2021 31st International Conference on Field-Programmable Logic and Applications (FPL) | 978-1-6654-3759-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/FPL53798.2021.00044

Turning PathFinder Upside-Down: Exploring FPGA Switch-Blocks by Negotiating Switch Presence

Stefan Nikolić and Paolo Ienne
École Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences, 1015 Lausanne, Switzerland
{stefan.nikolic, paolo.ienne}@epfl.ch

***Abstract*—Automated switch-block exploration gains in importance as technology scaling brings more emphasis on the physical constraints, making it insufficient to rely on abstract measures of routability alone. In this work, we take an approach that significantly differs from the previously used ones, relying mostly on general optimization methods: we essentially let the router itself design the switch-pattern. Of course, letting the router make arbitrary choices would be rather ineffective, as there would be nothing to prevent it from spreading routes over many different switches, making it difficult to understand if a particular one was used because it is essential for proper implementation of a given circuit, or simply due to some local, largely irrelevant decision. Instead, we change the method of node pricing in a negotiated-congestion router, by applying the same principles in the opposite direction, to make it reach a consensus on switches that are worthy of being included in the final switch-pattern. With this, we obtained a pattern that outperforms the one reached through simulated annealing optimization by 10.7% in terms of average routed critical path delay and uses less than half the number of switches, without compromising routability.**

I. Introduction

When FPGA architecture research started to develop, considerable attention was given to the design of the switch-patterns used in the programmable interconnect [1], [2], [3]. Typically, the goal was to maximize some metric of routability while minimizing the number of switches used. Most of the successful switch-patterns were invented and their effectiveness confirmed either experimentally [1], or by proving their optimality with respect to some proposed definition of what optimality could actually mean [3]. Since at the time the delays of connections implemented by the FPGA depended mostly on the number of hops through the switch-blocks [4], with some notable exceptions [5], little care was paid to wiring inside the switch-block itself. Over time, a few switch-patterns emerged as dominant and further research in the area subsided.

The detailed report of the routing architecture modifications in the recent Agilex FPGA family [6] leaves an impression that something of consequence is happening because of technology scaling and that the usual assumptions about switch-patterns should be revisited. This requires going beyond reassessing which of the major pattern families [7] or their variants [8] perform better in the scaled context.

Automated exploration methods could be of great use for quickly constructing new switch-patterns appropriate for present and future challenges. While such methods have also seen successful use in the past [9], [10], they mostly applied a generate-and-test approach, where architectures were first proposed and then evaluated using a separate place-and-route

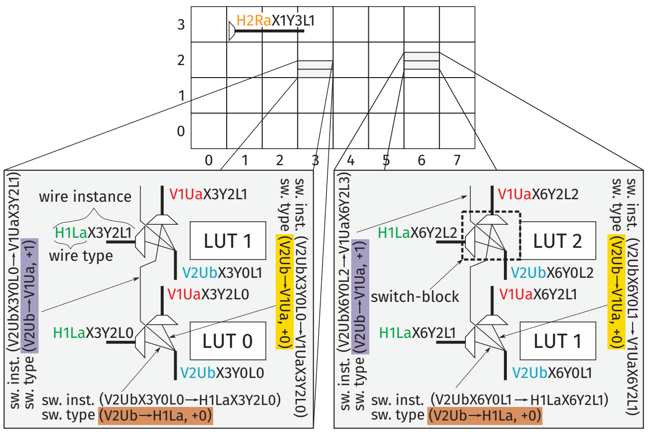


Fig. 1: Illustration of definitions. The architecture has no further meaning.

flow. This typically meant that the search space was fairly constrained, either by artificially imposed (though perhaps reasonable) constraints or by the varying effectiveness of the search method proposing the architectures.

In this paper, we attempt a different approach. We let the router itself freely explore the search space, without any externally-imposed constraints. However, we change the cost of switches that the router sees while routing circuits; along the same negotiation principles used to spread the routes among sufficiently many wires to remove congestion [11], but in the opposite direction, so that the routes can concentrate on a minimal set of switches that will enter the pattern.

After formalizing the problem in Section II, we look at a simple algorithm in Section III and explain why unconstrained exploration by the router without modifying the switch cost may not be effective. We then gradually introduce the idea behind the proposed cost updates in Section IV, first only intuitively, then also formally, reviewing the main concepts of congestion negotiation along the way. Some practical details about the complete search algorithm follow in Section V. We then proceed with an experimental evaluation of the effectiveness of the proposed method in Section VII, which is followed by a comparison with simulated-annealing-based optimization, inspired by prior work by Lin et al. [10], in Section VIII. Conclusions are drawn in Section X.

II. Problem Definition

We assume that the routing channel composition is given and fixed and that the task is to find a set of switches that will provide appropriate connectivity between wires in the routing channels. Without loss of generality, we focus on a routing architecture resembling that of Agilex, where wires in the routing channel are bundled together according to their type (defined shortly) and the same set of bundles starts next to each *Look-Up Table* (LUT) of the tile [6]. Let us first introduce some notation that will be used throughout the paper:

$N \in \mathbb{N}$	Cluster size.
$OLDI$	A wire *type* with orientation $O \in \{H, V\}$, standing for *horizontal* and *vertical*, respectively; length $L \in \mathbb{N}$; direction $D \in \{L, R, U, D\}$, standing for *left*, *right*, *up*, and *down*, respectively; and index $I \in [a..z]$. In Fig. 1, H2Ra designates a horizontal wire going two tiles to the right.
$W_T XxYyLl$	A wire *instance* of type W_T, starting at LUT $l \in [0, N)$, in tile (x, y). In Fig. 1, H2RaX1Y3L1 is a wire of type H2Ra, starting at LUT 1 of tile (1, 3).
$(W_I^i \rightarrow W_I^o)$	A switch *instance*, providing a programmable connection between wire instances W_I^i and W_I^o. In Fig. 1, (V2UbX3Y0L0 $\rightarrow$ V1UaX3Y2L1) provides a connection from the end of the V2Ub wire starting at LUT 0 of tile (3, 0) and the V1Ua wire starting at LUT 1 of tile (3, 2).
$(W_T^i \rightarrow W_T^o, d(l^i, l^o))$	A switch *type* providing a connection between wires of type W_T^i and W_T^o, with the distance between their LUTs equal to $d(l^i, l^o)$. In Fig. 1, (V2Ub $\rightarrow$ V1Ua, +1) is the switch type of the previous switch instance example.
$SB(x, y, l)$	*Switch-block*. The set of all switch *instances* driven by wire instances ending at LUT l of tile (x, y). The switch-block for $(x, y, l) = (6, 2, 2)$ is indicated in Fig. 1.
$SP(x, y, l)$	*Local switch-pattern*. $SP(x, y, l) = \{(W_T^i \rightarrow W_T^o, l^o - l^i) : (W_T XxYyLl^i \rightarrow W_T XxYyLl^o) \in SB(x, y, l)\}$.
V	A set of available wire types.
$E = V \times V \times (-N, N)$	A set of all switch *types* that could exist in any hypothetical local switch-pattern.

Definition 1. *(Switch-Pattern). $E_a \subseteq E$, such that for each (x, y, l) in the FPGA, $SP(x, y, l) = E_a$.*

Now we can define the problem itself:

Task 1. *(Switch-Pattern Exploration). Given a set of switch types E and a set of circuits of interest C, find the switch pattern E_a, such that all circuits in C can be routed and their critical path delays minimized.*

The following definitions will be useful later:

Definition 2. *(Usage, denoted as $U(e)$). The number of switch-blocks in the FPGA in which the switch* type *e is used to route at least one connection of the given circuit.*

Definition 3. *(Occupancy [12], denoted as $O(v)$). The number of circuit's different nets using the wire* instance *v.* Overuse (congestion) *is $O(v) - 1$, since v can legally route one net.*

Intuitively, the relation between a *type* and an *instance* can be understood as that between a *free* and a *bound* vector. Similarly, a *switch-block* is merely an instance of a *switch-pattern*. Unless explicitly specified, the term *switch* will denote a *switch type*, whereas *wire* will denote a *wire instance*.

III. Failure of a Simple Greedy Strategy

In this section, we look at a simple greedy solution to the problem. Its shortcomings will serve to motivate our solution.

A. The Algorithm

The simple greedy Algorithm 1 allows the router to freely use any switch that could exist in the pattern, without any

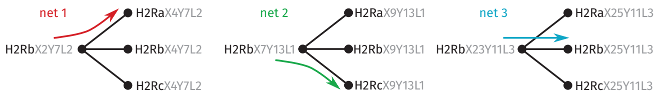


Fig. 2: An example of usage spreading over multiple switches.

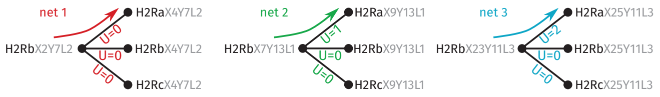


Fig. 3: If the perceived cost of a switch instance is inversely related to its type's usage, nets are motivated to concentrate on the same switch types.

artificial constraints. This is because all instances of all $E = V \times V \times (-N, N)$ switch types are always present in the routing-resource graph (line 1), modeling full connectivity among wires. After each iteration, the algorithm accepts all switch types with usage above $1/\theta$ of the maximum into the switch-pattern that will finally be fabricated, where the *adoption threshold* θ is a parameter. The search stops when no more switch types are added to the pattern. Differentiating the switch types already in the pattern is done through the small ε costs. Without them, the router would repeat the same choices, eventually accepting all switch types with usage > 0 in the first iteration. Despite the seeming simplicity of the algorithm, previous research successfully relied on usage to design novel interconnect architectures [13].

Algorithm 1 Simple Greedy

```
    Input: $\theta \in \mathbb{R}^+$—switch adoption threshold
    Output: switch-pattern
 1: Add all $e \in E$ to the routing-resource graph at cost $\varepsilon \in \mathbb{R}^+$
 2: $E_a = \{\}$, $E_p = E$
 3: do
 4:     Route the relevant circuits
 5:     $U_{max} = max(\{U(e) : e \in E_p\})$
 6:     $E_a = E_a \cup \{e \in E_p : U(e) \geq U_{max}/\theta\}$
 7:     Set cost of all $e \in E_a$ to 0
 8:     $E_p = E \setminus E_a$
 9: while $\exists e \in E_p : U(e) > 0$
10: return $E_a$
```

B. Shortcomings

Yet, let us look at the situation in Fig. 2, depicting three different nets being routed through three different switch-blocks. As all three nets can arbitrarily choose the switch instances they take, for they all seem equally good, it is possible that usage is spread equally among the three switch types. On arriving at line 6, the algorithm has to accept all of them. In other words, there is no way to know if all three switch types are essential for routing the circuit, or the router used all of them equally often simply because it had no incentive to do otherwise.

IV. Turning PathFinder Upside-Down

In this section, we present the main idea of the paper: using the principles of *congestion negotiation* [11] to make the nets reach a consensus on which switch types are really important for routing a given circuit.

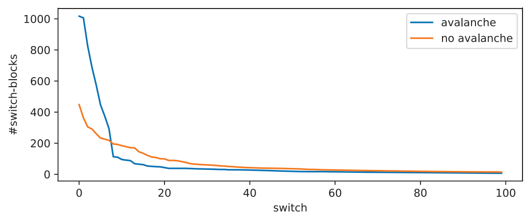


Fig. 4: Concentration effect achieved by the avalanche costs. Usage of the 100 most used switch types, out of the 564 available in one particular experiment, is shown for the case when avalanche costs are enabled and disabled, respectively. The area under the two curves is not identical, as concentration also changes the total number of wires used for routing.

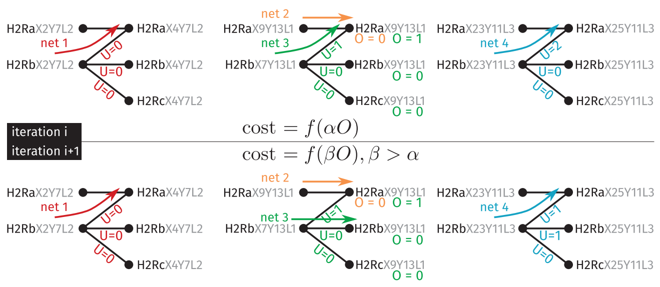


Fig. 5: Too much concentration can lead to congestion on wires. As the cost of wire occupancy, O (Section II) rises over time (indicated by the coefficient β in iteration $i+1 > \alpha$ in iteration i; see Section IV-C), at some point, a net will choose a less used switch type. Eventually, the two effects balance out, producing a legal routing with a minimized number of switch types.

A. *Avalanche Costs*

Let us now see what happens if the cost of switch instances perceived by the nets is not constant but inversely related to their type's usage. This is illustrated in Fig. 3. The first of the three nets in the example sees the same cost at all three switch instances, so it can freely make its choice. The second net, however, sees the switch instance of the type already used by the first net as cheaper, due to the inverse relationship between cost and usage. Hence, it is inclined to choose that same switch type. By the time the router starts processing the third net, the relative cost of (H2Rb $\rightarrow$ H2Ra, $+0$) becomes still smaller, so it is even more inclined to use it. Because the router will reroute nets in subsequent iterations, nets that chose other switches while the cost differences were not as pronounced will have a chance to rectify their choices. This will create an avalanche effect, where the positive feedback keeps reducing the cost of switches with large usage, increasing their usage even more. Thus, the evolving costs enable the nets to reach a consensus on which switch types are important for implementing the given circuit. Fig. 4 shows a concrete example of how the *avalanche costs* concentrate bulk of the usage in a limited subset of the available switch types, suppressing the long tail of others with moderate and low usage.

B. *Negotiating Both Congestion and Switch Presence*

Avalanche costs of Section IV-A are analogous to the very successful *congestion negotiation* heuristic of PathFinder [11]; there, the cost of overused wire *instances* gradually increases, pushing the nets towards a consensus on which ones will give up their desired wires, to spread overuse to other wires and eventually eliminate it. Inversely relating the cost of switch *types* to usage makes the principle act in the opposite direction, causing a consensus on concentration, instead of spreading.

Before discussing in detail the similarities and differences between the two negotiation mechanisms, let us see through the intuitive example of Fig. 5 how they naturally simultaneously act. At routing iteration i, *net 3* may choose to take (H2Rb $\rightarrow$ H2Ra, $+0$), as it is cheaper. However, after the cost of H2RaX9Y13L1 is increased in the next iteration due to congestion (Section IV-C), *net 3* will move to either of the two remaining switches, as the path through them will become cheaper. Because avalanche costs are bounded from both above and below (Section IV-D), while congestion costs are bounded only from below, resolution of congestion is guaranteed.

C. *A Brief Review of Negotiated-Congestion Routing*

In this Section, we give a brief, simplified review of congestion negotiation, focusing on aspects most relevant to this work. The reader should refer to the work of Betz et al. [12] and Murray et al. [14] for an in-depth discussion.

A negotiated-congestion router, such as the one implemented in VPR [12], operates on the so called *routing-resource graph* (rr-graph). In an rr-graph, each wire is represented by a node, while each switch instance is represented by an edge. Each node u has a timing cost $t(u)$ and a congestion cost $cong(u)$, representing the delay and the overuse of the respective wire.

At each routing iteration, each connection (i, j) of the circuit is routed by a shortest path between its endpoints in the rr-graph (fixed during placement). Typically, the timing and the congestion cost of a node u are combined as follows:

$$crit(i,j) \times t(u) + (1 - crit(i,j)) \times cong(u). \qquad (1)$$

Here $crit(i, j)$ is the timing criticality of the connection in the circuit. The first term attempts to route more critical connections through faster wires, whereas the second serves to eliminate congestion. For less critical connections, this term dominates and they release the resources to the more critical ones.

The crucial ingredient in the algorithm that leads to congestion removal is updating the congestion cost, which is a product of three terms: (1) a fixed base cost $b(u)$ of the node u; (2) a term $p(u)$ proportional to the current occupancy of u; and (3) a term $h(u)$ proportional to its cumulative historical overuse. The current congestion term $p(u)$ is updated after each net is routed, to reflect the wire's current occupancy. At the end of each routing iteration, when all the nets have been routed, the historical congestion term of each node, $h(u)$, is increased by its current overuse. This historical term serves to avoid oscillation. Before the new iteration starts, nets are ripped-up so that their new routes can reflect the updated costs. The proportionality constant determining $p(u)$ is typically also increased, to gradually shift the weight from other optimization goals to that of achieving a legal, congestion-free routing.

D. *Functional Form of the Avalanche Costs*

As mentioned in Section IV-C, switch instances are traditionally represented as edges in the rr-graph. However, cost is typically attributed to nodes. Hence, for each switch instance, we split the corresponding edge by a virtual node which allows seamless cost attribution and tracking of switch usage.

For the avalanche costs, we use a functional form that is similar to that of the congestion cost of Section IV-C:

$$a(u) = max(0, s(u) - a_p \times U(u) - a_h \times U_h(u)). \quad (2)$$

Here, $s(u)$ is the starting cost assigned to the given switch, which is also its maximum cost. Parameter a_p determines how quickly the avalanche cost drops as a function of the current usage of the switch, $U(u)$, while a_h determines how quickly it drops as a function of the cumulative historical usage $U_h(u)$.

Like the occupancy trackers of Section IV-C, $U(u)$ is updated each time a net is routed, while $U_h(u)$ is updated only once the current routing iteration completes. We note once more, however, that unlike the occupancy trackers, which are bound to individual nodes of the rr-graph (individual wire *instances*), the usage trackers $U(u)$ and $U_h(u)$ are shared between all nodes representing instances of switches of the same *type*. This allows for communicating switch type choices to nets using entirely different switch-blocks (Fig. 3) and eventually reaching a consensus on which switch types will enter the pattern.

E. Respecting the Critical Paths

A good switch-pattern must enable the router to properly optimize the critical path of each circuit of interest. Hence, during the pattern search, critical connections must be able to route even through switches with otherwise low usage.

Aside from the avalanche cost, we assign to each switch a timing cost equal to the projected delay increase of the wire that is driving it, due to the increased load at its end. Combining the two costs could be achieved as in Equation 1. However, as the maximum value of the avalanche costs, s, can be very large, discouraging even the most critical nets from using switches with low usage, we opted for another form:

$$c(u) = t(u) + e^{\left(\frac{ln(s_c/s)}{max\_crit^{\beta}} \times crit(i,j)^{\beta}\right)} \times a(u). \quad (3)$$

Here s_c is a parameter determining the perceived cost of a potential switch when routing the most critical possible net, with criticality max_crit (a standard parameter of VPR [14]), and β is a criticality exponent used to tune the selectivity of the function. As Fig. 6 shows, this provides better control of the trade-off between critical path optimization and minimization of the number of used switch types than Equation 1.

V. Completing the Algorithm

The complete algorithm is almost identical to Algorithm 1, apart from the fact that routing on line 4 is performed using a modified version of VTR 8 [14], which incorporates the avalanche costs of Section IV. Another difference is that if there are switches which got their avalanche cost reduced to zero in the current iteration, all of them are selected and the usage-threshold-based selection of line 6 is skipped. Nodes representing instances of the selected switch types are removed from the rr-graph and their neighbors are connected directly. This is equivalent to resetting their costs to 0 (line 7), but has a practical benefit of reducing the size of the rr-graph.

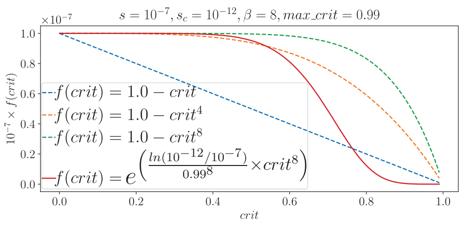
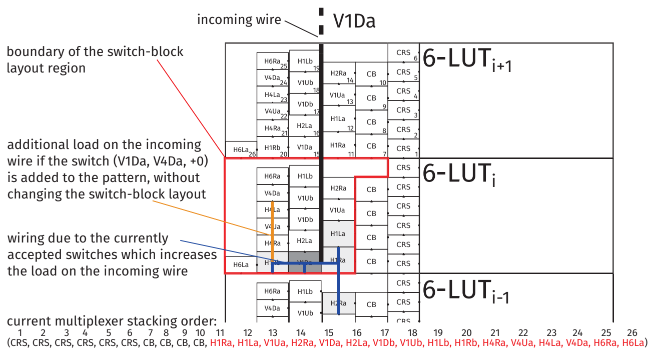


Fig. 6: Comparison of functions from Equation 1 and Equation 3. The proposed function of Equation 3 allows for precise tuning of the avalanche cost that the most critical nets perceive, so that the timing requirements are sufficient to motivate them to use switches with otherwise low usage. It also creates a relatively flat region of low avalanche cost for a wider range of high criticalities, necessary for actually optimizing the critical path delay, given that the timing analysis during routing is done only infrequently. A relatively steep rise in cost ensues once the criticality drops bellow the cut-off point, which is needed to discourage noncritical nets from increasing the switch-pattern size. The function of Equation 1 and its exponentiated versions [14] lack these features.



incoming wire
V1Da
boundary of the switch-block layout region
additional load on the incoming wire if the switch (V1Da, V4Da, +0) is added to the pattern, without changing the switch-block layout
wiring due to the currently accepted switches which increases the load on the incoming wire
6-LUT_{i+1}
6-LUT_{i}
6-LUT_{i-1}
current multiplexer stacking order:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
(CRS, CRS, CRS, CRS, CRS, CRS, CB, CB, CB, CB, H1Ra, H1La, V1Ua, H2Ra, V1Da, H2La, V1Db, V1Ub, H1Lb, H1Rb, H4Ra, V4Ua, H4La, V4Da, H6Ra, H6La)

Fig. 7: Floorplan construction. Not drawn to scale.

A. Conveying Physical Information

Apart from the delays of the routing wires, necessary for proper timing optimization, the router must be aware of the implications of using a certain switch, not yet in the final pattern, on the architecture's performance. Before each iteration of the algorithm, we run a physical modeling and optimization flow to provide this data for the modified switch-pattern.

1) Modeling Flow: To extract the delays of the routing wires, we rely on the modeling flow developed in our previous work [15]. The flow assumes a floorplan similar to that of the Stratix FPGAs [16], where LUTs are stacked on top of each other, while the routing multiplexers are arranged in columns padded to their left. Fig. 7 depicts one such floorplan. Each multiplexer column is filled from the bottom up, until its height matches the height of the adjacent LUT.

2) Multiplexer Position Optimization: Precise positions of all multiplexers allow for accurate modeling of intra-SB wiring (depicted in blue in Fig. 7, for one source routing wire), which in turn allows for correctly taking into account the influence of this wiring on the delay of the routing wires. However, the possibility of changing the multiplexer positions as the switch-pattern evolves also needs to be considered.

In our previous work, multiplexers were stacked in a fixed order, derived from their input count [15]. Now we adapt the multiplexer positions to the changing connectivity by performing a quick anneal of the stacking order. All moves represent swaps of two randomly selected multiplexers in the order, upon which a new floorplan is generated. For the cost function, we use a combination of the total intra-SB wirelength and a timing cost computed as a product of approximate routing wire delay and its exponentiated criticality extracted from the last routing run, summed over all routing wires. This cost function was adopted from VPR's timing-driven placer [17]. During multiplexer position optimization and routing wire delay measurement, only those switches which have already been adopted to the final switch-pattern are considered.

Routing wire delays reported to the router for the next iteration of the pattern search are obtained directly from SPICE simulations [15]. However, the annealing process uses approximate delays obtained by querying a model precomputed to relate a routing wire's delay increase due to intra-SB wiring that it drives to this wiring's total length.

3) Impact of Potential Switches: Output of the same model—merely a polynomial fitted to a set of SPICE simulations, with linear interpolation to zero, to allow for differentiating between switches that imply too little extra wiring (orange in Fig. 7) for their impact to be measurable—is assigned to the timing cost (Equation 3) of each switch not yet in the final pattern.

The impact that using each potential switch has on performance generally depends on which other potential switches are also used. However, if the adoption threshold θ (Section III) is sufficiently small to prevent adoption of too many switches between reevaluations of the physical model of the switch-block, the simple approach of only informing the router about the impact that each switch has in isolation should suffice.

B. Preventing Overspecialization

To prevent the resulting pattern from being specialized to a particular placement, we replace the circuits using a different placement seed after each iteration of the search algorithm. Ensuring that the pattern can support all circuits of interest can be achieved by expanding the circuit set used during the search. To minimize the dependence of the results on the order in which the circuits are processed, we route multiple circuits simultaneously, each on an FPGA of its own. This way, the natural structure and the timing requirements of each circuit are preserved, while the avalanche costs are shared across them, allowing their nets to jointly negotiate switch presence.

C. Parameters

The functional form of the avalanche costs (Equation 2) involves three parameters: the starting avalanche cost, $s(u)$ and the two parameters dictating the rate of cost decrease with respect to usage, a_p and a_h. For the search method to be effective, these parameters must be assigned reasonable values. Because different switches are already distinguished by their timing cost, we chose to fix all $s(u)$ to a single parameter s.

1) Adaptive Tuning: The rate at which avalanche cost should drop with respect to usage depends fundamentally on the actual usage values attained during routing: a single fixed drop rate could be too high if many nets naturally tend to use the same switch types, whereas it could be too low if the number of nets which do so is very small. This depends on the size and structure of the circuits being routed in the search process, making it difficult to choose a single value for a_p and a_h.

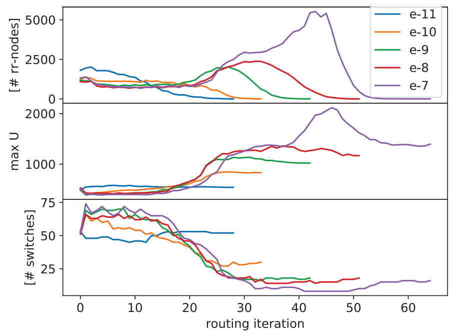


Fig. 8: Dependence of concentration on the starting avalanche cost. The top graph shows the number of congested nodes in the rr-graph after each iteration of the router in the first iteration of the search algorithm, for *iter_to_zero* set to 25 and various starting avalanche costs. The middle graph shows the corresponding maximum usage, while the bottom one shows the number of switches with usage $\geq 0.05\times$ the current maximum.

To resolve this issue, we first record the maximum usage during the first routing iteration, when the avalanche costs are temporarily reset to zero, to allow all nets to initially choose the timing-optimal resources (much like VPR typically neglects congestion in the first iteration [14]). Let this maximum usage be M_U^1. Then we compute a_p and a_h as follows:

$$a_p = a_h = \frac{s}{M_U^1 \times (iter\_to\_zero + 1)} \quad (4)$$

Hence, a_p and a_h are set to the value required for the avalanche cost to be reduced to zero in $iter\_to\_zero \in \mathbb{N}$ routing iterations, assuming a sustained usage of M_U^1. Thus we fix both a_p and a_h using a single metaparameter with a much more graspable meaning. Once computed in the first iteration of the algorithm, a_p and a_h do not change until the end of the search. Consequences of equating them are still to be understood.

2) Starting Cost: Fig. 8 shows the effect of various starting costs on concentration and congestion resolving when simultaneously routing the *alu4*, *ex5p*, and *tseng* MCNC circuits [18], with $iter\_to\_zero = 25$. In the first graph, we see that all explored values of s cause a rise in the number of congested nodes which disappears once congestion is penalized sufficiently for nets to move to switches with lower usage and higher avalanche cost. Larger values of s lead to higher peaks of congestion occurring later in the routing process.

The middle graph clearly shows the correlation between rising concentration and congestion. Larger values of s initially make it less likely for nets to route through switches with low usage, leading to larger peaks of maximum usage. However, excessive concentration is not sustainable, because it prevents congestion resolution. The overshoot for $s = 10^{-7}$ depicts this clearly and although its final maximum usage is also somewhat higher than for the other values of s, some routing iterations are inevitably wasted. Apart from the maximum usage, the number of switches with significant usage (here set at $\geq 5\%$ of the current maximum) is also illustrative. As the bottom graph shows, all explored values of s—apart from 10^{-11} and 10^{-10} which are clearly too low to prevent nets from using switches not required by other nets—lead to very similar results in this

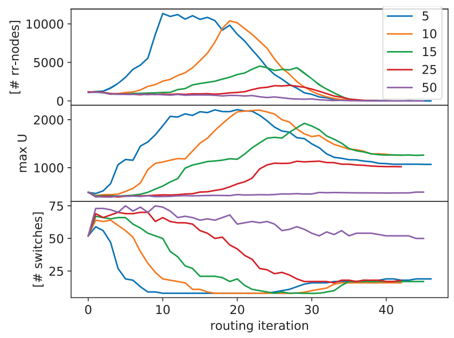


Fig. 9: Dependence of concentration on the rate of avalanche cost decrease. Graphs are analogous to those of Fig. 8, for the starting avalanche cost fixed at 10^{-9} and *iter_to_zero* $\in \{5, 10, 15, 25, 50\}$.

respect, by the end of the routing process.

While larger values of s, such as 10^{-7} can be used to attempt additional reduction of the obtained switch-pattern size, in the experiments which follow, we use $s = 10^{-9}$ since it provides a reasonable trade-off between concentration and runtime.

3) Rate of Decrease: Fig. 9 shows the results of sweeping *iter_to_zero* under the setup of Section V-C2, but with s fixed at 10^{-9}. Smaller values quickly reduce the cost of switches which are intrinsically in high demand (usage close to M_U^1), causing an early concentration and congestion increase. Upon congestion resolution, however, different explored values converge to very similar results. The exception is 50, which results in too slow drop in avalanche costs that does not allow the higher-usage switches to attract nets to route through them.

It appears that a good trade-off between concentration and runtime is given by values corresponding to about half the total number of routing iterations taken to achieve a congestion-free routing. In subsequent experiments, we use $iter\_to\_zero = 25$.

More comprehensive analyses could lead to parameter values resulting in solutions of better quality and/or runtime reduction. Whether the conclusions drawn here would change for larger and more complex circuits also remains an open question. Nevertheless, at the moment it does not seem that the proposed method is particularly sensitive to exact values of parameters.

VI. Experimental Setup

All experiments are performed on an architecture with eight 6-LUTs in the cluster and a channel composition reminiscent of that of Agilex, but for the longest wires [6]: $2 \times$ H1, H2, H4, H6, $2 \times$ V1, V4. These wires are repeated for each LUT of the cluster. Without loss of generality, we consider only switches with LUT distance $\in \{-1, 0, 1\}$ (Section II) and prohibit switches to a target wire going in the direction from which the driving one came. This results in 564 available switches. The connection-blocks and crossbars generated by the physical modeling flow are kept constant in all experiments, while delays are extracted from a 4-nm technology model [15].

VII. Effectiveness of Avalanche Costs

In this section, we assess the effectiveness of the proposed avalanche search method against the simple greedy algorithm of Section III. Instead of introducing explicit ε costs without a physical meaning to the greedy algorithm, we use the timing costs of the switches equally visible to all nets, regardless of criticality (Equation 3). Search was performed by simultaneously routing the *alu4*, *ex5p*, and *tseng* circuits. The switch adoption threshold θ was set to 1.1 for both algorithms. Final assessment of delay performance was done on all MCNC circuits, except for the pin-bound *dsip*, *des*, and *bigkey*.

TABLE I: Properties of the avalanche and greedy patterns.

	avalanche			greedy			truncated greedy		
#iterations	63			228			62		
#switches	93			438			92		
	fi_{avg}	fo_{avg}	t_{avg} [ps]	fi_{avg}	fo_{avg}	t_{avg} [ps]	fi_{avg}	fo_{avg}	t_{avg} [ps]
H1	5	5	14.5	31	25	23.1	6	4	14.3
H2	5	5	17.8	28	28	31.6	7	6	18.4
H4	8	7	25.9	21	27	43.2	4	7	26.4
H6	6	6	34.9	19	25	59.6	2	7	35.9
V1	7	7	22.2	38	31	35.5	10	7	22.0
V4	5	8	71.7	12	27	97.5	2	5	67.8
W(tile)	6816 nm			8904 nm			7368 nm		
CPD	1.40 ns			1.71 ns			1.41 ns		

A. Direct Comparison with Greedy

Avalanche search converged after 62 iterations, accumulating 93 switches, while greedy search converged only after 228 iterations, accepting 438 switches (Table I). This demonstrates that projected delay contributions of individual switches alone are insufficient to deter the router from using them. The large number of switches in the greedy pattern resulted in both a large increase of the tile width and the average fanin and fanout of intercluster wires. This in turn led to a large increase of average wire delays and the routed critical path delay (Table I).

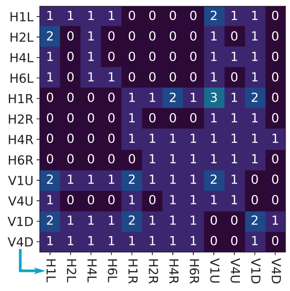
B. Comparison with Truncated Greedy

To better assess the differences in the choices made by the two search methods, we truncated the greedy pattern after the 62nd iteration, when the pattern contained 92 switches, which was the closest to the 93 of the avalanche one. The exact distribution of fanouts and fanins enables a tighter packing of the multiplexers of the avalanche pattern, leading to a lower tile width. Fanouts and fanins still determine the wire delays, however, which are very close between the two patterns, and on average slightly lower for the truncated greedy (Table I).

1) Adjacency: Adjacency between different wire types (here considered without the index; see Section II) is illustrated in Fig. 10. The more numerous zero and larger entries in the adjacency matrix of the greedy pattern show that greedy search selects multiple switches between the same types of wires, commonly connected by the router, where only a subset of them would suffice. As a result, with the same number of switches, fewer wire types can be connected.

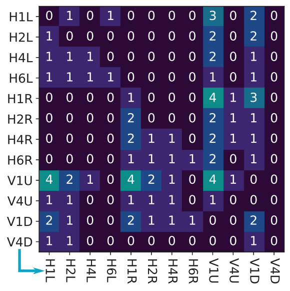
2) Grid Distances: Consequences of selecting multiple switches between the same wire types, instead of introducing more variety, can be seen in Fig. 11. Each entry of the matrices represents the minimum number of distinct intercluster wires needed to connect the center of the grid to the particular target, normalized by the minimum number of wires that would be needed if all switches were available in the pattern. The avalanche pattern is closer to being optimal in this respect.

This is also reflected on the minimum delay distances, relative to an unrealistic fully-connected pattern which disregards



	H1L	H2L	H4L	H6L	H1R	H2R	H4R	H6R	V1U	V4U	V1D	V4D
H1L	1	1	1	1	0	0	0	0	2	1	1	0
H2L	2	0	1	0	0	0	0	0	1	0	1	0
H4L	1	0	1	0	0	0	0	0	1	1	1	0
H6L	1	0	1	1	0	0	0	0	1	0	1	0
H1R	0	0	0	0	1	1	2	1	3	1	2	0
H2R	0	0	0	0	1	0	0	0	1	1	1	0
H4R	0	0	0	0	1	1	1	1	1	1	1	1
H6R	0	0	0	0	0	1	1	1	1	1	1	0
V1U	2	1	1	1	2	1	1	1	2	1	0	0
V4U	1	0	0	0	1	0	1	1	1	1	0	0
V1D	2	1	1	1	2	1	1	1	0	0	2	1
V4D	1	1	1	1	1	1	1	1	0	0	1	0

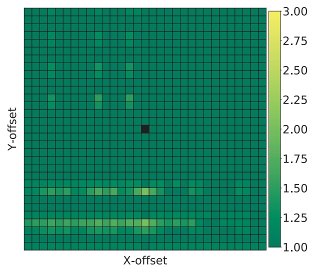
(a) Avalanche



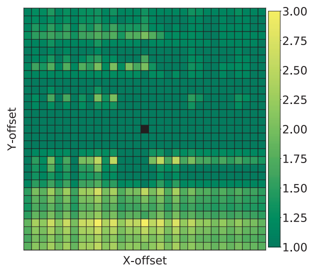
	H1L	H2L	H4L	H6L	H1R	H2R	H4R	H6R	V1U	V4U	V1D	V4D
H1L	0	1	0	1	0	0	0	0	3	0	2	0
H2L	1	0	0	0	0	0	0	0	2	0	2	0
H4L	1	1	1	0	0	0	0	0	2	0	1	0
H6L	1	1	1	1	0	0	0	0	1	0	1	0
H1R	0	0	0	0	1	0	0	0	4	1	3	0
H2R	0	0	0	0	2	0	0	0	2	1	1	0
H4R	0	0	0	0	2	1	1	0	2	1	1	0
H6R	0	0	0	0	1	1	1	1	2	0	1	0
V1U	4	2	1	0	4	2	1	0	4	1	0	0
V4U	1	1	0	0	1	1	1	0	1	0	0	0
V1D	2	1	0	0	2	1	1	1	0	0	2	0
V4D	1	1	0	0	0	0	0	0	0	0	1	0

(b) Truncated Greedy

Fig. 10: Adjacency of wire types: avalanche (a) and truncated greedy (b). Note that all H1 and V1 wires occur twice in each direction (see Section VI).



(a) Avalanche



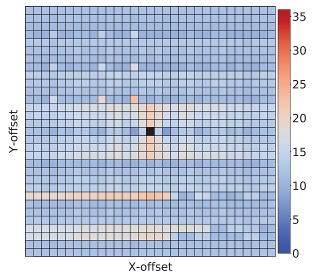
(b) Truncated Greedy

Fig. 11: Hop-distances from the center of the FPGA to other tiles, normalized by the distances computed on a pattern with all switches. Dark green is best.

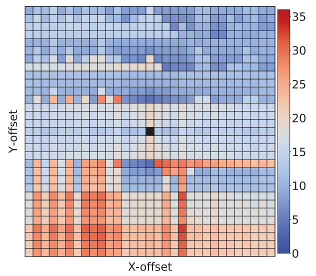
the impact of switch load on wire delay (Fig. 12). The relative inefficiency in connecting to the distant targets at the bottom of the grid was influenced by performing the search on small circuits requiring very short FPGAs. In a production setting, of course, larger and more complex circuits should be used.

3) Routed Delays: Despite the qualitative differences between the avalanche and the truncated greedy pattern, they are largely equivalent in terms of the routed critical path delays (Fig. 13). This could be due to the MCNC circuits imposing low stress on the routing architecture, making it easy to meet timing requirements. Another reason could lie in their large logic depth, unrepresentative of the modern pipelined circuits, which, combined with the lack of any interconnect pipelining in the architecture [19], [20] and oversimplified intracluster interconnect [6], makes the delays inside the cluster dominant.

While accounting for interconnect pipelining goes beyond the scope of the present paper, we believe that there is nothing that would prevent the proposed avalanche search method from also being applied to intracluster interconnect. However, to



(a) Avalanche



(b) Truncated Greedy

Fig. 12: Percentage increase of the delay needed to reach other tiles from the center, compared to a hypothetical switch-pattern containing all switches with no impact on wire delay. Dark blue is best.

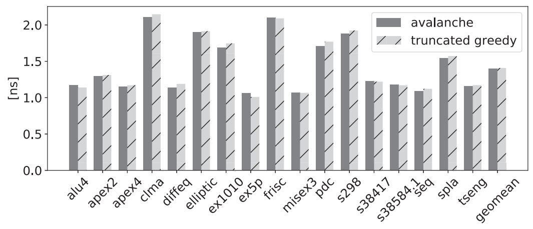


Fig. 13: Routed delays for the avalanche and the truncated greedy pattern.

TABLE II: Number of routing iterations taken by VPR to successfully route the Gnl circuits. Failure to route in 300 iterations is marked with “—”.

circuit	1	2	3	4	5	6	7	8	9	10
avalanche	147	145	57	73	56	71	82	59	65	74
trunc. greedy	—	—	—	—	278	—	—	—	149	—

the best of our knowledge, VPR currently does not support simultaneous intercluster and intracluster routing, so we decided to leave this for future work as well.

4) Routability: To see how the two patterns compare under increased stress, we generate ten synthetic circuits with about 10 000 LUTs using *Gnl* [21]. The Rent's exponent was set to 0.7—the maximum used in the ISPD'16 routability driven placement contest [22]. We take the distribution of different LUT sizes in the circuits from Hutton et al. [23]. Then, we place the circuits on architectures based on the two switch-patterns and attempt to route them with a limit of 300 iterations. We neglect timing optimization since the circuits are synthetic.

Table II shows the number of iterations needed for VPR to successfully route each circuit. While the avalanche pattern successfully routes all ten circuits, the truncated greedy succeeds only on two, and that with considerably more effort. This demonstrates the effectiveness of avalanche search in maximizing routability of the constructed switch-patterns.

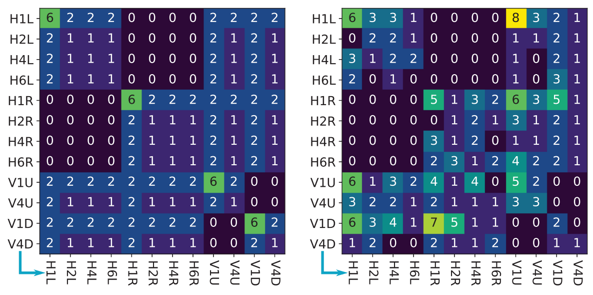
VIII. Comparison with Simulated Annealing

Lin et al. successfully used simulated annealing for simultaneously optimizing channel composition and the switch-pattern [10]. In this section, we investigate how a similar method compares with the proposed avalanche search.

A. Initial Pattern

We initialize the search with the default pattern produced by the physical modeling flow [15], which represents our best effort at capturing inter-wire-type connectivity of a modern tapless architecture [24], with the constraint dictated by the high resistance of the lower metal layers that bulk of this connectivity is contained within wires starting and ending at the same LUT-height [6]. Implementing e.g., a Wilton pattern is not possible under this constraint, since the very few instances of each wire type per LUT (1–2 [6]) do not allow for implementing the necessary track permutations [9]. Such a comparison could be done in an older technology, where resistance is not an issue, but there precise switch choices enabled by the proposed search method would be less relevant, we believe, as they would have little impact on performance, while good routability in presence of taps was demonstrated even on nonobvious patterns [25].

The initial pattern contains 180 switches organized as shown in Fig. 14a. The optimal hop-distances that it achieves are not



(a) Initial (b) Annealed

Fig. 14: Adjacency of wire types: initial (a) and annealed (b).

TABLE III: Properties of the initial and the annealed pattern.

	avalanche			initial			annealed		
#switches	93			180			210		
	fi_{avg}	fo_{avg}	t_{avg} [ps]	fi_{avg}	fo_{avg}	t_{avg} [ps]	fi_{avg}	fo_{avg}	t_{avg} [ps]
H1	5	5	14.5	10	10	16.0	13	13	19.6
H2	5	5	17.8	11	11	21.3	14	11	24.1
H4	8	7	25.9	11	11	30.8	16	12	32.1
H6	6	6	34.9	11	11	43.1	9	13	47.3
V1	7	7	22.2	12	12	24.6	14	15	29.2
V4	5	8	71.7	13	13	74.3	13	15	86.8
W(tile)	6816 nm			7464 nm			7488 nm		
CPD	1.40 ns			1.46 ns			1.55 ns		

sufficient to counter the wire delay increase due to a high load (Table III). As a result, the geomean routed delay is about 4% larger than for the avalanche pattern (Table III).

B. Setup

We use two very simple moves generated with equal probability: including or removing one of the 564 considered switches. The self-normalizing two-term cost function of Marquardt et al. [17] is used, with tile area and the geomean routed critical path delay of the circuits used in the search taken for the two terms, with equal contribution. To save runtime, wire delays are measured only when the switch-pattern differs from that of the previously measured architecture in at least five switches, while floorplan is optimized only on temperature change. The same three MCNC circuits driving the avalanche search of Section VII are used again. The initial temperature is set to 0.02 and we perform 100 temperature changes, at the rate of 0.95, with 100 moves per temperature.

C. Results

Including or removing a single switch from the pattern most often has little influence on the critical path delay, or tile area, which only dramatically changes with a change in the number of columns needed to fit the multiplexers (Fig. 7). This makes convergence of the optimization difficult, as visible in Fig. 15. In the present experiment, 30 new switches were added,

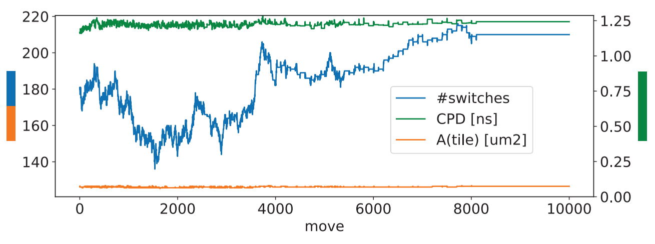


Fig. 15: Convergence of the simulated annealing optimization.

while both adjacency regularity (Fig. 14b), and hop-distance optimality were broken. The increased wire delays (Table III) further increased the geomean routed delay by about 6%.

We conjecture that for Lin et al. annealing the switch-pattern proved valuable as during the optimization of the channel composition—likely causing larger and easier to capture changes in performance—the switch-pattern grew increasingly inappropriate for the new composition and annealing it was just sufficient to rectify that. If applied to one fixed channel composition, success of the method seems less obvious.

Of course, we do not claim that simulated annealing, or any other general optimization method, cannot be made to work for switch-pattern exploration, if extensive engineering of the cost function and move generation is performed. Nevertheless, much like the original PathFinder removed the need for elaborate ad hoc heuristics of early FPGA routers [26], we believe that our avalanche-cost method, essentially relying on the same principles as PathFinder, removes the need for similarly elaborate heuristics to explore interconnect architectures.

IX. Runtime Scalability

A fundamental feature of avalanche search is that it presents the router with the entire search space at once, instead of using it for in-the-loop evaluation of explicitly constructed solutions. Hence, longer routing runs can be tolerated than if thousands of explicit solutions must be assessed in sequence. Some features of avalanche search significantly slow down routing, however: 1) more iterations required to eliminate congestion, 2) a need for rerouting even the uncongested nets so that they can choose higher-usage switches, and 3) large avalanche costs making lookaheads ineffective. Some experiments on Gnl circuits showed slowdowns exceeding $100\times$ compared to routing on the final pattern with original VTR 8. Since these runtimes were still on the order of hours, we did not attempt any remedies, although they are certainly possible. For instance, we used a single lookahead map [14], computed with zero avalanche costs to always be admissible, but multiple maps would enable tighter tracking of actual cost evolution. With such enhancements, we are confident that runtime of routing runs (line 4 of Algorithm 1) could also be greatly improved.

X. Conclusions

In this work, we presented an effective method for exploring switch-patterns of an FPGA using the router itself. We hope that this offers a new view on the problem and opens new perspectives for architectural research. For instance, it could reduce the need for assuming that the switch-pattern design is orthogonal to other architectural parameters [27]; one may simply use the method to search for a new pattern appropriate for the given values of the other parameters. Similarly, the method could be effective for automated specialization of switch-patterns in custom FPGAs [28]. The presented experiments focused only on programmable intercluster interconnect, but we believe that the method can also be successfully used for exploring other aspects of FPGA routing. This we plan to attempt in the future.

The source code used in this work is available at the following link: https://github.com/EPFL-LAP/fpl21-avalanche.

REFERENCES

[1] J. Rose and S. Brown, “Flexibility of interconnection structures for field-programmable gate arrays,” *IEEE Journal of Solid-State Circuits*, vol. 26, no. 3, pp. 277–82, 1991.

[2] S. J. Wilton, “Architectures and algorithms for field-programmable gate arrays with embedded memory,” Ph.D. dissertation, University of Toronto, 1997.

[3] Y.-W. Chang, D. F. Wong, and C. K. Wong, “Universal switch modules for FPGA design,” *ACM Transactions on Design Automation of Electronic Systems*, vol. 1, no. 1, pp. 80–101, Jan. 1996.

[4] P. Gopalakrishnan, Xin Li, and L. Pileggi, “Architecture-aware FPGA placement using metric embedding,” in *2006 43rd ACM/IEEE Design Automation Conference*, 2006, pp. 460–65.

[5] H. Schmit and V. Chandra, “FPGA switch block layout and evaluation,” in *Proceedings of the 2002 ACM/SIGDA Tenth International Symposium on Field-Programmable Gate Arrays*, Monterey, CA, USA, Feb. 2002, pp. 11–18.

[6] 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 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Seaside, CA, USA, Feb. 2020, pp. 140–49.

[7] X. Tang, E. Giacomin, A. Alacchi, and P. Gaillardon, “A study on switch block patterns for tileable FPGA routing architectures,” in *2019 International Conference on Field-Programmable Technology (ICFPT)*, Tianjin, China, Dec. 2019, pp. 247–50.

[8] O. Petelin and V. Betz, “The speed of diversity: Exploring complex FPGA routing topologies for the global metal layer,” in *Proceedings of the 26th International Conference on Field Programmable Logic and Applications*, Lausanne, Switzerland, Aug. 2016, pp. 1–10.

[9] G. G. Lemieux and D. M. Lewis, “Analytical framework for switch block design,” in *Field-Programmable Logic and Applications, Reconfigurable Computing Is Going Mainstream, 12th International Conference, FPL 2002, Montpellier, France, September 2-4, 2002, Proceedings*, ser. Lecture Notes in Computer Science, M. Glesner, P. Zipf, and M. Renovell, Eds., vol. 2438. Springer, Sep. 2002, pp. 122–31.

[10] M. Lin, J. Wawrzynek, and A. E. Gamal, “Exploring FPGA routing architecture stochastically,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 10, pp. 1509–22, Sep. 2010.

[11] L. McMurchie and C. Ebeling, “PathFinder: A negotiation-based performance-driven router for FPGAs,” in *Proceedings of the 1995 ACM Third International Symposium on Field-Programmable Gate Arrays*, Monterey, CA, USA, Feb. 1995, pp. 111–17.

[12] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, 1999.

[13] G. Wang, S. Sivaswamy, C. Ababei, K. Bazargan, R. Kastner, and E. Bozorgzadeh, “Statistical analysis and design of HARP FPGAs,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 10, pp. 2088–102, 2006.

[14] K. E. Murray, O. Petelin, S. Zhong, J. M. Wang, M. Eldafrawy, J.-P. Legault, E. Sha, A. G. Graham, J. Wu, M. J. P. Walker, H. Zeng, P. Patros, J. Luu, K. B. Kent, and V. Betz, “VTR 8: High-performance CAD and customizible FPGA architecture modelling,” *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 13, no. 2, pp. 9:1–9:60, May 2020.

[15] S. Nikolić, F. Catthoor, Z. Tőkei, and P. Ienne, “Global is the new local: FPGA architecture at 5nm and beyond,” in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Virtual Event, USA, 2021, pp. 34–44.

[16] D. Lewis, D. Cashman, M. Chan, J. Chromczak, G. Lai, A. Lee, T. Vanderhoek, and H. Yu, “Architectural enhancements in Stratix V™,” in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, CA, USA, Feb. 2013, pp. 147–56.

[17] A. Marquardt, V. Betz, and J. Rose, “Timing-driven placement for FPGAs,” in *Proceedings of the 2000 ACM/SIGDA Eighth International Symposium on Field Programmable Gate Arrays*, Monterey, CA, USA, Feb. 2000, pp. 203–13.

[18] S. Yang, “Logic synthesis and optimization benchmarks user guide, version 3.0,” Microelectronics Center of North Carolina, Technical Report, Jan. 1991.

[19] D. Lewis, G. Chiu, J. Chromczak, D. Galloway, B. Gamsa, V. Manoharara-jah, I. Milton, T. Vanderhoek, and J. Van Dyken, “The Stratix™ 10 highly pipelined FPGA architecture,” in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Monterey, CA, USA, Feb. 2016, pp. 159–68.

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

[21] D. Stroobandt, J. Depreitere, and J. V. Campenhout, “Generating new benchmark designs using a multi-terminal net model,” *Integration*, vol. 27, no. 2, pp. 113–29, 1999.

[22] S. Yang, A. Gayasen, C. Mulpuri, S. Reddy, and R. Aggarwal, “Routability-driven FPGA placement contest,” in *Proceedings of the 2016 on International Symposium on Physical Design*, Santa Rosa, CA, USA, 2016, pp. 139–43.

[23] M. D. Hutton, J. Schleicher, D. M. Lewis, B. Pedersen, R. Yuan, S. Kaptanoglu, G. Baeckler, B. Ratchev, K. Padalia, M. Bourgeault, A. Lee, H. Kim, and R. Saini, “Improving FPGA performance and area using an adaptive logic module,” in *Field Programmable Logic and Application, 14th International Conference, FPL 2004, Leuven, Belgium, August 30-September 1, 2004, Proceedings*, ser. Lecture Notes in Computer Science, J. Becker, M. Platzner, and S. Vernalde, Eds., vol. 3203. Springer, 2004, pp. 135–44.

[24] M. B. Petersen, S. Nikolić, and M. Stojilović, “NetCracker: A peek into the routing architecture of Xilinx 7-Series FPGAs,” in *Proceedings of the 2021 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Feb. 2021, pp. 11–22.

[25] A. Li, T.-J. Chang, and D. Wentzlaff, “Automated design of FPGAs facilitated by cycle-free routing,” in *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, Aug 2020, pp. 208–13.

[26] S. B. G. Lemieux, “A detailed router for allocating wire segments in FPGAs,” in *ACM/SIGDA Physical Design Workshop*, Lake Arrowhead, CA, USA, Apr. 1993, pp. 215–26.

[27] A. Yan, R. Cheng, and S. J. E. Wilton, “On the sensitivity of FPGA architectural conclusions to experimental assumptions, tools, and techniques,” in *Proceedings of the 2002 ACM/SIGDA Tenth International Symposium on Field-Programmable Gate Arrays*, Monterey, CA, USA, Feb. 2002, pp. 147–56.

[28] D. Koch, N. Dao, B. Healy, J. Yu, and A. Attwood, “FABulous: An embedded FPGA framework,” in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Virtual Event, USA, 2021, pp. 45–56.