Guaranteed Yet Hard to Find: Uncovering FPGA Routing Convergence Paradox

Shashwat Shrivastava*, Stefan Nikolić†, Sun Tanaka*, Chirag Ravishankar‡,
Dinesh Gaitonde‡, and Mirjana Stojilović*
*EPFL, †University of Novi Sad, ‡AMD

***Abstract*—Routing is one of the major challenges of FPGA compilation. PathFinder is a ubiquitous FPGA routing algorithm used in industry and academia due to its ability to adapt to arbitrary routing architectures and user circuits. However, to this day, we do not fully understand why PathFinder works so well and what its limitations are. When a circuit fails to route, it is difficult to pinpoint the problem: architecture or algorithm. Usually, in such cases, either Pathfinder is fine-tuned or routing resources are added to the architecture to improve routability, thereby ignoring the inherent inefficiencies that may exist in Pathfinder, which further prevents us from designing silicon-efficient architectures. In this work, to pinpoint the problem, we construct constrained routing problems where nets have access to limited but specific routing resources that guarantee a legal routing solution. Yet, even with a state-of-the-art implementation, PathFinder fails to find the guaranteed routing solution or any other solution, highlighting issues specific to PathFinder. The reduced search space makes the underlying behavior more accessible for analysis and reasoning, allowing us to identify inefficiencies in the current PathFinder paradigm and propose a solution to address them. We uncovered that PathFinder's greedy approach of routing individual connections yields an inefficient route tree, in terms of the total number of nodes. We then transfer this insight—through a simple yet effective algorithm—from the constrained to the standard setting, where the search space is not reduced. By constructing more efficient route trees, the routed wirelength and the number of routed connections were reduced by 6.4% and 3.6%, respectively, on average.**

I. Introduction

Routing runtime has grown significantly with the increasing size of FPGAs [1] [2] and user circuits [3]. PathFinder [4] has been the de facto standard algorithm for FPGA routing in academia [5] and industry [6] [7] for more than 30 years [8]. PathFinder's success has largely resulted from its ability to adapt to arbitrary routing architectures and user circuits. In cases, however, when it fails to route a circuit even with the best achievable placements, either PathFinder's parameters are tuned [9] or, if the FPGA is in the design phase, additional routing resources are added to increase routability [1] [2].

However, the aforementioned approaches overlook the inherent inefficiencies that may still exist in PathFinder, partly because it is difficult to pinpoint if PathFinder is indeed the culprit and, to some extent, because no framework exists to identify its fundamental problems. Even Sinan Kaptanoglu, in his endorsement of the PathFinder paper [4] as one of the most influential papers published in the International Symposium on FPGAs between 1992 and 2011, mentions the following [8]:

> *Today, we are able to use the negotiated congestion widely and very successfully. Despite that, why it works so well eludes us at a theoretical level.*

In this work, to identify if issues exist in PathFinder, we construct constrained routing problems, where nets can access limited but specific routing resources that ensure a legal routing solution. Restricting access to particular resources makes it easier for us to analyze the routes of the nets and reason about PathFinder's behavior. In contrast, a guaranteed solution prohibits us from blaming the architecture when the routing fails. Both combined, limited resources and a guaranteed solution give us the means to identify fundamental problems in PathFinder.

Upon routing in this constrained setting, we observe something surprising: instead of converging quickly with the reduced search space, not only does the runtime increase, but the router even fails to converge. Intuitively, one would expect that PathFinder would not steer a routable circuit to unroutability, but in the current paradigm, it does.

In this work, we investigate the root cause of the unintuitive convergence failure, which helps us uncover underlying inefficiencies in PathFinder. Building on this understanding, we propose a solution to address PathFinder's inefficiencies and improve performance.

The paper is organized as follows. Section II introduces the FPGA model and provides an overview of PathFinder. Section III details the methodology for constructing constrained routing settings that guarantee a legal solution. Section IV describes the experimental setup. Section V presents the unexpected results of routing in the constrained setting. Sections VI and VII analyze the causes of PathFinder's behavior. Section VIII proposes a new paradigm for PathFinder. Finally, Section IX evaluates its impact on routing in a standard setting using industry-designed benchmarks and an architecture closely resembling a state-of-the-art industrial one.

II. Preliminaries

A. FPGA Model

Fig. 1a shows a high-level view of an FPGA. The routing architecture comprises *wires* (grouped in routing channels), *switch blocks* (SBs) (in gray), and *input interconnect blocks*

This work is partially supported by the Swiss National Science Foundation (grant No. 182428) and the Ministry of Science, Technological Development and Innovation of the Republic of Serbia (grant No. 451-03-137/2025-03/ 200125 & 451-03-136/2025-03/ 200125).

2576-2621/25/$31.00 ©2025 IEEE
DOI 10.1109/FCCM62733.2025.00060

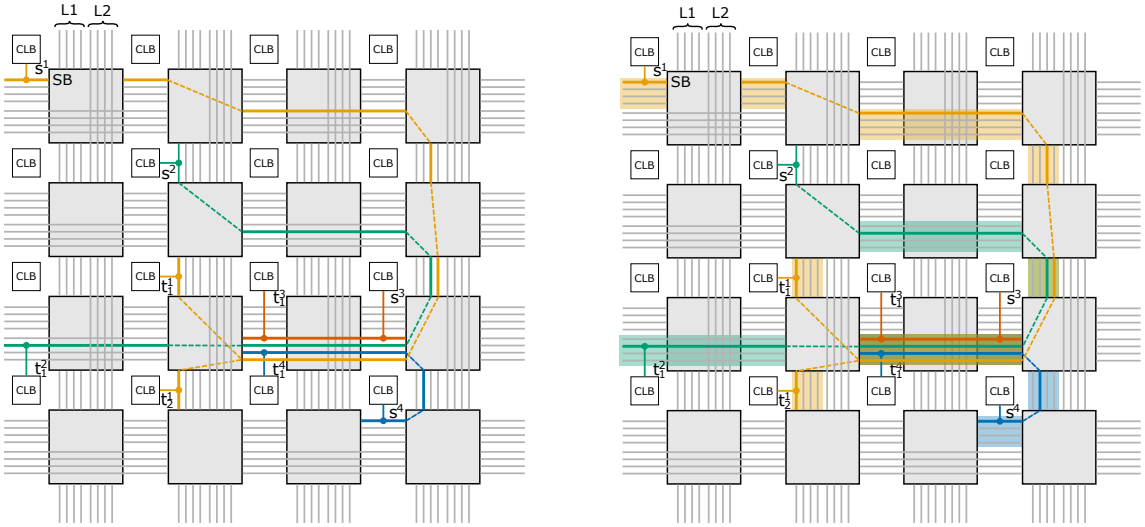

Fig. 1: (a) A high-level view of an FPGA model showing CLBs and general routing interconnect with switch blocks (in gray) and routing wires of lengths one and two, as well as four example nets and their legal routing paths. (b) Constrained routing problems are constructed from legal paths by mapping each used wire instance to a group of wires of the corresponding type. Routing is repeated, exploring the resources within the highlighted paths only. See Section III for more detail.

(IIBs) (not explicitly shown). SBs group routing multiplexers providing connectivity between wires and between *configurable logic block* (CLB) output pins and wires. Similarly, an IIB connects wires and CLB outputs to CLB inputs.

The FPGA model we use in this work is more complex and closely resembles the AMD UltraScale+ architecture. The routing channels comprise four wire *segments* corresponding to wires of lengths 1, 2, 4, and 12 [6], [10], labeled as L1, L2, L4, and L12, respectively. Each wire segment defines four wire types, one per direction: north, south, east, and west. In all FPGA tiles (logic, DSP, BRAM, I/O, etc.), each wire type has eight wire instances, except L12, which has four. For illustration purposes, only L1 and L2, with four instances each, without explicit directionality, are shown in Fig. 1a.

Contrary to modern FPGA architectures with sparse IIBs, we simplify the IIB by making it fully connected. This modification eliminates the recently analyzed routing challenges [6], [11], which PathFinder is not tuned for, allowing us to focus solely on PathFinder's intrinsic inefficiencies. Although not routing within the IIB simplifies PathFinder's task, the router still fails to converge, as we will later demonstrate—further reinforcing our conclusions.

B. PathFinder

The FPGA routing architecture can be modeled as a routing resource graph (RRG) where nodes are wires and pins, and edges are programmable connections (multiplexers) between them [4]. Let this routing graph be $G = (V, E)$, where V and E are the sets of vertices and edges, respectively. Given the circuit placement and the netlist, the goal of routing is to find an overlap-free set of trees in the routing graph. The algorithm most commonly used to perform routing is PathFinder [4].

Algorithm 1 briefly outlines the pseudo-code of a state-of-the-art *adaptive incremental router* (AIR) [12], based on PathFinder and implemented in the open-source *Verilog-to-Routing* (VTR) framework that we use in this work [5]. PathFinder iteratively routes the nets until either *congestion* (i.e., node overuse) is eliminated or the maximum number of iterations is reached (Line 1). Each node is associated with a *cost*, which dynamically increases based on the node *occupancy* (i.e., the number of nets using the node) across iterations. The cost is gradually increased to incentivize the exploration of alternative paths. As defined in Eq. 1a, the cost function of a node u has three components: the *base* cost, the *present congestion* cost, and the *historical congestion* cost. The present congestion cost in Eq. 1b reflects the node occupancy within a single PathFinder iteration, increasing proportionally with the node occupancy and scaled by a present congestion factor, *pres_fac*. On the other hand, the historical congestion cost accumulates the overuse across iterations.

$$\begin{aligned} NodeCost(u) =& PresentCost(u) \cdot \\ & HistoryCost(u) \cdot BaseCost(u), \end{aligned} \tag{1a}$$

$$PresentCost(u) = pres\_fac \cdot (1 + Occupancy(u)) \tag{1b}$$

Each net is decomposed into two-point *connections*, which are then routed sequentially (Line 2, Algorithm 2). As the routing progresses, the routes of previous connections collectively form the partial route tree of the net, as shown in Algorithm 2 (Line 5). Before routing the next connection, the nodes of the tree are inserted on the heap (Line 3) to serve as a starting point (Line 4). The heap is implemented as a priority queue-based data structure that enables efficient traversal of the FPGA routing graph by iteratively popping nodes with the lowest cost and pushing their neighbors until the target is reached.

Algorithm 1 Pathfinding Algorithm for FPGA Routing

```
for iter in max_iterations do
    for net in nets do
        tree ← RouteNet(unrouted_targets)
    end for
    for net in nets do
        RipupCongestedConnections(net)
    end for
    if not congestion_exists then break
    end if
end for
```

Algorithm 2 RouteNet Function

```
function RouteNet(unrouted_targets)
    for target in unrouted_targets do
        heap ← InitHeapFromRouteTree(target)
        path ← FindPathFromHeap(heap)
        UpdateRouteTree(path)
    end for
end function
```

III. Generating Routing Regions That Guarantee a Legal Solution

To analyze PathFinder's behavior, we design constrained routing problems with a guaranteed legal solution. This is achieved by first obtaining a legal solution and then using it to construct a constrained problem where routing is restricted to specific *routing regions*. Besides ensuring a legal solution, this approach improves pathfinding efficiency by reducing the search space and enables closer analysis of the router's behavior. The following subsection formalizes this method.

A. Routing Regions

Let N be the set of all nets to route. Each net $n \in N$ has a unique source s^n and one or more targets (sinks) T^n. A net with multiple targets is decomposed into two-point connections (s^n, t_i^n) such that $t_i^n \in T^n \ \forall \ i = 1, 2, ..., m$ where $m = |T^n|$. A legal routing path for a connection (s^n, t_i^n) can be represented as $s^n, v_1, v_2, v_3, \cdots, t_i^n$ where $v_i \in V$.

To generate routing regions that guarantee a legal solution for each net, we first perform routing in the standard setting, i.e., *unconstrained* setting. As an illustration, Fig. 1a shows legal routing paths for four nets. Then, we replace the RRG nodes in these legal paths with nodes from a new coarsened routing resource graph $G_c = (V_c, E_c)$. Here, V_c is the set of the coarsened nodes and E_c the set of edges between them. To map the routing paths from G to G_c, we define a coarsening function $C(v)$ that takes $v \in V$ and outputs $v' \in V_c$. As a result, the original routing path for connection (s^n, t_i^n) is converted to $s^n, C(v_1), C(v_2), C(v_3), \cdots, t_i^n$. The resulting *coarse paths* define the *constrained routing regions* with reduced search space for net n. This transformation is repeated for the entire netlist, resulting in a constrained routing problem suitable for analyzing PathFinder's behavior.

Specifically, in this work, a coarse node v' in G_c corresponds to one wire type (Section II-A), i.e., it contains wires (wire instances) of the same length and direction, originating in the same tile. The function $C(v)$ takes as input a wire and outputs the corresponding wire type; for example, wires $E1_0$ to $E1_7$, representing eight east-directed wires of length one, are mapped to a wire type $E1$. In Fig. 1b, shaded, colored regions correspond to coarsened nodes (for simplicity, we only show L1 and L2 segments, ignoring explicit mention of the direction). Chaining the coarsened nodes in the same sequence as the nodes in the original paths gives us the corresponding coarse paths, which we term *predefined paths*. Finally, for the constrained routing problem we obtained, we rerun PathFinder for the same netlist, this time only along the predefined paths (exploring the corresponding subset of routing resources of G). We refer to this step as *routing in the constrained setting*.

B. Handling Reconvergent Nets

After constructing the constrained routing regions and rerunning PathFinder, we encountered a class of nets that require special handling due to the challenges they present in maintaining a valid tree structure. Consider Fig. 2a, which shows a legal routing tree for a net n with two targets t_1^n and t_2^n, obtained in the unconstrained setting. Then, imagine that after coarsening, nodes $v_2 \in V$ and $v_4 \in V$ map to the same coarse node: $C(v_2) = C(v_4)$, in Fig. 2b. As a result, the coarse route tree reconverges after diverging at the source, violating the fundamental property of a tree that every node, except the root, should have exactly one parent.

When rerouted along the predefined, coarsened paths, the connections to t_1^n and t_2^n follow the orange and blue paths in Fig. 2b by construction. However, they may decide to use the *same* wire within the reconverging coarse node. Let us imagine that they use v from the coarse node $C(v_2) = C(v_4)$. In such a case, both $C(v_1)$ and $C(v_3)$ are valid parents of v, and the router must decide which to keep to preserve the tree property for the net. Discarding one of the two parents results in one of the two connections taking a path that differs from the predefined one, thereby violating the premise that a connection should route through the predefined path.

To prevent the issue, we handle the reconvergent nets differently when routing in the constrained setting: we do not construct the coarsened paths for these nets. Instead, we load the legal paths obtained from the unconstrained setting and lock the corresponding routing resources so that the problematic nets are not rerouted. Although these nets represent a small

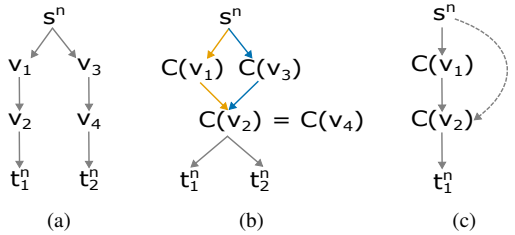

Fig. 2: (a) Routing tree of an example net n with targets t_1^n and t_2^n. (b) Coarsened routing tree showing reconvergence of n, occurring when v_2 and v_4 both map to the same coarse node. (c) A coarsened tree of an arbitrary net (not the net shown in (a)) and an edge from routing graph G between s^n and $C(v_2)$ bypassing coarse node $C(v_1)$ (see Section VI-A).

fraction of nets in our benchmarks (∼1% on average), they are typically high-fanout nets spanning large FPGA regions. Not routing them makes the routing problem somewhat easier. However, even then, we will see that PathFinder struggles to route the remaining nets. Thus, including reconvergent nets would only exacerbate the problem we uncovered.

IV. Experimental Setup

We use ISPD16 benchmarks (Table I), designed for routability-driven research [13] and placements produced by UTPlaceF [14], which won the contest for producing the best placements for these benchmarks. We further use the UltraScale architectural model from the VTR repository, which features an FPGA with 480×168 tiles, and the VTR8+ router [5] in non-timing-driven mode, to focus solely on congestion. To reduce net order dependence, we set *pres_fac* to five and keep it constant, as proposed by Zha and Li [15]. The other parameters are set to the default values of VTR8+, except for the base cost, which is made proportional to the frequency of the wire type. We set the base cost of L1, L2, and L4 to one, whereas that of L12 to two, because there are half as many L12s as other wires. The maximum number of router iterations is set to 1000. Unless mentioned otherwise, we use these parameters throughout this study. The artifacts that enable reproduction of the reported results are available in an online repository [16].

V. Routing Convergence Paradox

When PathFinder routes each net along its specified (coarsened) path, we observe something surprising and unintuitive. Although a legal solution is guaranteed to exist by how the constrained problems are constructed (the router can simply find the same route trees as in the legal solution from which the particular constrained problem was obtained), PathFinder *fails* to converge to a legal routing solution.

Table II lists the routing results for all benchmarks. All but one hit the maximum number of iterations and failed to converge. Fig. 3 shows the drop in overused nodes across iterations and compares it with the unconstrained setting on benchmark FPGA09, which is large and has a high Rent exponent (Table I). Although the constrained setting starts (at the end of the first iteration) with fewer overused nodes than the unconstrained one, the latter resolves congestion faster. In addition, the overused nodes also saturate after some iterations in the constrained setting, as shown in Fig. 4, indicating that routing hits a wall that prevents further progress. This can be seen in Table II, by comparing the number of overused nodes at the mid and the 1000^{th} iteration.

The failure of PathFinder to find a solution when it is present, and that too in a reduced search space, where faster convergence is expected, contradicts intuitive expectations.

TABLE I: Characteristics of ISPD16 contest benchmarks, including Rent's exponent p.

Benchmark	p	LUTs	FFs	RAM	DSP	Nets (K)
FPGA01	0.4	50K	55K	0	0	105
FPGA02	0.4	100K	66K	100	100	168
FPGA03	0.6	250K	170K	600	500	429
FPGA04	0.7	250K	172K	600	500	430
FPGA05	0.8	250K	174K	600	500	433
FPGA06	0.6	350K	352K	1000	600	713
FPGA07	0.7	350K	355K	1000	600	716
FPGA08	0.7	500K	216K	600	500	725
FPGA09	0.7	500K	366K	1000	600	877
FPGA10	0.6	350K	600K	1000	600	961
FPGA11	0.7	480K	363K	1000	400	851
FPGA12	0.6	500K	602K	600	500	1111
Average	0.625	327.5K	291K	675	450	626.5

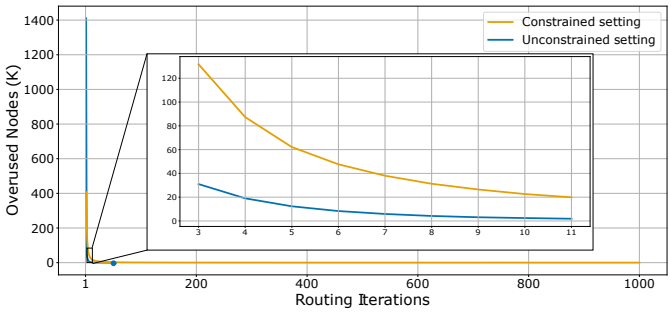

Fig. 3: Comparing the drop of overused nodes in the constrained and unconstrained setting for benchmark FPGA09. The inset zooms in on iterations three to 11.

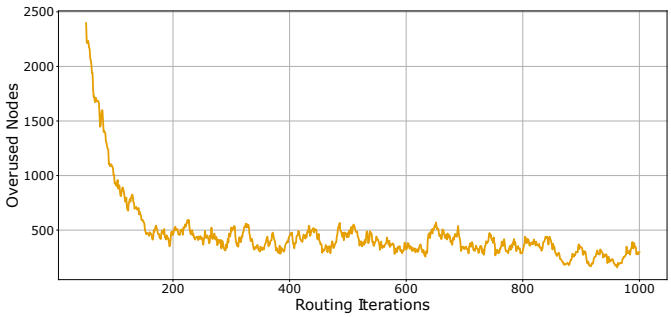

Fig. 4: Overused nodes saturating after 200 iterations in the constrained setting for benchmark FPGA09.

VI. Tuning Does Not Help

A standard approach to resolving routing convergence issues is to tune the router's parameters [9]; we present a thorough analysis of the impact of changing the values of various router parameters in Section VI-B. However, one of the major benefits of constraining the search space in which the route trees are constructed is that it allows us to inspect failures, which is what we do first. We reiterate that the constrained routing problems are designed to ensure the existence of a legal routing solution. Hence, there is no need to relax the constraints, which would be somewhat similar to progressively increasing the bounding box within which a net is allowed to explore the routing resources [12] or adding new routing resources as is commonly done in the FPGA design phase. Furthermore, doing so would only hide the inherent limitations of PathFinder, and our main goal is precisely to expose them.

TABLE II: The total iterations to route in the constrained setting are compared to the standard setting, where all benchmarks converge. Overused nodes at the end of the middle and final iterations are noted for the constrained setting, with and without multiple random sink shuffles (Section VIII). The **Min** column lists the lowest overused nodes reached across all routing iterations. Benchmarks requiring 1000 iterations indicate convergence failure. With multiple random sink shuffles, routing improves, with five benchmarks converging and others reaching single-digit overused nodes.

Benchmark	Standard setting	Constrained setting				Constrained setting with multiple random sink shuffles			
	Total iterations	Overused nodes			Total iterations	Overused nodes			Total iterations
		Mid iter	Last iter	Min		Mid iter	Last iter	Min	
FPGA01	3	3	0	0	61	29	0	0	**30**
FPGA02	3	7	5	2	1000	35	0	0	**79**
FPGA03	4	48	18	6	1000	10	0	0	**229**
FPGA04	13	118	88	65	1000	18	12	3	1000
FPGA05	108	268	216	108	1000	36	7	3	1000
FPGA06	16	27	17	9	1000	39	0	0	**210**
FPGA07	46	446	295	168	1000	7	16	1	1000
FPGA08	9	451	412	245	1000	29	15	5	1000
FPGA09	47	473	298	158	1000	35	6	3	1000
FPGA10	24	88	92	17	1000	3	0	0	**806**
FPGA11	80	420	300	165	1000	44	11	3	1000
FPGA12	42	129	209	40	1000	10	1	1	1000

A. Enforcing Strict Node Hopping Sequence

A deeper analysis shows that some routed connections, even within the constrained region, do not follow the exact node sequence of their predefined coarsened paths. In particular, this may happen when two nonadjacent nodes of a predefined path have an edge between them in the routing graph G. Such edges can result in connections bypassing some intermediate nodes while routing and, in doing so, deviating from their predefined paths. Fig. 2c illustrates one such case.

To avoid nets bypassing coarse nodes and deviating from their predefined paths, we introduce a mechanism to enforce the node hopping sequence determined by the coarsened predefined path. For any node $v \in V$, we define $hop(n, t_i^n, v)$ for a net n and its target t_i^n. The function $hop(n, t_i^n, v)$ returns the distance, measured in coarse edges, between the source s^n and the coarse node $C(v)$ (illustrated in Fig. 6a). When a node is popped from the heap, only its child nodes corresponding to the next coarse node in the predefined path are pushed. To implement this mechanism, we use the following equation,

$$hop(n, t_i, u) = hop(n, t_i, v) + 1, \tag{2}$$

where v is the popped node, and $u \in V$ is the pushed node. Node u is pushed onto the heap only if it satisfies Eq. 2. This mechanism significantly reduces the number of overused nodes after 1000 routing iterations: by 57.8% on average. However, only the smallest benchmark (FPGA01) converges.

B. Tuning PathFinder Parameters

After fixing the node hopping sequence, we employed the standard strategy to address convergence issues: PathFinder tuning. We tune parameters that solely focus on resolving congestion. VTR's router AIR [12] reroutes only congested net sub-trees to reduce computational effort; by default, AIR completely rips up nets if their fanout is below 16. As we had already significantly reduced the path search space, we ran the router in high-effort mode to reduce congestion by increasing this threshold to 32. The average minimum number of overused nodes decreased by 10% compared to completely ripping up nets with fanout below 16. However, no new benchmarks converged.

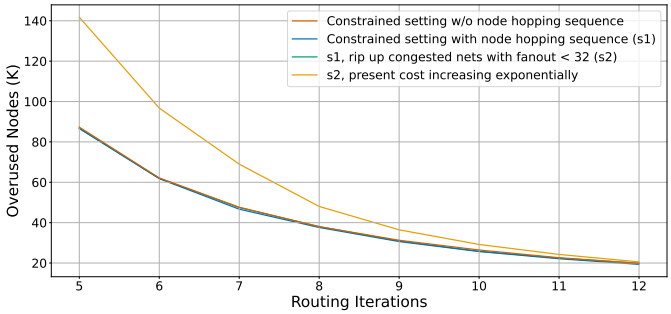

Fig. 5: Almost no improvement across various PathFinder configurations for benchmark FPGA09. The configurations are labeled as si ($i = 1, 2$). What follows si is the parameter that changed in the configuration of si.

We also experimented with exponential *pres_fac* increments (*initial_pres_fac* = 0.5 and *mult_pres_fac* =1.3), as done by default in VTR, instead of a fixed value of five. We combined it with completely ripping up nets with fanout below 32, as this slightly improved the overused nodes at the end of 1000 iterations. This configuration only enabled the FPGA01 benchmark to converge after six additional iterations. In addition, the average minimum number of overused nodes increased by 58.7% compared to completely ripping up nets with fanout below 32 and fixed *pres_fac*. Fig. 5 shows that no significant improvements arose from various PathFinder configurations. However, we determine that the best setting for routing in the constrained regions is to enforce node hopping sequence, rip up nets with fanout below 32, and keep a fixed *pres_fac* of five.

VII. Where Is the Problem?

Before we describe the main problem, we define three metrics for any arbitrary coarse node $v' \in V_c$: $ld(v')$, $ld_{init}(v')$, $cap(v')$. Here, $ld(v')$ represents the *load* of v', corresponding to the number of nodes from G within v' that are used by at least one net, as illustrated in Fig. 6. If more nets passed through the coarse node, then its load would increase accordingly. Similarly, $ld_{init}(v')$ corresponds to the number of nodes from G within v' that were used by at least one net in the legal routing solution from which the constrained routing paths were constructed (Fig. 6a). Finally, $cap(v')$ is the *capacity* of v', defined as the number of nodes from G that exist within v'.

A. Problem Identification

Each coarse node v' in a constrained region constructed from a legal routing solution initially satisfies the following:

$$ld_{init}(v') \leq cap(v'). \tag{3}$$

During routing along the constrained regions, we observe that for some coarse nodes, the value of $ld()$ could increase above $ld_{init}()$. This occurs because the connections of a net end up using more nodes within a coarse node than in the legal routing solution. For example, if in the legal solution, multiple connections of a net used only one node within a coarse node, but during routing along the constrained path, the connections branched somewhere higher up in the partial route tree, the net will use multiple nodes within this coarse node, driving up its $ld()$. The described behavior can be seen in Fig. 6b.

Due to branch point movement, a coarse node v' could be in three different states depending on $ld(v')$: ① *initial load state*, where $ld(v')$ is the same as $ld_{init}(v')$ (see Eq. 4); ② *load increase state*, where $ld(v')$ increases above $ld_{init}(v')$ but is less than or equal to $cap(v')$ (see Eq. 5); and ③ *overload state*, where $ld(v')$ increases beyond $cap(v')$ (see Eq. 6).

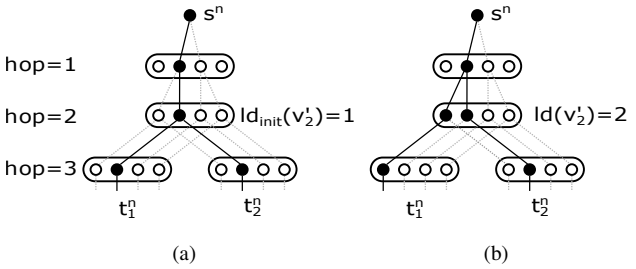

Fig. 6: Illustration of load increase due to branch point movement. All available paths from the source of a net n to its two targets are shown within the constrained routing region. Small circles depict nodes from G while the rounded rectangles depict coarse nodes from G_c, each having a capacity of four. The original route tree used to construct the constrained paths is shown in subfigure (a), whereas a route tree of the same net during routing in the constrained setting is depicted in subfigure (b). The load of v'_2 increased by one because the connection to target t_2^n branched from the path taken by the previously routed sink t_1^n at a node closer to the root.

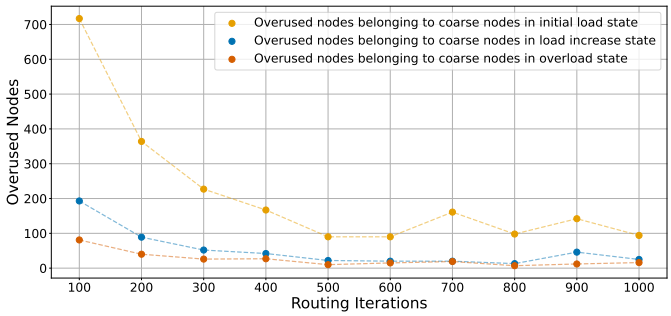

Fig. 7: Overused nodes in G across routing iterations for FPGA09, categorized based on the state of the coarse nodes they belong to (initial load, load increase, or overload).

$$\text{Initial load state: } ld(v') = ld_{init}(v'), \tag{4}$$

$$\text{Load increase state: } ld_{init}(v') < ld(v') \leq cap(v'), \tag{5}$$

$$\text{Overload state: } ld(v') > cap(v'). \tag{6}$$

For benchmark FPGA09, Fig. 7 shows the overused (congested) nodes in G across routing iterations, categorized by the state of the coarse node to which they belong. We can see that some overused nodes belong to coarse nodes in the *overload* state, which creates a routing bottleneck that cannot be resolved without a decrease in load.

On the other hand, one could expect that after some routing iterations, there would be no more overused nodes within the coarse nodes in *initial load state*, as their disappearance does not require any load reduction. However, those nodes also remained overused until the last iteration. Moreover, even after the final iteration, they were still the most numerous. This suggests that the increase in the load of the coarse nodes may not be the sole contributing factor to the convergence problem. Nevertheless, it is also possible that the overused nodes belonging to the *initial load state* category of coarse nodes are a consequence of congestion originating in those coarse nodes in *increase load state* and the *overload state*.

To determine whether the load increase is the sole cause of the problem for the benchmark under consideration, we design an experiment where we sample the nets for which the branch point moves in the first two iterations. Then, we extract the route trees of these nets from the legal routing solution that was used to construct the constrained routing region. To isolate the effect of routing these nets—15% of all nets of FPGA09—we use the extracted legal route trees and lock their routing resources, rather than routing them from scratch along the constrained path. After performing this importing and locking, the remaining nets of FPGA09 were successfully routed in 80 iterations. This finding indicates that the nets for which the branch point moves result in convergence failure.

VIII. How to Fix the Load Increase?

The fundamental problem with branch node movement is that the route tree of a net ends up using more nodes than it did in the legal routing solution from which the constrained

paths were obtained. An inefficient route tree is constructed primarily because PathFinder routes the connections sequentially in the current paradigm. As a result, the connections that route initially are oblivious of the path that would be better suited for connections that route later.

To break this dependence on sink order and enable construction of more optimized route trees, we propose a straightforward yet effective solution: routing each net with multiple permutations of its sink order—ideally, all of them. Each permutation of sinks of the given net results in a different route tree, out of which we select the one with the fewest nodes, before proceeding to route the following net. The superexponential growth of the number of permutations with respect to net fanout makes exhaustive exploration of all sink permutations infeasible for most nets; however, for those with five or fewer sinks, it is feasible.

To avoid the scalability issue with sink permutation, we approximate the method by randomly shuffling the sinks multiple times to get a subset of permutations (Line 3 in Algorithm 3). Then, out of the thus constructed trees, we select the one with the fewest nodes (see Line 11 in Algorithm 3). We break ties using the minimal total congestion cost, calculated by summing the individual congestion cost of the nodes (Eq. 1a).

Algorithm 3 PathFinder with Multiple Random Sink Shuffles

```
1:  for iter in max_iterations do
2:      for net in nets do
3:          multiple_sink_orders ← GenerateRandomShuffles(sinks)
4:          map = {}
5:          for sink_order in multiple_sink_orders do
6:              tree ← RouteNet(sink_order)
7:              total_nodes ← TotalNodesInTree(tree)
8:              congestion_cost ← TreeCost(tree)
9:              map[total_nodes, congestion_cost] = tree
10:         end for
11:         tree ← TreeSelection(map)
12:     end for
13:     for net in nets do
14:         RipupCongestedConnections(net)
15:     end for
16: end for
```

Table II shows the convergence improvement originating from routing with multiple random sink shuffles. With 300 shuffles for the first nine iterations and 48 for the later ones (determined empirically to keep runtime from exploding), we observed that six benchmarks converged. For others, the minimum number of overused nodes dropped to a single digit—a significant reduction from routing with a single default sink order. To push the other benchmarks to convergence, we could increase the number of sink shuffles; however, even with a high number of random shuffles, landing on a sink order that minimizes the route tree size is still not guaranteed, given the probabilistic nature of the random shuffle.

IX. Application in Standard Setting

Having identified an inefficiency in PathFinder and proposed a solution, we next test the impact of this new routing paradigm in the standard setting, where the search space of nets is unconstrained. The results of applying 48 random shuffles to each net and picking the smallest encountered tree are in Table III. As we can see, minimizing tree size can significantly improve routing quality. In particular, after using sink shuffling to optimize the tree size, the total wirelength dropped by 6.4% on average, while the number of segments used by the nets decreased by 7.9%. We note that 48 shuffles cannot cover all possibilities for nets with fanout over four. Hence, it is likely that significantly better optimization can be obtained by replacing the tree construction algorithm altogether and directly searching for minimum trees. Nevertheless, shuffling sinks to generate varied route trees and selecting the best one is a general approach that can optimize metrics beyond wirelength and congestion, such as timing and power. A less naive implementation could better explore permutations. We turn to this next.

1) Parallelization Potential: Given that our primary goal in this work was to understand the limitations of PathFinder and how they could be overcome, we opted for the most straightforward implementation of the sink-permutation algorithm, which is sequential. Of course, increasing the runtime of the routing process severalfold would be prohibitive; therefore, it is essential to note that exploring different permutations of sinks for a particular net creates completely independent pathfinding problems and can hence be fully parallelized. Since it has been shown that parallelizing PathFinder across different nets results in limited runtime reductions on large and complex circuits [6], we believe that employing the available hardware threads—up to 192 on the state-of-the-art CPUs [17]—for exploring different sink permutations is a good way to leverage modern highly-parallel hardware to speed up routing. The primary way in which routing can be sped up by exploring different sink permutations is that choosing the smallest encountered tree leads to a quicker reduction of congestion, which in turn results in fewer connection reroutes, as shown in Table III.

With the most straightforward approach, a net can be considered routed only after the slowest thread completes the search for its tree. To model the runtime that would be obtained in this scenario, we collect the number of heap operations required to get the tree for the given net. Then, summing the largest of these numbers across all nets and iterations gives us the total runtime that would be required if one were always to wait for the slowest thread to finish. This is reported in the middle section of Table III. We believe that investing 30% more runtime to obtain significant wirelength reductions is a very good trade-off, as it could, for instance, enable the utilization of FPGAs with fewer routing resources.

Moreover, by adopting some filters, permutations that are not promising can be terminated early without much loss to the quality of the results. Finally, if routing is considered not in isolation but as part of the entire CAD flow, convergence and timing closure with the given placement, chances of which are boosted by the appropriate minimization of the route trees, removes the need to revisit placement or apply any manual modifications, thus potentially greatly reducing the overall runtime on large and complex circuits. This, however, goes beyond the scope of the present work.

TABLE III: Comparing routing in the standard setting with single sink order and multiple sink orders generated by random sink shuffling. In the latter case, trees are selected using two different selection strategies: the minimum number of nodes and the minimum number of heap pushes. *CR* is the total number of connections routed, *Avg. seg.* is the number of average segments used per net, *TWL* is the total wirelength, whereas *pops* and *pushes* are the corresponding heap operations. The numbers adjacent to arrows are normalized with respect to routing in the standard setting with single sink order.

Benchmark	Standard setting with single sink order					Standard setting with multiple random sink shuffle									
						Tree with minimum number of nodes					Tree with minimum heap pushes				
	CR (K)	Avg. seg.	TWL	Pops (K)	Pushes (K)	CR	Avg. seg.	TWL	Pops	Pushes	CR	Avg. seg.	TWL	Pops	Pushes
FPGA01	418	3.2	668	31,806	118,526	1.6% ↓	7.0% ↓	6.8% ↓	1.1× ↑	1.2× ↑	0.4% ↓	3.3% ↓	3.1% ↓	0.9× ↓	0.8× ↓
FPGA02	743	3.4	1,167	55,345	210,177	1.5% ↓	7.6% ↓	6.8% ↓	1.1× ↑	1.2× ↑	0.4% ↓	3.8% ↓	3.5% ↓	0.9× ↓	0.8× ↓
FPGA03	2,149	4.6	5,222	201,023	928,448	1.2% ↓	7.3% ↓	5.6% ↓	1.2× ↑	1.3× ↑	0.3% ↓	3.4% ↓	2.5% ↓	0.9× ↓	0.8× ↓
FPGA04	2,270	6.3	8,529	260,935	1,453,301	2.5% ↓	8.6% ↓	6.4% ↓	1.3× ↑	1.4× ↑	1.4% ↓	3.8% ↓	2.5% ↓	0.8× ↓	0.7× ↓
FPGA05	6,850	9.7	14,810	57,214,347	189,342,499	31.6% ↓	9.0% ↓	7.4% ↓	1.2× ↑	1.2× ↑	23.6% ↓	3.4% ↓	3.3% ↓	0.3× ↓	0.3× ↓
FPGA06	3,297	4.9	9,563	340,699	1,683,991	1.7% ↓	7.0% ↓	5.4% ↓	1.3× ↑	1.4× ↑	0.7% ↓	3.0% ↓	1.9% ↓	0.8× ↓	0.8× ↓
FPGA07	4,077	6.5	15,051	944,445	6,074,410	7.4% ↓	7.9% ↓	6.0% ↓	1.3× ↑	1.3× ↑	4.3% ↓	3.1% ↓	2.1% ↓	0.5× ↓	0.5× ↓
FPGA08	3,844	5.6	12,499	367,447	1,951,403	1.5% ↓	9.2% ↓	6.5% ↓	1.3× ↑	1.4× ↑	0.6% ↓	4.1% ↓	2.8% ↓	0.8× ↓	0.7× ↓
FPGA09	5,384	6.4	18,409	952,470	6,545,285	12.0% ↓	10.1% ↓	8.1% ↓	1.0× ↑	1.1× ↑	7.8% ↓	4.3% ↓	3.5% ↓	0.5× ↓	0.4× ↓
FPGA10	3,631	4.0	11,403	387,805	2,011,013	2.7% ↓	5.5% ↓	4.5% ↓	1.1× ↑	1.2× ↑	1.4% ↓	2.4% ↓	1.9% ↓	0.8× ↓	0.8× ↓
FPGA11	6,220	5.8	15,625	5,995,218	30,288,428	14.0% ↓	9.3% ↓	7.5% ↓	1.0× ↑	1.0× ↑	10.5% ↓	3.9% ↓	2.9% ↓	0.3× ↓	0.3× ↓
FPGA12	4,493	3.3	10,196	387,295	1,871,566	2.7% ↓	7.6% ↓	6.3% ↓	1.2× ↑	1.3× ↑	1.3% ↓	3.4% ↓	2.4% ↓	0.8× ↓	0.7× ↓
Geomean	2,855	5.0	7,486	533,840	2,598,553	**3.6**% ↓	**7.9**% ↓	**6.4**% ↓	1.2× ↑	1.3× ↑	**1.6**% ↓	**3.5**% ↓	**2.6**% ↓	**0.6**× ↓	**0.6**× ↓

2) Early Stopping: One easy way to apply early termination is to choose the first tree found by any thread. While this does not guarantee selecting the smallest explored tree, the heuristic is not entirely arbitrary. Assuming A* exhibits the same effective branching factor for all permutations, fewer heap operations should lead to shorter paths forming the tree. The right section of Table III shows the case where we always select the tree with the fewest heap pushes, which would roughly correspond to the first found. This reduces the total number of heap operations by 40% on average, while still maintaining tangible improvements to wirelength, although smaller than when the smallest trees are selected. Thus, there is ample room to trade off runtime and quality of results by choosing different trees, which is something we plan to explore more thoroughly in future work.

X. Related Work

Over the years, various PathFinder enhancements have been proposed to improve its runtime or timing optimization. However, to our knowledge, only two works have specifically highlighted inherent inefficiencies of PathFinder in the context of FPGA routing: Rubin and DeHon identified significant variations in critical path delay caused by small perturbations in the net routing order, demonstrating an inherent sensitivity of the algorithm to it [18], whereas Zha and Li showed that keeping the present congestion factor constant across routing iterations reduces these variations [15].

CRoute [19] and RWRoute [10] investigated cost biasing to encourage node sharing among net connections. Yet, their work lacks a detailed analysis of minimizing tree size or transitioning from a greedy connection to holistic net routing.

Our work parallels PEKO [20], which revealed suboptimalities of ASIC placers of the time, by carefully constructing circuits with known optimum placements. Similarly, we present a framework to analyze PathFinder that leverages constrained routing problems with a known legal routing. Unlike PEKO, we extract these problems from real-world circuits instead of generating specific synthetic benchmarks.

Perhaps the most thorough theoretical analysis of PathFinder's congestion negotiation mechanism was given by Roy and Markov [21] in the context of ASIC global routing. By drawing parallels between congestion negotiation and Lagrangian relaxation, they proposed a more general formulation that allows assigning different weights to different nets. They also noted that optimal precomputed Steiner-minimal trees that are oblivious to congestion are inferior to the approximate Steiner-minimal trees computed by PathFinder through connection-based routing with node sharing, which take congestion into account. In this work, we demonstrate that while congestion awareness is essential, it is not sufficient, as poor tree minimization can still prevent the router from converging on a legal solution when routing resources are limited. The issue is much more important in FPGAs because to avoid such failures, architects need to add expensive routing resources not only where the particular user circuit requires them, but uniformly across the chip.

XI. Conclusions

This paper aims to highlight inefficiencies in PathFinder that have persisted for nearly three decades. We propose shifting the paradigm from greedily routing individual connections to constructing efficient route trees holistically. Our results, obtained through a simple algorithm, demonstrate that this new routing paradigm can reduce wirelength and heap operations by employing tailored tree selection strategies. Our algorithm appears simple, yet it is effective and generalizable to various user requirements by adjusting the tree-selection strategy, where each candidate tree can be processed by tree analysis algorithms of arbitrary complexity. These findings open up avenues for further research to enhance PathFinder, thereby bridging the gap between its current capabilities and its full potential.

REFERENCES

[1] B. Gaide, D. Gaitonde, C. Ravishankar, and T. Bauer, "Xilinx adaptive compute acceleration platform: Versal™ architecture," in *Proceedings of the 27th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Seaside, CA, USA: ACM, Feb. 2019, pp. 84–93.

[2] J. Chromczak, M. Wheeler, C. Chiasson, D. How, M. Langhammer, T. Vanderhoek, G. Zgheib, and I. Ganusov, "Architectural enhancements in Intel® Agilex™ FPGAs," in *Proceedings of the 28th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Seaside, CA, USA: ACM, Feb. 2020, pp. 140–49.

[3] S. Hong, S. Moon, J. Kim, S. Lee, M. Kim, D. Lee, and J.-Y. Kim, "DFX: A low-latency multi-FPGA appliance for accelerating transformer-based text generation," in *55th IEEE/ACM International Symposium on Microarchitecture*. Cupertino, CA, USA: IEEE, Oct. 2022, pp. 616–30.

[4] L. McMurchie and C. Ebeling, "PathFinder: A negotiation-based performance-driven router for FPGAs," in *Proceedings of the 3rd ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Napa Valley, CA, USA: ACM, Feb. 1995, pp. 111–17.

[5] K. E. Murray, O. Petelin, S. Zhong, J. M. Wang, M. Eldafrawy, J.-P. Legault, E. Sha, A. G. Graham, J. Wu, M. J. Walker, H. Zeng, P. Patros, J. Luu, K. B. Kent, and V. Betz, "VTR 8: High-performance CAD and customizable FPGA architecture modelling," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 13, no. 2, pp. 1–60, May 2020.

[6] S. Shrivastava, S. Nikolić, C. Ravishankar, D. Gaitonde, and M. Stojilović, "IIBLAST: Speeding up commercial FPGA routing by decoupling and mitigating the intra-CLB bottleneck," in *Proceedings of the 42nd International Conference on Computer-Aided Design*. San Francisco, CA, USA: IEEE, Oct. 2023, pp. 1–9.

[7] E. Vansteenkiste, A. Kaviani, and H. Fraisse, "Analyzing the divide between FPGA academic and commercial results," in *Proceedings of the 2015 International Conference on Field Programmable Technology*. Queenstown, New Zealand: IEEE, Jan. 2015, pp. 96–103.

[8] S. Kaptanoglu, "PathFinder: A negotiation-based performance-driven router for FPGAs," *FPGA and Reconfigurable Computing Hall-of-Fame Endorsement*, 2012.

[9] N. Kapre, H. Ng, K. Teo, and J. Naude, "Intime: A machine learning approach for efficient selection of FPGA CAD tool parameters," in *Proceedings of the 23rd ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Monterey, California, USA: ACM, Feb. 2015, pp. 23–26.

[10] Y. Zhou, P. Maidee, C. Lavin, A. Kaviani, and D. Stroobandt, "RWRoute: An open-source timing-driven router for commercial FPGAs," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 15, no. 1, pp. 1–27, 2021.

[11] A. Mohaghegh and V. Betz, "Tear down the wall: Unified and efficient intra- and inter-cluster routing for FPGAs," in *Proceedings of the 33rd International Conference on Field Programmable Logic and Applications*. Gothenburg, Sweden: IEEE, Apr. 2023, pp. 130–36.

[12] K. E. Murray, S. Zhong, and V. Betz, "AIR: A fast but lazy timing-driven FPGA router," in *Proceedings of the 25th Asia and South Pacific Design Automation Conference*. Beijing, China: IEEE, Jan. 2020, pp. 338–44.

[13] International Symposium on Physical Design (ISPD), "Routability-driven FPGA placement contest," https://www.ispd.cc/contests/16/ispd2016_contest.html, 2016, retrieved Aug. 2022.

[14] W. Li, S. Dhar, and D. Z. Pan, "Placements for ISPD16 benchmarks," http://wuxili.net/project/utplacef/, 2016, retrieved Sep. 2022.

[15] Y. Zha and J. Li, "Revisiting PathFinder routing algorithm," in *Proceedings of the 30th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Seaside, CA, USA: ACM, Feb. 2022, pp. 24–34.

[16] S. Shrivastava, S. Nikolić, S. Tanaka, C. Ravishankar, D. Gaitonde, and M. Stojilović, "Guaranteed yet hard to find—Artifacts," Available on: https://doi.org/10.5281/zenodo.15024666, 2025.

[17] AMD, "AMD Ryzen™ Threadripper™ processors," https://www.amd.com/en/products/processors/workstations/ryzen-threadripper.html#specifications, 2025, retrieved Jan. 2025.

[18] R. Y. Rubin and A. M. DeHon, "Timing-driven PathFinder pathology and remediation: Quantifying and reducing delay noise in VPR-PathFinder," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Monterey, California, USA: ACM, Feb. 2011, pp. 173–76.

[19] D. Vercruyce, E. Vansteenkiste, and D. Stroobandt, "CRoute: A fast high-quality timing-driven connection-based FPGA router," in *Proceedings of 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines*. San Diego, CA, USA: IEEE, Apr. 2019, pp. 53–60.

[20] C.-C. Chang, J. Cong, and M. Xie, "Optimality and scalability study of existing placement algorithms," in *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*. Kitakyushu, Japan: ACM, Jan. 2003, pp. 621–27.

[21] J. A. Roy and I. L. Markov, "High-performance routing at the nanometer scale," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 6, pp. 1066–77, May 2008.