By checking the track occupancies, the available segments on these tracks can be obtained (shown as dashed lines in Figure 4). The edge capacity of an edge in the BGG is then computed as the number of available track segments with enough length to connect the two adjacent G-cells. In Figure 4, for the edge e_1, its edge capacity is 1 since only the track t_5 (out of t_1, t_3 and t_5) has enough length to connect its two neighboring G-cells. The edge capacities of edges (e_1, e_2, ..., e_6) are (1, 2, 1, 0, 1, 0) respectively. For history cost, it can reflect the degree of routing congestion in the edge and will be accumulated when the rip-up and reroute process is performed.

3.2 Topology-Aware Path Planning (TAP)

In order to obtain a routable topology that all the bits can follow, TAP is needed to route the bits concurrently. In TAP, the frontline, which is a row of G-cells, will concurrently propagate to find a good path for all the bits. During initialization, the frontline sizes, i.e. the number of G-cells contained, are determined. Note that each bus has different frontline sizes on different layers because the track spacing on different layers varies. The frontline size for a layer considers both the bit number of the bus and the average edge capacity of that layer. The frontline sizes will all be computed at the beginning and will be changed only during rip-up and reroute.

3.2.1 A toy example. In Figure 5, the bus has two pins and four bits. The BGG has two layers where the frontline sizes of the bus are 3 and 2 respectively. The aim of TAP is to generate a "path" to connect Pin 0 marked at F_1 and Pin 1 marked at F_4. The path consists of a set of TAP regions. It can be generated as follows. Starting from position F_1, the frontline will go up along the routing direction of the metal layer until reaching F_2. At F_2, the frontline will switch to the adjacent layer and reach F_3. Finally, the frontline will go right on the adjacent layer to reach the destination F_4. This path planner result consists of two connected TAP regions T_1 and T_2, formed by propagating the frontline on one layer.

For the bus with more than two pins, similar to the classic maze routing, MARCH will first find the path between the source pin and one of the sink pins through the propagation from the source pin. The propagation will then start from the current path to connect to the next pin. This process is repeated until all the pins are connected.

3.2.2 Same Layer Propagation. When propagating the frontline on the same layer, it is necessary to know the number of tracks that can be used concurrently. Thus, the frontline will maintain a set of values, called *running capacity*, each of which is associated with a G-cell in the frontline. When propagating on the same layer, the values in the running capacity of the frontline will decrease if the capacities of the edges the frontline goes through get smaller. It can happen because some tracks are broken midway, while the new ones will not be counted since the track needs to run continuously from the beginning to the end in order to be useful. For instance, at F_1 of Figure 5, the running capacity (from the left G-cell to the right) of the frontline is (2, 2, 3). Reaching F_2, the running capacity becomes (1, 2, 1). The feasibility of the propagation can be determined by comparing the summation of the running capacity values with the bit number of the bus. The running capacity of the last frontline in a TAP region is considered as the running capacity of the whole TAP region. After switching layers, the running capacity will be reinitialized.

3.2.3 Layer Switching. Switching layers is the process that the frontline goes from one layer to its upper layer or lower layer which is usually of different routing direction. The main difficulty of switching layer is to decide whether it is safe or not to switch, or in another word, whether it will have enough routing resources without causing any spacing violation. Figure 6 (a) denotes a simple switching node which is comprised of 4×4 G-cells. In the following we will assume that the wires enter the node from below and leave from the right without loss of generality, and other situations can be handled similarly. The number on each edge denotes the edge capacity. Moreover, there are two ways of passing through a switching node, one is passing with the bit order unchanged (Figure 6 (b)), and the other is passing with the bit order flipped (Figure 6 (c)).

To decide whether passing through a switching node is safe or not, we have to compute the maximum number of bits that can pass through the node with bit order either flipped or not. Take the node in Figure 6 (a) as an example, the number of bits that can pass through the node are very different for the flipping and not flipping cases. Keeping the bit order unchanged, the node allows at most 5 bits to pass through (Figure 6 (b)), whereas by flipping the bit order, up to 8 bits can be routed through the node (Figure 6 (c)).

We propose an efficient algorithm to compute the maximum bit number that can pass through a given switching node for the two cases. For the case of keeping the bit order unchanged, we start by routing the bits from the rightmost column, and continue to the left one by one. Every time, we will use up all the resources on a lower row before using the resources on an upper row. In this way, the maximum number of routable bits can be counted.

For the case of flipping the bits, the situation is more complicated, since a bit routed earlier may sometimes block the path of the bits to be routed later due to the topology constraints. Therefore, the greedy approach used in the former case is not applicable. For example, if we try to route through the node in Figure 6 (d) greedily from the leftmost to the rightmost column, only 5 bits can be routed (Figure 6 (e)).

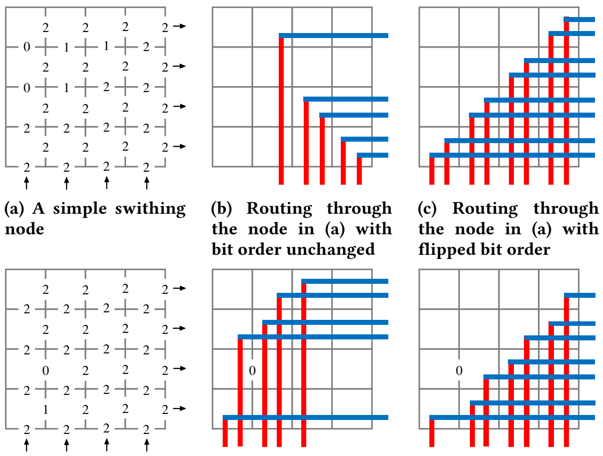

(a) A simple swithing node **(b) Routing through the node in (a) with bit order unchanged** **(c) Routing through the node in (a) with flipped bit order** **(d) A switching node with a bottleneck edge** **(e) Routing through the node in (d) with flipped bit order greedily** **(f) Routing through the node in (d) with flipped bit order by giving up one bit**

Figure 6: 4×4 switching nodes

However, if we give up 1 bit in the first column, we can end up routing 7 bits through (Figure 6 (f)). Unfortunately, it is usually unknown which bits to sacrifice would give us better result before all the bits are routed. Therefore, in our strategy, both will be tried, and the better result will be adopted.

Algorithm 1 demonstrates our methodology to compute the maximum switching node capacity with flipped bit order. Assume that the columns are indexed $1, 2, \ldots, n$ from left to right, and the rows are indexed $1, 2, \ldots, m$ from bottom to top. The function NodeCapacityFlip(i, j) returns the maximum number of bits that can pass through the node with flipped bit order and under the constraint that the bits can only enter the columns with indexes larger than i, and leave from the rows with indexes larger than j. Hence, NodeCapacityFlip(0, 0) will make use of the whole node, and returns the desired result of the maximum layer switching capacity with flipped bit order. The time complexity of the algorithm in the worst case is exponential. However, it can be finished very efficiently for most of the cases, because the runtime is linear for a congestion-free node, and will at most double when one bottleneck edge (Figure 6 (d)) exists in the node. Empirically, very few nodes (less than 5%) contain such bottleneck edges. The actual percentage depends highly on the sufficiency of the routing resources as indicated by the BGG. Besides, the algorithm improves the overall runtime of TAP, because the nodes with insufficient capacities are pruned in an early stage and will no longer be explored.

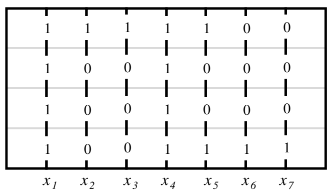
After exiting the switching nodes, the running capacity of each G-cell in the frontline will be reinitialized as the maximum number of bits that can exit from the G-cell for subsequent propagation.

3.2.4 Cost Estimation. During TAP, the actual cost that will be induced is not known yet, but it is essential to estimate the cost accurately in order to find a better path having potentially lower actual cost. The estimated cost is the summation of three values: wire length cost, segment count cost and spacing violation cost. Note that the compactness cost is not taken into consideration, because our algorithm will always propagate with the most compact frontline and will enlarge its size only when necessary.

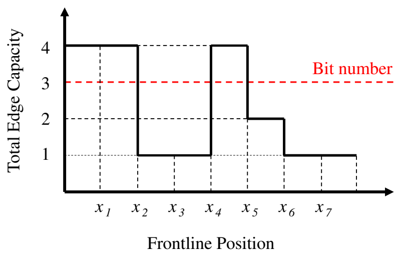
The first two costs are fairly easy to estimate. Therefore, in the following we will mainly focus on estimating the violation cost. We

Algorithm 1 Compute $m \times n$ switching node capacity with flipped bit order

```
1:  function NodeCapacityFlip($i$, $j$)
2:      if $i >= n$ or $j >= m$ then
3:          return 0
4:      $capacity \leftarrow$ The maximum number of bits that can be routed
        from column $i$ to row $j$
5:      Record capacity change
6:      if no more bits can enter column $i$ then
7:          return $capacity$ + NodeCapacityFlip($i + 1, j$)
8:      if no more bits can exit row $j$ then
9:          return $capacity$ + NodeCapacityFlip($i, j + 1$)
10:     return $capacity$ + max(NodeCapacityFlip($i + 1, j$),
        NodeCapacityFlip($i, j + 1$))
```



(a) BGG edge capacities in a TAP region



(b) Total edge capacity change when propagating from left to right

Figure 7: Spacing violation cost estimation

estimate the violation count based on the changes of the total edge capacity of the edges that the frontline passes through. More specifically, when the total edge capacity drops below the bit number of the bus, the amount of the drop below the bit number will be accumulated as violation count. For instance, consider a frontline with 4 G-cells moving from x_1 to x_7 as in Figure 7 (a), the capacity change in the frontline is illustrated in Figure 7 (b). If the bit number of the bus is 3, the total capacity drops below the bit number at x_2, x_5 and x_6 respectively. Consequently the estimated violation cost is calculated as $(3 - 1) + (3 - 2) + (2 - 1) = 4$. This approach can well avoid counting the same violation multiple times. Additionally, the history cost will be added into the total cost at last.

3.3 Track Assignment for Bits (TAB)

TAB selects a track and determines the exact positions on the track (called track segment range) to be used for each bit. However, this is a chicken-and-egg problem. On one hand, in a TAP region, the track segment range of a bit determines which track can be selected. On the other hand, the track selections also determines the exact positions on the tracks where the bits can be routed. For instance, the track selections of T_1 and T_2 determine the track segment ranges of each other in Figure 8. To handle this problem, TAB is conducted in four steps: rough track selection, track segment range estimation, exact track selection, and exact track segment range assignment.

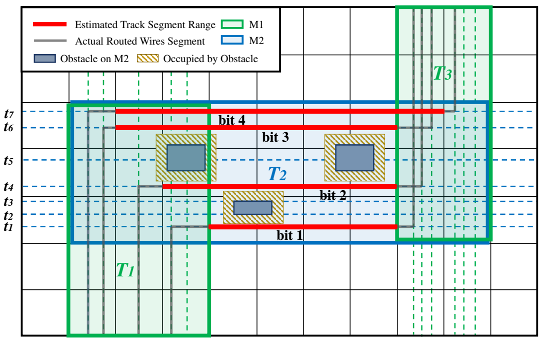


Figure 8: An example of track assignment for bits

3.3.1 *Rough Track Selection.*

First, the track for each bit is roughly selected by determining the column/row of G-cells where the bit will be routed. To perform this rough track selections for all the bits, a simple greedy method based on the running capacity of the TAP region is adopted. Take T_3 in Figure 8 as an example. Assume that its running capacity is (3, 3) (as explained in Section 3.2.2), the bits will be roughly routed, following the bit order. Therefore, bits 1~3 will be routed in the leftmost column of G-cells, while bit 4 will be routed in the next column on the right. The rough track selections in other TAP regions (e.g. T_1 and T_2) will be performed in the same way.

3.3.2 *Track Segment Range Estimation.*

For a TAP region, its estimated track selection determines the track segment range estimations of its neighboring TAP regions (e.g. T_1 and T_3 affects T_2). For bit 2 in T_2 of Figure 8, it needs to reach the middle column of G-cells in T_1 and the leftmost column in T_3. Thus, its track segment range in T_2 can be estimated conservatively marked as the red solid line in Figure 8.

3.3.3 *Exact Track Selection.*

In each TAP region, with the estimated track segment ranges, the track for each bit can be exactly selected. The track meeting the following three requirements will be selected for the bit: (1) satisfying the width constraint of the bus, (2) with long enough segment, and (3) without spacing violation.

The last two requirements are checked with the information of the estimated track segment ranges. If a track cannot meet all the requirements for a bit, the next track will be attempted. In a horizontal (vertical) TAP region, the tracks will be attempted from bottom (left) to top (right). The track selection will follow the bit order. That is, the track selection for a bit will start from the track next to the track selected for the previous bit. For T_2 in Figure 8, the track t_1 will first be selected for bit 1 because it is long enough and violation-free. For bit 2, t_4 will be selected, instead of t_2 and t_3 because of spacing violation. The other bits will be processed in the same way. Suppose the spacing between each pair of tracks in T_2 does not violate the spacing constraint, the exact track selection is shown in Figure 8.

Sometimes, there are not enough violation-free tracks in a TAP region, a violation threshold will be set which is an upper bound on the number of spacing violations caused by selecting a track. This violation threshold will be incremented gradually from zero to relax the requirement of the spacing constraint step by step until finding an enough number of tracks.

3.3.4 *Exact Track Segment Range Assignment.*

After finishing the exact track selections in all the TAP regions, the track segment range for each bit can be decided. One can see the actual routed wire segments in Figure 8.

Table 1: Detailed Results of MARCH on IC/CAD 2018 Benchmarks

	Characteristics				Metric Weights					MARCH Scores							
	bus no.	net no.	layer no.	track no.	w_{wire}	w_{seg}	w_{com}	w_{space}	w_{fail}	C_{wire}	C_{seg}	C_{com}	C_{route}	N_{space}	N_{fail}	C_{total}	Time (s)
`beta1`	34	1260	3	49209	5	1	5	8	2000	34	34	112	765	0	0	765	50
`beta2`	26	1262	3	49209	5	1	5	8	2000	26	26	85	578	0	0	578	9
`beta3`	60	665	3	22732	12	1	4	8	2000	72	62	253	1942	0	0	1942	72
`beta4`	62	698	3	22732	12	1	4	8	2000	76	71	294	2165	0	0	2165	39
`beta5`	6	1964	4	54150	8	1	5	8	2000	6	6	13	118	231	0	1966	12
`final1`	18	1032	3	81226	10	1	5	10	2000	18	22	30	356	84	0	1196	352
`final2`	70	1285	3	14209	10	1	5	10	2000	70	81	259	2071	148	0	3551	199
`final3`	47	852	4	21379	10	1	5	10	2000	47	51	558	3313	15	0	3463	133

* $C_{wire} = \sum_{bus\ b} C^b_{wire}$, $C_{seg} = \sum_{bus\ b} C^b_{seg}$, $C_{com} = \sum_{bus\ b} C^b_{com}$

Table 2: Comparison with Winners of IC/CAD 2018 Contest

	First Place					Second Place					Third Place					MARCH				
	C_{route}	C_{space}	C_{fail}	C_{total}	Time (s)	C_{route}	C_{space}	C_{fail}	C_{total}	Time (s)	C_{route}	C_{space}	C_{fail}	C_{total}	Time (s)	C_{route}	C_{space}	C_{fail}	C_{total}	Time (s)
`beta1`	689	280	0	969	3600	701	5096	0	5797	-	641	8744	4000	13385	-	765	0	0	**765**	**50**
`beta2`	515	760	0	1275	3600	563	4904	0	5467	-	484	9472	2000	11956	-	578	0	0	**578**	**9**
`beta3`	1936	0	0	**1936**	**71**	2024	0	0	2024	-	1999	1928	0	3927	-	1942	0	0	1942	72
`beta4`	2192	0	0	2192	64	2271	0	0	2271	-	2250	1048	0	3298	-	2165	0	0	**2165**	**39**
`beta5`	119	1848	0	1967	3600	95	616	2000	2711	-	98	1216	2000	3314	-	118	1848	0	**1966**	**12**
`final1`	327	830	2000	3157	3317	367	2750	2000	5117	-	252	0	10000	10252	-	356	840	0	**1196**	**352**
`final2`	1824	4500	8000	14324	3600	1890	2990	8000	12880	-	1976	6910	0	8886	-	2071	1480	0	**3551**	**199**
`final3`	2966	490	10000	13456	3600	2678	300	2000	4978	-	4238	20	24000	28258	-	3313	150	0	**3463**	**133**
Avg. Ratio				2.130	105.45				3.731					7.832					1.000	1.000

3.4 Rip-up and Reroute Scheme

Rip-up and Reroute (RR) is widely used in classic routing problems to negotiate between different nets routed in congested regions. The RR scheme in MARCH will do two things: (1) Add history cost to the edge of BGG; (2) Enlarge the frontline size when violation-free routing resources are not sufficient.

After an evaluation, the wire segments violating spacing constraints will be known. The corresponding edges of the BGG covered by these segments will be added a history cost. The history cost of an edge is accumulated by this equation: $h_{new} = \alpha \cdot n_{space} + \beta \cdot h_{old}$ where n_{space} is the number of spacing violations on this edge, and α and β are weights. When more iterations of RR are executed, the congested regions on the BGG can be eliminated gradually.

Recall that each bus has different frontline sizes for different layers. when the number of spacing violations in a TAP region of one layer is more than the bit number of the bus, this layer will be marked. In the next RR, if the same problem occurs, the frontline size of the bus on that layer will be increased by one.

4 EXPERIMENTAL RESULTS

We implement MARCH in C++. Experiments are performed on a 64-bit Linux workstation with Intel Xeon 3.4 GHz CPU and 32 GB memory. Benchmarks are from IC/CAD 2018 Bus Routing Contest [10], the statistics of which is shown in Table 1. The runtime limit of each case is 1 hour.

The detailed scores of MARCH are shown in Table 1. It can be observed that the penalty cost $C_{fail} + C_{space}$ can be reduced to a level relatively smaller than the routing cost C_{route}.

Table 2 shows the comparison with the winners of IC/CAD 2018 Contest[1]. Compared with them, MARCH not only reduces spacing violations greatly but also gets rid of all routing failures, achieving the best total cost C_{total} in seven out of eight cases. On average, C_{total} of MARCH is 2.130, 3.731, and 7.832 times better than the first, second and third place. At the same time, MARCH runs tremendously faster than the first place (with 105× speed-up on average). This indicates the effectiveness and efficiency of the concurrent and hierarchical scheme of MARCH.

[1] The scores of top 3 teams of IC/CAD 2018 Contest are provided by the contest organizer. A binary is also obtained from the first place to get its runtime information.

5 CONCLUSION

In this paper, we propose MARCH for bus routing. Compared with classic net-by-net routing methods, MARCH routes all the bits of a bus concurrently for topology consistency. MARCH also has an efficient hierarchical framework, consisting of a coarse-grained TAP and a fine-grained TAB. To reduce the routing congestion, a RR scheme is used. Experiments show that compared with the top contest teams, MARCH greatly reduces spacing violations and avoids any routing failure with competitive routing costs in a much shorter runtime.

REFERENCES

[1] M. M. Ozdal and M. D. F. Wong, "Archer: a history-driven global routing algorithm," in *Proc. ICCAD*, 2007, pp. 488–495.

[2] W.-H. Liu, W.-C. Kao, Y.-L. Li, and K.-Y. Chao, "NCTU-GR 2.0: Multithreaded collision-aware global routing with bounded-length maze routing," *IEEE TCAD*, vol. 32, no. 5, pp. 709–722, 2013.

[3] Y.-J. Chang, Y.-T. Lee, J.-R. Gao, P.-C. Wu, and T.-C. Wang, "NTHU-route 2.0: a robust global router for modern designs," *IEEE TCAD*, vol. 29, no. 12, pp. 1931–1944, 2010.

[4] A. B. Kahng, L. Wang, and B. Xu, "TritonRoute: an initial detailed router for advanced vlsi technologies," in *Proc. ICCAD*, 2018.

[5] G. Chen, C.-W. Pui, H. Li, J. Chen, B. Jiang, and E. F. Y. Young, "Detailed routing by sparse grid graph and minimum-area-captured path search," in *Proc. ASPDAC*, 2019.

[6] H. Kong, T. Yan, and M. D. F. Wong, "Optimal simultaneous pin assignment and escape routing for dense PCBs," in *Proc. ASPDAC*, 2010, pp. 275–280.

[7] Q. Ma, E. F. Y. Young, and M. D. F. Wong, "An optimal algorithm for layer assignment of bus escape routing on PCBs," in *Proc. DAC*, 2011, pp. 176–181.

[8] P.-C. Wu, Q. Ma, and M. D. F. Wong, "An ILP-based automatic bus planner for dense PCBs," in *Proc. ASPDAC*, 2013, pp. 181–186.

[9] D. Liu, B. Yu, V. Livramento, S. Chowdhury, D. Ding, H. Vo, A. Sharma, and D. Z. Pan, "Synergistic topology generation and route synthesis for on-chip performance-critical signal groups," *IEEE TCAD*, 2018.

[10] 2018 IC/CAD Contest. [Online]. Available: http://iccad-contest.org/2018/