Exploring FPGA Switch-Blocks without Explicitly Listing Connectivity Patterns

STEFAN NIKOLIĆ and PAOLO IENNE, École Polytechnique Fédérale de Lausanne, Switzerland

Increased lower metal resistance makes physical aspects of Field-Programmable Gate Array (FPGA) switch-blocks more relevant than before. The need to navigate a design space where each individual switch can have significant impact on the FPGA's performance in turn makes automated switch-pattern exploration techniques increasingly appealing. However, most existing exploration techniques have a fundamental limitation—they use the CAD tools as a black box to evaluate the performance of explicitly listed switch-patterns. Given the time needed to route a modern circuit on a single architecture, the number of switch-patterns that can be explicitly tested quickly becomes negligible compared to the size of the design space. This article presents a technique that removes this fundamental limitation by making the entire design space visible to the router and letting it choose the switches to be added to the pattern, based on the requirements of the circuits being routed. The key to preventing the router from selecting arbitrary switches that would render the final pattern excessively large is to apply the same negotiation principle used by the router to remove congestion, just in the opposite direction, to make the signals reach a consensus on which switches are worthy of being included in the final switch-pattern.

CCS Concepts: • **Hardware → Programmable interconnect**; **Methodologies for EDA**; **Wire routing;**

Additional Key Words and Phrases: FPGA, interconnect, switch, switch-block, switch-pattern, multiplexer, automated exploration, optimization, design automation, algorithm, avalanche, PathFinder, router

ACM Reference format:
Stefan Nikolić and Paolo Ienne. 2024. Exploring FPGA Switch-Blocks without Explicitly Listing Connectivity Patterns. *ACM Trans. Reconfig. Technol. Syst.* 17, 1, Article 14 (February 2024), 39 pages.
https://doi.org/10.1145/3597417

1 WHY AUTOMATE SWITCH-PATTERN EXPLORATION?

When FPGA architecture research started to develop, considerable attention was given to the design of the switch-patterns used in programmable interconnect [1–3]. Typically, the goal was to optimize some metric of routability (e.g., minimum channel width) 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 that they are optimal in a certain way [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 given to wiring inside the switch-block itself. Over time, a few switch-patterns emerged as dominant and further research in the area subsided.

Authors' address: S. Nikolić and P. Ienne, École polytechnique fédérale de Lausanne, School of Computer and Communication Sciences, CH-1015 Lausanne, Switzerland; emails: {stefan.nikolic, paolo.ienne}@epfl.ch.

1936-7406/2024/02-ART14 $15.00
https://doi.org/10.1145/3597417

This has not been the case in industry, however. For instance, it has long been accepted in academic research that the optimal fanout of all wires in a switch-block (*switch-block flexibility* (F_s) [6]) is 3 [1]. It turns out that this largely underestimates the connectivity of more recent industrial architectures, such as the 28 nm Xilinx 7-Series [7].

1.1 Troubles with Resistance

Most significant changes were brought by the latest fabrication technologies, however. For instance, in the 10 nm Intel Agilex architecture, most switches are constrained to connecting wires that start and end at the height of a single ***Adaptive Logic Module*** **(ALM)** [8]. This minimizes the distances crossed at highly resistive low metal layers, which is crucial for allowing the new architecture to achieve a significant performance increase, despite poor resistance scaling.

Each of the one-ALM-high *planes* contains at most two wires of each of the available lengths, in each direction [8]. This means that creating permutation-defined switch-patterns, such as Wilton [2], between two groups of wires of particular length is no longer even possible. Hence, when revisiting the long-standing assumptions, it is important to go beyond reassessing which of the major pattern families [9] or their variants [10] performs best in the scaled context.

1.2 Managing Complexity

Increased metal resistance in new technologies means that each particular switch could have significantly different impact on the performance of the FPGA. However, significance of certain connections that the switch-pattern must support depends on the topology of the circuits that the architecture is intended for. Taking all these variables into account when designing a new switch-pattern quickly becomes difficult for manual design. Furthermore, both silicon technology and application domains continuously evolve, making it increasingly hard to develop intuition and keep it up to date. Availability of automated exploration techniques that can take into account all of the existing variables and constraints and easily adapt to future ones could thus be highly important for designing next-generation FPGAs.

In this article—a significantly extended version of the original that appeared at the 31st International Conference on Field-Programmable Logic and Applications (FPL) [11]—we introduce a novel technique for automated switch-pattern exploration that overcomes the longstanding issue of all prior ones: inefficient exploration of the search space based on using the routing algorithm as a black box to assess the quality of individually listed solutions.

1.3 Scope of the Article

The goal of this article is not to suggest specific rules for constructing switch-patterns in a particular technology, such as that they should contain a certain number of switches between wires of certain length. Rather, we propose a method that, given a fabrication technology and a set of target circuits, could be used to automatically produce switch-patterns that will enable routing circuits of the target set with low critical path delay. We provide ample evidence that the proposed method is effective. For instance, we demonstrate that it can produce in 10 hours a switch-pattern that outperforms the one produced by an alternative simulated-annealing-based approach in a comparable amount of time by close to 11% on geomean routed critical path delay. However, it must be noted that its present implementation has certain limitations that make it difficult to use it for designing interconnect for large FPGAs. As discussed at length in Section 15, the main reason is its inability to scale up to circuit sizes that capture the trends of current industrial designs and that would saturate the channel width of modern FPGAs [8]. Due to some orthogonal limitations in modeling the delays of hard IPs [12], at the moment, we are also not able to target designs that contain them.

We strongly believe that these limitations are not fundamental and that they can be overcome. The original PathFinder algorithm [13] would likely be unable to route large modern circuits in reasonable time without subsequent enhancements such as lookaheads [14, 15] and incremental rerouting [14, 16]. We believe that with certain enhancements (perhaps including those suggested in Section 15), our method—entirely based on PathFinder—will also be able to cope with large circuits. Until then, this article could be considered a proof of concept of a method that removes what we believe to be the most fundamental barrier towards automating exploration of switch-patterns. Constructing the exploration method around PathFinder—which is ultimately going to use the switch-patterns that the method produces—also gives confidence about soundness of the chosen heuristic. Nevertheless, as is often the case when designing heuristic algorithms, we cannot give any guarantees about the optimality of the produced solutions. The method's performance can only be experimentally compared to the alternatives, which is what we do in this work.

2 ARTICLE OUTLINE

The rest of the article is organized as follows: In Section 3, we give a brief review of *Island-Style* FPGAs [6] and *negotiated-congestion* routing [13], which form the background for the remainder of the article. In Section 4, we review the previously proposed methods for automated switch-pattern exploration and identify sources of their inefficiency in exploring the search space. In particular, we argue that their main problem is explicit listing of solutions and using the router as a black box to assess their performance. In Section 5, we proceed to briefly and intuitively introduce the main idea of the article, which lets us overcome the aforementioned problems of the prior methods. The proposed solution consists of (1) presenting the router with all switches that could possibly be fabricated instead of just those that belong to some particular solution—this effectively allows the router itself to design the switch-pattern—and (2) applying the negotiation mechanism in the opposite direction to make the signals reach a consensus on the switch types that should enter the final pattern. These two points are presented in more detail in Sections 7 and 8, respectively, after the switch-pattern design problem has been formally introduced in Section 6. Some practical aspects of the algorithm are presented in Section 9, while first results of its application are presented in Section 11. The algorithm is then compared in Section 13 with a simulated-annealing-based approach, inspired by prior work by Lin et al. [17]. In Section 14, we analyze different aspects of the algorithm in an attempt to provide a better understanding of how it operates. Runtime is discussed in detail in Section 15, followed by the final conclusions drawn in Section 16.

3 BACKGROUND

Before proceeding with introducing the main ideas of this work, we give a brief review of Island-Style architecture and negotiated-congestion routing. Readers familiar with these concepts may wish to skip this section.

3.1 Island-style Architecture

Figure 1 shows a portion of an Island-style FPGA. Conceptually, it consists of a regular grid of *logic clusters* (CLBs), each composed of N LUTs, that are surrounded by *routing channels* made up of prefabricated wires [6]. LUTs of a cluster can both take inputs from the channel wires and drive them. This is accomplished by driving each wire by a multiplexer; some of its inputs are provided by neighboring LUTs, while some are provided by other wires that end in its vicinity. Conceptually, these multiplexers are assumed to be located at the intersection of a horizontal and a vertical routing channel and are together, at one such intersection, said to form a *switch-block*. A cluster and an adjacent switch-block together form a *tile*, which is replicated to construct an FPGA grid. Hence, all switch-blocks throughout an FPGA are identical. In reality, wires are traced on top

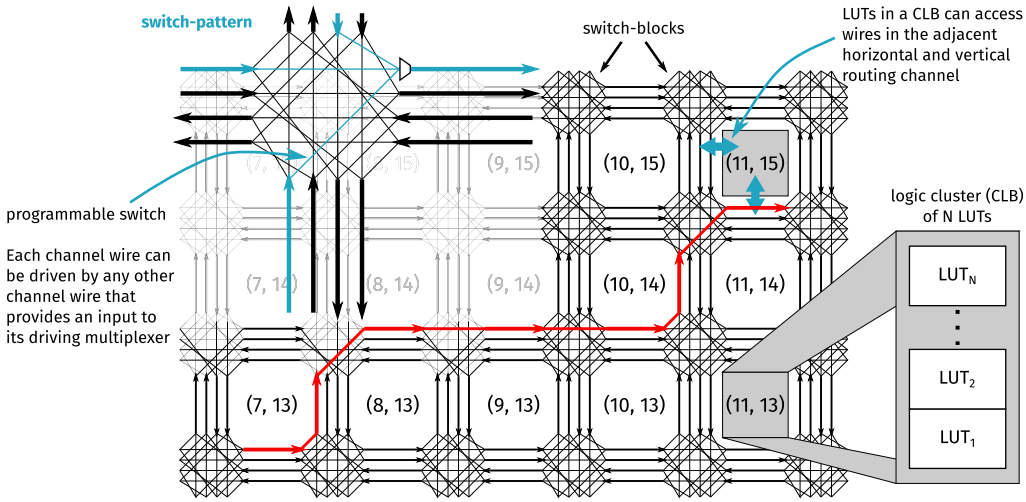


Fig. 1. Example of a simple Island-Style FPGA with a disjoint switch-pattern.

of the logic clusters and multiplexers of the switch-blocks are placed to their side [18]. This will become relevant and explained in further detail in Section 6.

The *switch-pattern* used in all switch-blocks of Figure 1 is shown in its top-left corner. It describes which channel wires can be connected together: a line connecting the head of a wire W_I^i to the tail of a wire W_I^o means that W_I^i can drive W_I^o; otherwise, there is no such posibility. In practice, this means that one of the inputs to the multiplexer driving W_I^o is provided by W_I^i. However, we say more generically that there exists a *programmable switch* between W_I^i and W_I^o. Since the switch-pattern describes connectivity in *all* switch-blocks, we say that switch-blocks are *instances* of a switch-pattern, or, in turn, that a switch-pattern specifies their *type*. Hence, we call a wire and a switch in a switch-pattern a *wire type* and a *switch type*, respectively, while we call a wire and a switch in one particular switch-block of a physical FPGA a *wire instance* and a *switch instance*, respectively. This will be formalized further in Section 6.

3.2 Placement and Routing

In a step of the FPGA CAD flow called *placement*, LUTs of the circuit being implemented on an FPGA are assigned a physical location on the FPGA grid [6]. Those LUTs that end up in the same cluster are said to have been *packed* together. Sometimes, packing constitutes a separate step in the CAD flow [16]. Once placement is complete, physical locations of both endpoints of every edge of the circuit being implemented on an FPGA are known and fixed. It is the duty of the *router* to connect these endpoints together by forming appropriate paths from wire instances connected by switch instances. Paths implementing edges with different tail nodes (different nets) must not intersect in a legal routing solution, as that would constitute a short circuit. When two or more such paths do intersect on a wire u, u is said to be *congested* [6]. The number of different nets using a wire u is typically called the *occupancy* of u, $O(u)$ [6]. We can then express the magnitude of congestion on u as $C(u) = O(u) - 1$, since every wire can legally carry one signal.

3.3 Negotiated-congestion Routing

Congestion negotiation was first introduced by McMurchie and Ebeling in their seminal article, which presented the *PathFinder* routing algorithm [13]. Likely the most popular open

ALGORITHM 1: Simplified PathFinder [6, 13]

Input: $G = (V, E)$—rr-graph, $E_c \subseteq V \times V$—all connections to be routed; **Output:** A routing tree of each signal

1: **function** CONGESTION_COST(u, s) ▷ computes the congestion cost of node u when routing signal s
2: **if** $u \in RT(s)$ **then return** 0 ▷ if u is already used by one connection of s, it can freely be used by another
3: **return** $b(u) \times (1 + p_{fac} \times O(u)) \times (1 + h_{fac} \times C_h(u))$ ▷ otherwise, account for congestion
4: **for** $u \in V$ **do**
5: $O(u) = 0; C_h(u) = 0$ ▷ set occupancy and historical congestion of all nodes to 0
6: **if** $(\exists v \in V)\,((u, v) \in E_c)$ **then**
7: $RT(u) = \{u\}$ ▷ initialize the routing tree of each signal
8: $i = 0; p_{fac} = p_{fac}^{init}$
9: **do**
10: **if** $i \geq$ max_iter **then return** UNROUTABLE ▷ no congestion-free routing was found in max_iter iterations
11: **for** $s \in \{u \in V : (\exists v \in V)\,((u, v) \in E_c)\}$ **do** ▷ all signals are ripped up and rerouted in each iteration;
12: ▷ modern incremental routers deviate from this [16]
13: **for** $u \in RT(s)$ **do**
14: $O(u) = O(u) - 1$ ▷ reduce the occupancy of all nodes used by the signal s that is ripped up
15: $RT(s) = \{s\}; O(s) = O(s) + 1$ ▷ rip up the signal
16: **for** $t \in V : (s, t) \in E_c$ **do**
17: $P =$ SHORTEST_PATH(s, t, $\forall u \in V : cong(u) =$ CONGESTION_COST(u, s)) ▷ (re)route the connection $s \rightarrow t$
18: **for** $u \in P$ **do**
19: **if** $\neg(u \in RT(s))$ **then**
20: $O(u) = O(u) + 1$ ▷ increase the occupancy of all nodes not already used by the signal s
21: $RT(s) = RT(s) \cup P$ ▷ add the connection route to the routing tree of s
22: **for** $u \in V$ **do**
23: $C_h(u) = C_h(u) + \max(0, O(u) - 1)$ ▷ update historical congestion
24: $i = i + 1$
25: $p_{fac} = p_{fac} \times p_{fac}^{mult}$ ▷ increase the penalty of using occupied nodes; $p_{fac}^{mult} > 1$ (1.3 is default in VPR [16])
26: **while** $\exists u \in V : O(u) > 1$ ▷ finish if there is no congestion
27: **return** $\forall RT$ ▷ return all routing trees

implementation of PathFinder is *VPR*, first developed by Betz and Rose [19], which introduced several refinements to the original algorithm. Only a simplified review of congestion negotiation is given in this section, focusing on aspects most relevant to this work. The reader should refer to the works of McMurchie and Ebeling [13], Betz et al. [6], and Murray et al. [16] for an in-depth discussion.

A negotiated-congestion router operates on the so called ***routing-resource graph*** **(rr-graph)**. In an rr-graph, each wire and each pin (endpoint of one of the circuit's edges after placement) is represented by a node, while each switch is represented by an edge [13]. A simplified version of PathFinder is shown in Algorithm 1. The algorithm proceeds iteratively by routing all connections of a circuit using the shortest path in the rr-graph between their respective endpoints (fixed during placement). This is designated by the loop starting at line 11, while the shortest-path search itself is performed on line 17. All signals are routed independently and hence their paths can intersect. As mentioned before, this constitutes a short circuit and must be avoided. The key to this lies in how PathFinder assigns costs to each rr-graph node. Namely, each node u has a *base cost*, $b(u)$, which determines how preferable it would be for any signal to use it, if congestion is entirely ignored. There are many ways to compute $b(u)$, some of which are discussed by Murray et al. [16]. This base cost is then multiplied by a product [6] of two additional costs: one directly related to *present* occupancy, $O(u)$ [13] and another directly related to historical congestion, $C_h(u)$ [13]. Computation of this *congestion cost* is performed on line 3. Occupancy is updated on lines 14 and 20: Whenever a signal's routing tree is ripped up, occupancy of all of the nodes of the tree is reduced by one; whenever an rr-graph node is added to a signal's routing tree, its occupancy is increased by one [6]. Finally, historical congestion of each node is updated on line 23 [6].

Because of the dependence of the cost of each node on its current occupancy, even though each signal is routed independently, it is motivated to deviate from its preferred path as determined by the base costs and avoid using the nodes already in use by other signals. Deviation from preferred paths happens only gradually, however: initially, the present occupancy coefficient p_{fac} is made very small [16] and increases exponentially with iterations (line 25); additionally, historical congestion is zero at first, but gradually increases, driving signals away from repeatedly congested areas [13]. Finally, to make the router timing driven, an additional term is added to the cost of each node:

$$cost(u)\Big|_{(s,t)} = crit(s,t) \times t(u) + (1 - crit(s,t)) \times cong(u). \tag{1}$$

Here, $t(u)$ is the intrinsic delay of the node u, while $crit(s,t)$ is the timing criticality of the connection (s,t) of the circuit [13]. The first term attempts to route more critical connections through faster wires, whereas for others, the second term dominates, causing them to release the congested wires to the more critical connections.

4 INAPTNESS OF THE BLACK BOX APPROACH

In this section, we analyze some of the existing methods applied to exploring FPGA architectures and identify their inefficiencies when applied to the problem of designing switch-patterns. Most of the conclusions about what constitutes a good FPGA architecture reached in the past 30 years came from applying a variant of the following approach:

(1) Select an architectural parameter p, the influence of which is to be assessed, and fix a range P for it.
(2) For every value in P, create an architectural model with p taking that value, run the CAD flow on a number of preselected benchmark circuits, and record some performance metrics (e.g., critical path delay and area).
(3) Choose the value (or a value range) of p, which optimizes the performance metrics.

For instance, optimal ranges of ***Look-Up Table*** **(LUT)** [20] and cluster [21] sizes were discovered in this manner. Given that LUT area increases exponentially with input count and that the size of a crossbar with fixed sparsity increases roughly quadratically with cluster size, it is not surprising that this parameter-sweeping approach was highly successful: Reasonable ranges of these parameters are very small and can easily be exhaustively explored.

4.1 How Large Is the Switch-pattern Search Space?

To better illustrate why a brute-force exploration approach cannot be applied to designing switch-patterns, let us first try to make a quick assessment of how large the search space for switch-patterns could be. Let there be 10 wires exiting a switch-block from one side and 30 entering it on the three remaining sides. Assuming that each multiplexer driving an exiting wire should be able to select from 6 incoming wires, there are $\binom{30}{6}^{10} \sim 10^{57}$ ways to form the pattern. This includes many pathological cases, where, e.g., some wires have no fanout, as well as isomorphic duplicates, but, among the 10 wires per side, it is likely that most will be of different lengths or coming from neighboring planes [8]. Such a large space clearly cannot be exhaustively explored. In fact, most of the well-known switch-patterns from the past, such as *Disjoint* [1], *Wilton* [2], and *Universal* [3], did not come from exploration at all, but from manual design. This is in stark contrast with the aforementioned experiments that revealed, e.g., the optimal LUT size ranges.

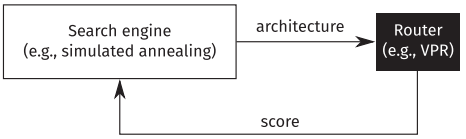


Fig. 2. An example of using the router as a "black box in the loop" to drive iterative switch-pattern improvement. At each iteration, a modification of the switch-pattern is proposed and a complete architecture model is generated. The performance metrics obtained from the router are used to decide on accepting the proposed modification. (Based on an approach proposed by Lin et al. [17].)

4.2 Black Box in the Loop

A step towards more efficient exploration of the switch-pattern search space is to iteratively improve a starting pattern based on postrouting performance metrics. This approach—illustrated in Figure 2—has been successfully used by Lin et al. [17] and subsequently by Shi et al. [22]. Starting from some, perhaps arbitrary, switch-pattern, a search engine—typically based on simulated annealing—proposes a modification that is then evaluated by the router on a preselected set of benchmark circuits. Performance metrics obtained from the router are then used to decide on whether to accept the proposed modifications. While the exploration is no longer brute-force, it is important to note that the algorithm still explicitly constructs a pattern and then uses the router as a complete black box, merely to obtain the performance metrics. In this article, we argue that this is, on one hand, a fundamental limitation and, on the other hand, completely unnecessary. Given that routing a modern circuit even with a state-of-the-art router can easily take minutes, if not hours [16], the number of modifications that can be evaluated in this manner is rather limited.

4.3 Proxy Oracles

To speed up the evaluation process, which is the bottleneck of the black-box-in-the-loop approach, some authors have attempted to substitute the router for a proxy oracle that tries to predict the score that the router would output, in a fraction of the time. An example of such an approach was proposed by Petelin and Betz [23]. Although appealing, proxy oracles can only reduce exploration time by a constant factor; evaluating each switch-pattern might be significantly faster, but the number of switch-patterns that have to be explicitly listed to cover any sizeable fraction of the search space remains prohibitively large. Another downside is that while it is possible to assess how closely oracles mimic the router on a limited set of test architectures, it is difficult to claim that they appropriately approximate the router for all architectures that may occur during the course of the exploration. Failure to do so could silently lead the search astray.

An interesting approach to proxy design was also introduced by Lemieux and Lewis [24]. Namely, they first limited the set of switch-patterns to be explored to those that can be described using *permutation functions* [24]. Then they conjectured that a certain characteristic of a switch-pattern that can be quickly measured (*diversity*) has an important influence on its routability. Instead of optimizing the performance metrics obtained from the router or some proxy trying to mimic it, they used this characteristic as the maximization objective. Given the fast computation of the objective and a search space significantly reduced by the initial constraints, it was possible to find solutions maximizing the chosen objective using randomized and even brute-force search. While such an approach can help to understand which characteristics lead to highly routable switch-patterns, proposing the characteristics is left to the human designer. Similarly, constraining the search space *a priori* can be very useful for allowing it to be searched in practice, but it is often difficult to make sure that the imposed constraints do not exclude promising solutions.

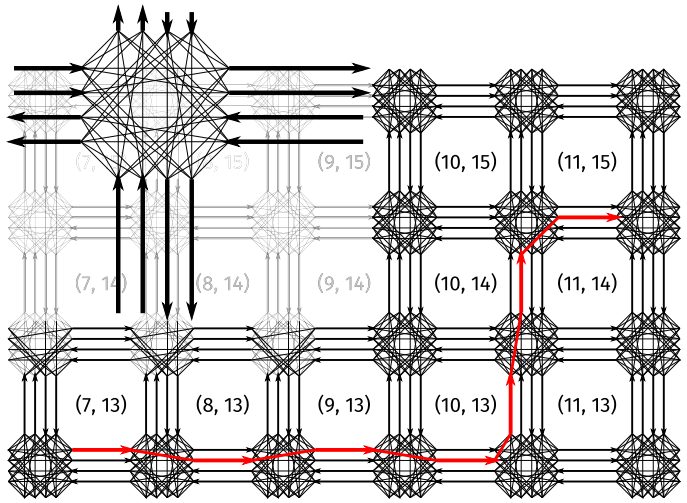


Fig. 3. Architecture of Figure 1, but with all possible switches between channel wires. One of the main downsides of the disjoint switch-pattern identified in prior work was the lack of a possibility to switch between different tracks in a channel [2]. If track changes are needed for implementing a circuit, the router will be able to determine it in the architecture model that holds all possible switches.

For instance, permutation functions as defined by Lemieux and Lewis assume that each incoming wire has exactly one target at each of the three remaining sides of the switch-block. However, some newer industrial architectures do not meet this constraint [7].

Instead of proposing another method to bring down routing time and enable exploration of more points of the search space, we propose a method that altogether removes the need to explore individual points. While we believe that this is an important step towards scalable automated switch-pattern design, as we have already mentioned, for the reasons discussed at length in Section 15, the proposed method does not fully achieve this goal yet.

5 MAIN IDEA

Before diving into technical details, including a formal definition of the switch-pattern design problem, let us first briefly explain the main idea behind the proposed approach.

5.1 Implicit Search Space Representation

The particular switch-pattern of Figure 1 is known as *subset* or *disjoint* [1]. Given this switch-pattern, the shortest path through the channel wires that connects a source in tile (7, 13) to a sink in tile (11, 15) could be the one depicted in red in the figure. Figure 3 shows the same portion of almost the same simple FPGA architecture, with one major difference: Instead of the switch-blocks containing switches corresponding to the disjoint switch-pattern, they contain all switches that could potentially be fabricated. When routing the same signal from tile (7, 13) to tile (11, 15), it is possible that the router discovers that changing tracks is beneficial to avoid congestion, as depicted by the shortest path in the figure. Note that if the router sees all switches that could be fabricated, then there is no need for a designer to guess that track changing is useful and construct a switch-pattern that allows it, nor is there a need for some randomized exploration process to propose a modification that enables track changes; the router itself can select the appropriate switches and reap the benefits of track changing where they exist. In other words, presenting the router with the entire search space embedded in the rr-graph lets it explore this space on its own, alleviating

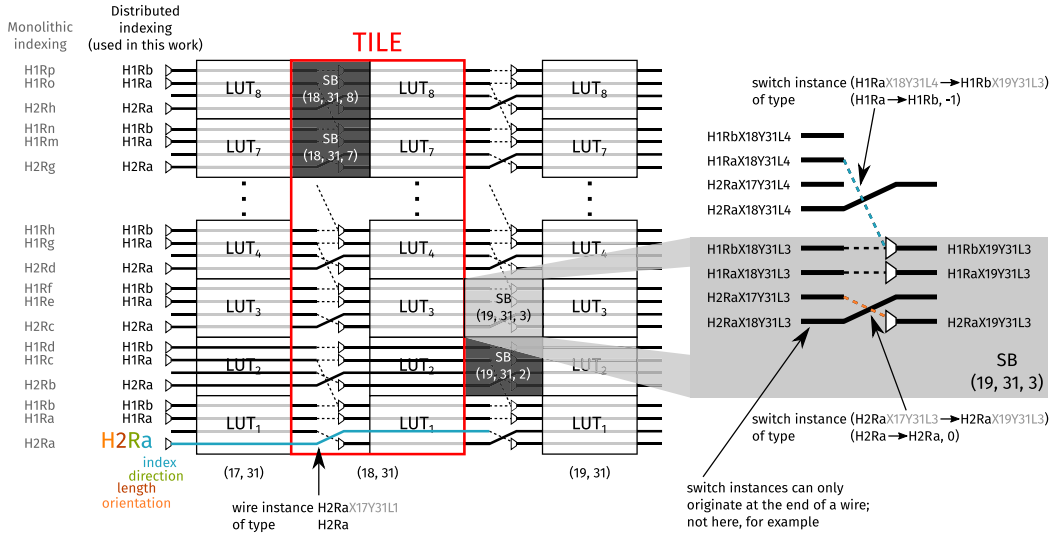


Fig. 4. Illustration of definitions. A switch-block (SB) is defined at the level of one LUT in one tile. All switch-blocks are identical, apart from those at the edges of a cluster, where inputs from neighboring tiles are omitted.

the need to explicitly construct switch-patterns during automated exploration and thus removing the aforementioned scalability issues.

5.2 Negotiating Switch Types

Without any constraints, the router is free to use different switch types in different switch-blocks, while the switch-pattern that is fabricated must be common to all. Hence, it is crucial to be able to find a minimal set of switch types that allows all connections to be appropriately realized in every tile. In negotiated-congestion routers, evolving congestion costs allow the signals of a circuit to negotiate which ones will deviate from their respective shortest paths and *spread* to less congested wire *instances* (Section 3.3). As will be described in greater detail in Section 8, we use the same principle of evolving costs, only applied in reverse, to allow the signals of a circuit to negotiate which ones will deviate from their respective shortest paths and *concentrate* on a minimal set of switch *types* that will enter the final pattern.

6 PROBLEM DEFINITION

Let us now precisely define the problem tackled in the rest of the article. Since our goal is to design switch-patterns, we assume that the rest of the routing architecture—namely, the connection-block, the intracluster interconnect, and the wires in the routing channels—is given and fixed. We have already provided an informal definition of a switch-pattern and a switch-block in Section 3. However, as we already mentioned there, instead of the channel wires surrounding the logic clusters, they are actually traced above them [18]. This is illustrated in Figure 4; for the sake of clarity, only horizontal wires going right are shown. In the most recent FPGA architectures, designed for scaled technologies for which this work is the most relevant, channels are composed in such a way that the same number of wires of the same length and direction start and end in the vicinity of each LUT in a tile [8]. This is also illustrated in Figure 4, where next to each LUT, wires H1Ra, H1Rb, and H2Ra start. These three wires, replicated at the height of every LUT together with the corresponding ones running leftward, give a combined effective channel width

$OLDI$	A wire *type* with orientation $O \in \{H, V\}$, standing for *horizontal* and *vertical*, respectively; length $L \in \mathbb{N}$ corresponding to the number of tiles between its start and end; direction $D \in \{L, R, U, D\}$, standing for *left*, *right*, *up*, and *down*, respectively; and index $I \in [a..z]$. In Figure 4, 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). Constant N stands for cluster size. In Figure 4, H2RaX17Y31L1 is a wire of type H2Ra, starting at LUT 1 of tile (17, 31).
$(W_I^i \rightarrow W_I^o)$	A switch *instance* providing a programmable connection between wire instances W_I^i and W_I^o. In Figure 4, (H1RaX18Y31L4 → H1RbX19Y31L3) provides a connection from the end of the H1Ra wire starting at LUT 4 of tile (18, 31) and the H1Rb wire starting at LUT 3 of tile (19, 31).
$(W_T^i \rightarrow W_T^o, d(l^i, l^o))$	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 Figure 4, (H1Ra → H1Rb, -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 $(x, y, l) = (19, 31, 3)$ is indicated in Figure 4.
$SP(x, y, l)$	*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 local switch-pattern. Constant N stands for cluster size.

of $8 \times 2 \times (1 + 1 + 2) = 64$—same as if the horizontal channel was composed of 16 H1R and 8 H2R wires specified under "Monolithic indexing."

In such an architecture, each wire *type* can be defined by its length, direction, and index within the LUT-height (plane). Then, each wire *instance* can be defined by specifying its type along with coordinates of the tile and index of the LUT at which it originates. Similarly, a switch *type* can be defined by specifying the types of wires that it connects, along with the offset between their respective LUTs. Since, in scaled technologies, loading wires at non-terminal tiles damages performance too much and is thus no longer practiced [8], there is no need to specify the offset between the origin tiles of the two wires connected by a switch type—it is assumed in the length and direction of the driving wire.

This way of defining switch-patterns is very practical, since due to high resistance of lower layers of metal in scaled technologies, it is not feasible for switches to span a large number of LUTs. Let us now formalize these concepts that we draw from the *bundles* and *planes* of Agilex [8] by introducing some notation that we will use throughout the article:

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

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.

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 $E_a \subseteq E$, such that all circuits in C can be routed and their critical path delays minimized.

7 BASIC ALGORITHM

As mentioned in Section 5, our proposed method relies on implicitly representing the entire switch-pattern search space by embedding it in the rr-graph. It then simply observes the usage statistics of the different switch types across all switch-blocks and in all circuits used in exploration and constructs the pattern from the most-used ones. This is more precisely defined by Algorithm 2. All switch types that can be fabricated are added to the rr-graph on line 1. Initially, the switch-pattern E_a is empty and all switch types are allowed to enter it (line 2). The algorithm then proceeds iteratively, first routing the benchmark circuits chosen for the exploration using PathFinder on

line 4 (see Algorithm 1) and then adding to the pattern on line 6 all of the switch types with usage $1/\theta$ of the maximum over all that are not already in the pattern. Here, the *adoption threshold*, θ, is a parameter. Upon growing the switch-pattern, costs of all added switch types are reset to 0 on line 7, because once a switch type has been marked for fabrication, it can be used by the router for free in subsequent iterations of the algorithm—using it no longer implies any increase in pattern size. The algorithm stops when the router no longer uses any switch types that are not already in the pattern. We note here that once a switch type enters the pattern, it is never removed from it. It may be beneficial to revisit this in future work, but it guarantees that the algorithm always converges: In the worst case, all possible switch types are taken ($E_a = E$). Preventing this situation will be the main focus of the next section. We first further discuss the algorithm's general structure.

7.1 Benefits of Iteration

In principle, usage statistics obtained from a single run of the router could already provide valuable information about which switch types would be useful in the switch-pattern. In fact, prior research has successfully relied on usage to design novel interconnect architectures [25]. Nevertheless, there are two important benefits of progressively growing the switch-pattern. First, after each run of the router, some switch types will have a significantly higher usage than others and can thus be clearly deemed useful. An example of this is illustrated by the orange curve of Figure 7. Once these switch types are adopted at a lower cost (note the small initial cost used on line 1 to distinguish switch types not yet in the pattern), though, it may happen that more signals will use them instead of other switch types, thus leading to minimization of the entire pattern. Second, in between iterations, physical optimization of the switch-block can be performed by changing the positions of different multiplexers, depending on which switch types were added to the pattern. Similarly, up-to-date implications on delay and area increase of choosing each switch type can be presented to the router in the subsequent iterations. We will discuss this in more detail in Section 9.1.2.

7.2 Shortcomings of Uncompressed Usage Statistics

The idea of constructing the switch-pattern from the post-routing usage statistics relies on the intuition that the router itself will be able to best determine which switch types are useful for routing the given circuits. However, since it greedily routes each connection of the circuit using a shortest path in the rr-graph (line 17 of Algorithm 1), independent of others, by default, it has no incentive to maximize the number of common switch types between the routes of different nets, which would lead to minimizing the switch-pattern size. Before suggesting a remedy to this problem, let us first illustrate it on an example. Figure 5 depicts three different nets being routed through three different switch-blocks (note the different tile coordinates). As all three nets can arbitrarily choose the switch instances they take, for they all seem equally good, it is possible that

ALGORITHM 2: Simple Greedy

Input: $\theta \in \mathbb{R}^+$—switch adoption threshold; **Output:** switch-pattern

1: Add all $e \in E$ to the rr-graph at cost $\varepsilon \in \mathbb{R}^+$ ▷ represent in the rr-graph all switch types that can be fabricated
2: $E_a = \{\}, E_p = E$ ▷ at the beginning, no switch type is in the pattern to be fabricated
3: **do**
4: Route the chosen benchmark circuits
5: $U_{max} = max(\{U(e) : e \in E_p\})$ ▷ find the maximum usage among all switch types not yet in the pattern
6: $E_a = E_a \cup \{e \in E_p : U(e) \geq U_{max}/\theta\}$ ▷ extend the pattern by all switch types with usage $\geq 1/\theta$ of the max.
7: Set cost of all $e \in E_a$ to 0 ▷ switch types already in the pattern can be used for free in subsequent iterations
8: $E_p = E \setminus E_a$ ▷ switch types already in the pattern cannot be added again
9: **while** $\exists e \in E_p : U(e) > 0$ ▷ if the router used only switch types already in the pattern, stop
10: **return** E_a

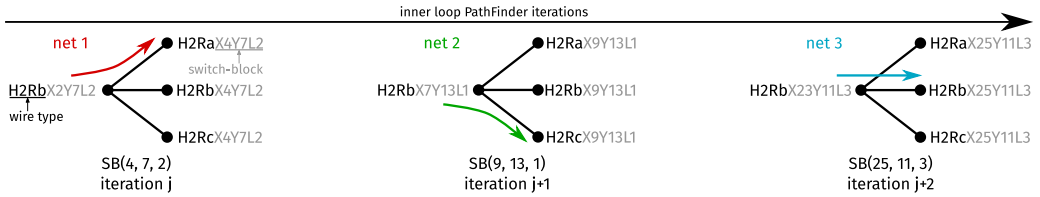


Fig. 5. An example of usage spreading over multiple switch types. Colored arrows mark the paths chosen by the router for three different nets passing through three different switch-blocks in three different regions of the FPGA. Each net uses an instance of a switch of a different type, even though this may not have been necessary. As a result, a switch-pattern common to all three switch-blocks would need to contain all three switch types even if one would have sufficed.

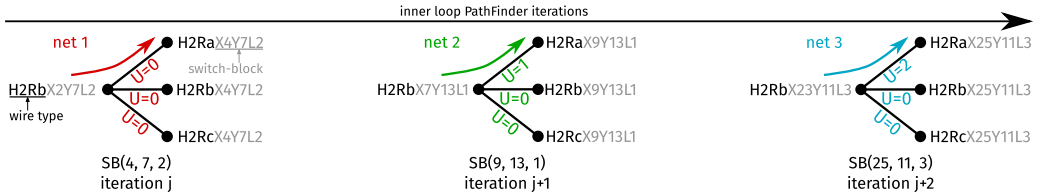


Fig. 6. Inversely relating switch instance cost to its type's usage across all switch-blocks motivates nets to concentrate on the same switch types. Current usage of each switch type at the time of routing of each net is depicted next to the corresponding edge.

usage is spread equally among the three switch types. On arriving at line 6, Algorithm 2 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.

8 TURNING PATHFINDER UPSIDE-DOWN

In this section, we introduce a remedy to the above problem: using the principles of *congestion negotiation* [13] to make the nets reach a consensus on which switch types are really important for routing a given circuit.

8.1 Avalanche Costs

Figure 6 shows the same routing process as Figure 5 with one important difference: Switch instance costs are no longer constant, but inversely related to their type's usage, indicated on the corresponding edge. For the first net that is routed, nothing changes: It still sees the same cost at all three switch instances and freely chooses one of them. 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 → H2Ra, +0) becomes still smaller, so it is even more inclined to use it. In subsequent iterations, the router will rip up and reroute nets, leading some of them to choose switch types that have in the meantime become cheaper than the ones they chose in the previous iterations when the cost differences among switch types may have not been as pronounced. This will create an avalanche effect, where the positive feedback keeps reducing the cost of switch types 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.

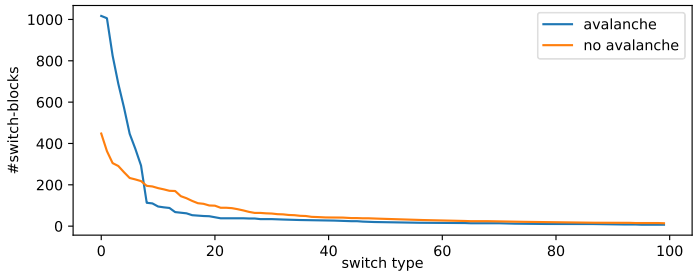


Fig. 7. Concentration effect achieved by the avalanche costs. Usage of the 100 most used switch types, out of the 564 available in one particular experiment, is shown for the case when avalanche costs are enabled and disabled, respectively. The area under the two curves is not identical, as switch type concentration also changes the total number of wires used for routing.

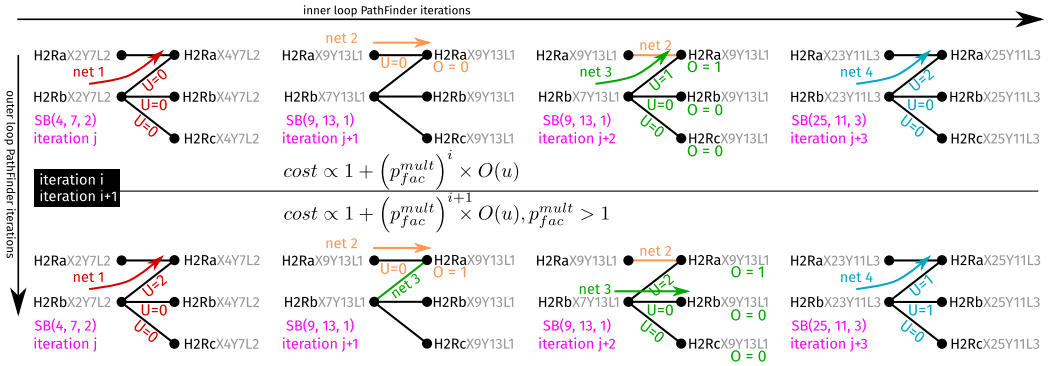


Fig. 8. Too much concentration on switch types can lead to congestion on wire instances. As the cost of using occupied wires (indicated by $O > 0$) rises over time (through exponential increase of p_{fac} [6]), at some point, a net will choose a less-used switch type. Eventually, the two effects balance out, producing a legal routing with a minimized number of switch types.

Figure 7 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. It is interesting to note that if, for example, the cost of the switch type (H1La → H1Lb, +0) drops significantly below the cost of (H2La → H2La, +0), then more noncritical nets may choose to use four H1L wires instead of two H2La wires, thus increasing the total usage compared to the situation where the cost differences did not exist. This effect causes the area under the blue curve of Figure 7 to be larger than that under the orange curve.

8.2 Negotiating Both Congestion and Switch Presence

In Section 3.3, we have seen that PathFinder gradually increases the cost of congested wire *instances*, pushing the nets towards a consensus on which ones will deviate from their desired paths, to spread congestion to other wires and eventually eliminate it [13]—making the same *instance* choices as other nets is *penalized*.

Inversely relating the cost of switch *types* to usage in avalanche costs makes the principle act in the opposite direction, causing a consensus on concentration, instead of spreading—making the same *type* choices as other nets is *rewarded*.

Let us see through the example of Figure 8 how these two directions of the same principle naturally act together. At routing iteration *i*, *net 3* may choose to take the switch instance

(H2RbX7Y13L1 → H2RaX9Y13L1), as it is cheaper because its type, (H2Rb → H2Ra, +0), is already used by *net 1*. This causes congestion on wire H2RaX9Y13L1 already in use by *net 2*. However, in the subsequent iteration, p_{fac} will be increased by a factor of $p_{fac}^{mult} > 1$ (line 25 of Algorithm 1), in turn increasing the cost of congestion on any wire instance (line 3 of Algorithm 1). Hence, even though the switch instance (H2RbX7Y13L1 → H2RaX9Y13L1) will still be cheaper than (H2RbX7Y13L1 → H2RbX9Y13L1) in the next iteration, the router will choose the latter to avoid congestion on H2RaX9Y13L1.

8.2.1 Can Congestion Elimination Be Guaranteed? To guarantee that avalanche costs will never prevent congestion resolution, we must ensure that this tipping point when penalization of congestion on wire instances surpasses the reward of concentration on switch types always occurs. Given that congestion penalization is not bounded from above, due to the exponential increase of p_{fac} (line 25 of Algorithm 1), it suffices to ensure that the difference between the maximum and the minimum switch instance cost (which constitutes the maximum switch type concentration reward) is bounded. As we will see in Section 8.3, we make the maximum switch instance cost—corresponding to unused switch types—constant, while we prohibit the avalanche costs from dropping below zero. Hence, the above requirement is satisfied and avalanche costs do not prevent congestion from being eliminated, though they may increase the number of iterations (line 9 of Algorithm 1) needed to achieve this. Further details on this problem will be provided in Section 15).

8.2.2 Is Congestion Elimination Always Necessary? When PathFinder is being used for implementing a circuit on an existing FPGA (its usual intended use), it is necessary to eliminate all congestion—otherwise, the routing is illegal. However, here, we are not using PathFinder to implement a circuit on an existing FPGA but to design a new switch-pattern by observing which switch types are most useful for routing the circuits selected for the exploration. This may become apparent long before congestion is fully eliminated. Especially in the early iterations of Algorithm 2, switch types that once surpass the adoption threshold are unlikely to drop below it again. As discussed further in Section 15, stopping PathFinder at this point could be used to greatly speed up the exploration process.

8.3 Functional Form of Avalanche Costs

As discussed in Section 8.1, avalanche costs should be high for switch types that are unused and drop in proportion with usage of the particular type. Additionally, not to prevent congestion resolution, they must be bounded from both below and above. To satisfy these criteria, we use a functional form similar to congestion costs of PathFinder (Section 3.3):

$$a(u) = max(0, s(u) - (a_p \times U(u) + a_h \times U_h(u))). \tag{2}$$

The $s(u)$ term in Equation (2) is the starting cost assigned to the given switch type, 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 type, $U(u)$, while a_h determines how quickly it drops as a function of its cumulative historical usage, $U_h(u)$.

Usage tracking is completely analogous to occupancy tracking of Section 3, and $U(u)$ is updated each time a net is routed (lines 14 and 20 of Algorithm 1). Similarly, historical usage tracking is completely analogous to historical congestion tracking, and $U_h(u)$ is updated at the end of each routing iteration (line 23 of Algorithm 1). The main difference, however, is that unlike the occupancy trackers that 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 (Figure 6) and eventually reaching a consensus on which switch types will enter the pattern that is common to them all.

We note here that there are many other functions that would satisfy the requirement of avalanche costs dropping in proportion with switch type usage and being bounded from both sides. Adding together the present and historical usage terms as in Equation (2) has a benefit of providing a relatively easy way of tuning the coefficients. This will be discussed in Section 14.1. It also has a downside, compared to the analogous product used in the congestion costs of PathFinder: The historical term quickly dominates. This may lead to a reduced capacity for driving nets to common switch types. Aside from multiplying the two terms or making $a_p \gg a_q$, a possible remedy for this could be to update the historical usage using an exponential decay, giving higher importance to more recent history. Nevertheless, investigating the effectiveness of functional forms other than the one of Equation (2) goes beyond the scope of the present work.

8.4 A Note on Implementation

Most implementations of PathFinder, including the one in VTR-8 [16] that we use in this work, assign weights only to the nodes of the rr-graph. To retain the existing data structures, we simply split each edge that represents a potential switch instance by an additional node and assign the appropriate avalanche cost to this node. In particular, we compute the congestion cost of these virtual nodes as follows:

$$cong(u) = b(u) = a(u). \tag{3}$$

Splitting edges with additional nodes doubles the total edge count in the rr-graph and drastically increases its node count. As will be discussed in Section 15, this has a significant impact on the exploration time. However, implementational effort needed to adapt VTR's algorithms to accept both node and edge weights went beyond the scope of this work.

8.5 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 switch types with otherwise low usage. Critical path delay of a typical circuit is on the order of 10^{-9}. In our experiments, we have determined that the starting avalanche cost is best set to the same order, or larger, even up to 10^{-7}. Under those circumstances, linearly scaling avalanche costs by $(1-crit)$, like congestion costs in Equation (1), would not give enough freedom to the critical paths to choose switch types with low usage; switch cost variations would simply overshadow the timing optimization.

Another problem with linear scaling is that somewhat critical (e.g., criticality 0.5) nets are given unfair advantage in choosing switches compared to nets that are just slightly less critical. While exponentiating the criticality can help mitigate this second problem, it further worsens the first, as shown by the blue, orange, and green curves of Figure 9.

To provide a solution to both problems, we designed a function represented by the red curve of Figure 9. The curve shows a relatively wide, flat range of very small values close to criticality of 1, which allows for the critical paths to actually be optimized. At the same time, there is a steep rise in the value of the function as criticality drops, which prevents the nets with comparatively low criticality from unnecessarily increasing the number of used switch types.

The combined timing and avalanche cost assigned to a node splitting an edge that represents a potential switch is

$$cost(u)\Big|_{(i,j)} = t(u) + e^{\left(\frac{ln(sc/s)}{max\_crit^{\beta}} \times crit(i,j)^{\beta}\right)} \times a(u). \tag{4}$$

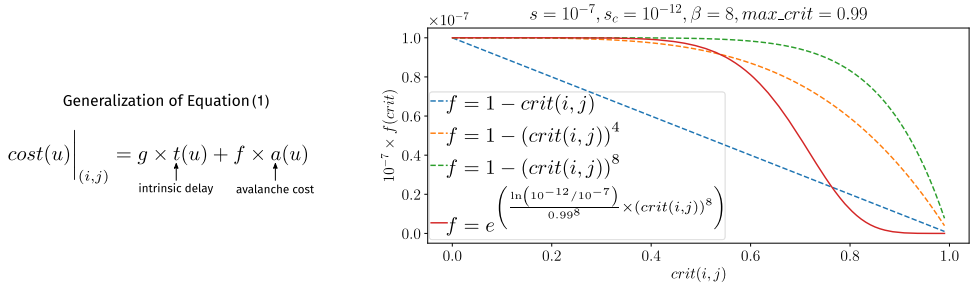


Fig. 9. Comparison of functions for criticality-scaling of avalanche costs (generalizations of Equation (1)). The proposed function of Equation (4) (solid red line) allows for precise tuning of the avalanche cost that the most critical nets perceive, so the timing requirements are sufficient to motivate them to use switch types 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 [16] lack these features (dashed lines).

Here, s_c is a parameter determining the perceived avalanche cost of a potential switch when routing the most critical possible net, with criticality *max_crit* (a standard parameter of VPR [16]), and β is a criticality exponent used to tune the selectivity of the function. Approximate delay contribution of the switch to the wire that is driving it is represented by the term $t(u)$, modeled as described in Section 9.1. We do not scale it by criticality of the net being routed, because all nets—regardless of their criticality—should be aware of the implications of including a switch type in the pattern.

9 COMPLETING THE ALGORITHM

The complete algorithm is almost identical to Algorithm 2, apart from the fact that routing on line 4 is performed using a modified version of VTR 8 [16], which incorporates the avalanche costs of Section 8. Another difference is that if there are switch types that got their avalanche cost reduced to zero in the current iteration, then 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 conceptually equivalent to resetting their costs to 0 (line 7) but has a practical benefit of reducing the size of the rr-graph.

9.1 Conveying Physical Information

Apart from the delays of the routing wires, necessary for proper timing optimization, the router must be aware of the implications on the architecture's performance of using switches of a type that is not yet in the pattern. Before each iteration of the algorithm, we run a physical modeling and optimization flow to provide this data for the modified switch-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 7) is sufficiently small to prevent adoption of too many switches between reevaluations of the physical model of the switch-block, then the simple approach of only informing the router about the impact of each switch in isolation, through the $t(u)$ term of Equation (4), should suffice.

9.1.1 Modeling Flow. To extract delays of the channel wires, we rely on a modeling flow developed in our previous work [12]. The flow assumes a floorplan similar to that of the Stratix FPGAs [18], where LUTs are stacked on top of each other, while the routing multiplexers are

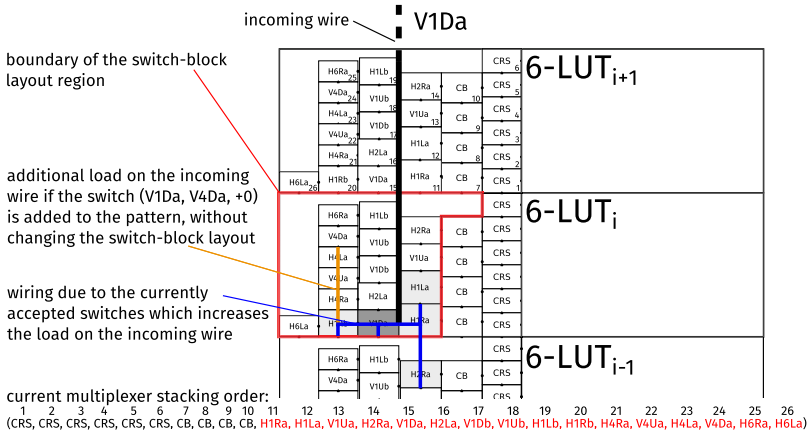


Fig. 10. Floorplan construction. Not drawn to scale.

arranged in columns padded to their left. Figure 10 depicts one such floorplan. The crossbar multiplexers are placed immediately next to the LUTs, followed by the connection-block multiplexers, and then finally the switch-block ones. Each multiplexer column is filled from the bottom up, until its height matches the height of the adjacent LUT. Only then a new column is started. Placement of multiplexers next to each LUT is identical.

9.1.2 Multiplexer Position Optimization. Precise positions of all multiplexers allow for accurate modeling of intra-switch-block wiring (depicted in blue in Figure 10, for one source channel wire), which in turn allows for correctly taking into account the influence of this wiring on the delay of the channel wires. However, as the pattern evolves during the course of the avalanche search, positions of multiplexers in the tile floorplan may become suboptimal. In our previous work, multiplexers were stacked in a fixed order, derived from their input count [12]. Now, we adapt the order 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-switch-block 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 [26]. During multiplexer position optimization and routing wire delay measurement, only those switch types that 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 [12]. However, annealing uses approximate delays obtained from a polynomial fitted to a set of SPICE simulations, which relates the total length of the intra-switch-block wiring that a routing wire drives to the increase in its delay. Output of the same model is used to set the timing costs of the potential switches during routing.

9.2 Preventing Overspecialization

To prevent the resulting pattern from being specialized to a particular placement, we replace the circuits using a different placement seed at the start of each iteration of the search algorithm (just before line 4 of Algorithm 2).

Line 4 of Algorithm 2 does not specify how multiple circuits are routed before measuring the usage statistics needed for deciding which switch types are added to the pattern. Two possible

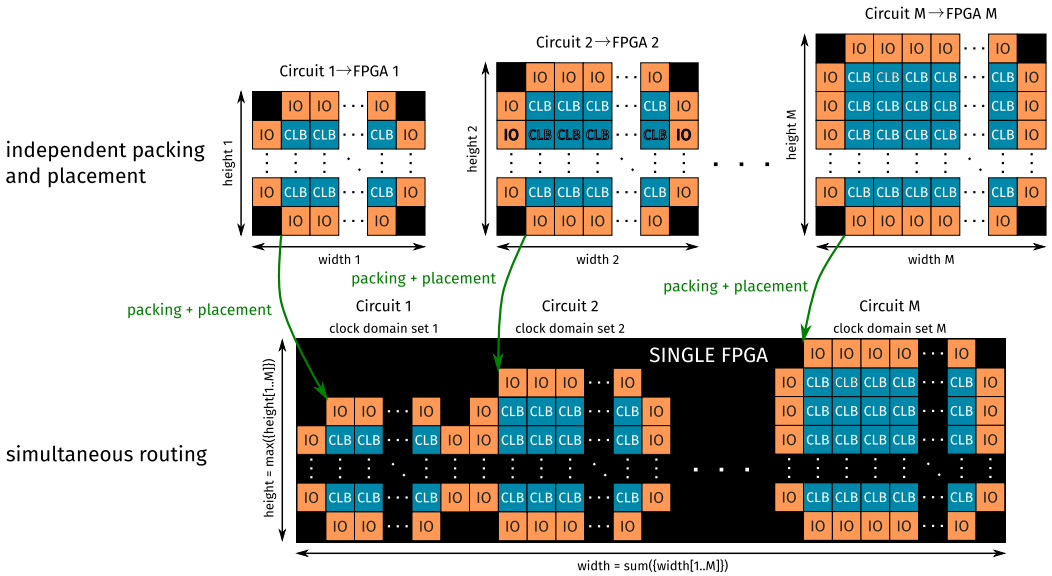


Fig. 11. To enable joint negotiation of switch-presence among multiple circuits, we route them simultaneously.

ways of doing this will be presented in detail in subsequent sections. Namely, in Section 12, a switch-pattern is first obtained by performing the exploration on a set of circuits C_1 and is then used as a starting point for a continued exploration on a set of circuits C_2. This can be highly beneficial to reducing the runtime of exploration on large circuits, as discussed later, but it has a downside of giving preference to circuits from C_1 in deciding which switch types should enter the pattern. An alternative approach, presented in Section 14.2, is to route multiple circuits independently in parallel on line 4, combining the usage statistics observed on each of them before proceeding to switch type selection. While this avoids the problem of *a priori* favoring one set of circuits, it suffers from the usage spreading problem of Figure 5: introducing avalanche costs enabled nets passing through switch-blocks in different regions of the FPGA to negotiate which common switch types they will use; however, if the three nets in the example of Figure 5 belong to three independently routed circuits, then this negotiation is no longer possible and they may again end up using different switch types even if a common one would have sufficed.

Note that these problems do not exist in the black-box-in-the-loop approach (Section 4), since in that case all circuits are routed on one and the same switch-pattern. To prevent the problems in the context of avalanche search as well, here, we simply route multiple circuits simultaneously, allowing their nets to jointly negotiate the presence of different switch types. The implementational details of this are illustrated in Figure 11. First, each circuit used in the exploration is packed and placed independently on the smallest FPGA that can fit it, as determined by VPR [6], with its logical width and height adjusted to make the physical ones roughly equal [12]. Then, a new large FPGA is created, so its height equals the maximum height of all individual FPGAs and its width equals the sum of the widths of all individual FPGAs. The smaller individual FPGAs are placed on this larger one, much like if it were a passive interposer, but without any connections between dies. The packing and placement of each circuit are transferred to their respective isolated region on the new FPGA and assigned their own set of clock domains. The netlists are then merged and routed simultaneously, allowing them to share avalanche costs. Since different circuits are placed in isolated regions with no connectivity between them, nets of one circuit cannot cause

Table 1. Properties of the Different Patterns; *Manual* and *Manual Annealed* Will Be Discussed in Section 13

	avalanche			greedy			truncated greedy			manual [12]			manual annealed		
#iterations	36			228			54						10,000 moves		
#switch types	78			438			78			180			210		
average →	fi	fo	t[ps]	fi	fo	t[ps]	fi	fo	t[ps]	fi	fo	t[ps]	fi	fo	t[ps]
H1	5	3	13.9	31	25	23.1	6	3	13.2	10	10	16.0	13	13	19.6
H2	5	4	16.8	28	28	31.6	5	5	18.1	11	11	21.3	14	11	24.1
H4	4	7	27.4	21	27	43.2	4	6	25.7	11	11	30.8	16	12	32.1
H6	5	5	35.7	19	25	59.6	2	6	35.7	11	11	43.1	9	13	47.3
V1	7	6	21.8	38	31	35.5	8	7	22.1	12	12	24.6	14	15	29.2
V4	2	5	70.1	12	27	97.5	1	4	67.0	13	13	74.3	13	15	86.8
W(tile)	6,792 nm			8,904 nm			6,816 nm			7,464 nm			7,488 nm		
CPD	1.38 ns			1.71 ns			1.38 ns			1.46 ns			1.55 ns		

congestion in others. Similarly, since each clock domain is individually optimized, timing characteristics of each circuit are preserved.

10 EXPERIMENTAL SETUP

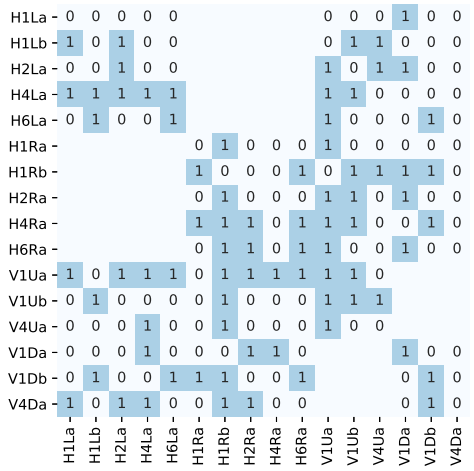
All experiments are performed on an architecture with eight 6-LUTs in the cluster and a channel composition reminiscent of that of Agilex, but for the longest wires [8]: $2\times$H1, H2, H4, H6, $2\times$V1, V4. These wires are repeated for each LUT of the cluster, leading to an equivalent width of a horizontal channel equal to $2\times8\times(1+1+2+4+6)=224$ and an equivalent width of the vertical channel equal to $2\times8\times(1+1+4)=96$. As mentioned in Section 6, wires are only allowed to drive other wires from their end tile. Without loss of generality, we consider only switch types with LUT offset $\in\{-1,0,1\}$ (Section 6) and prohibit switch types to a target wire going in the direction from which the driving one came [7]. This results in 564 available switch types. 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 [12]. Avalanche parameters are set as described in Section 14.1.

11 EFFECTIVENESS OF AVALANCHE COSTS

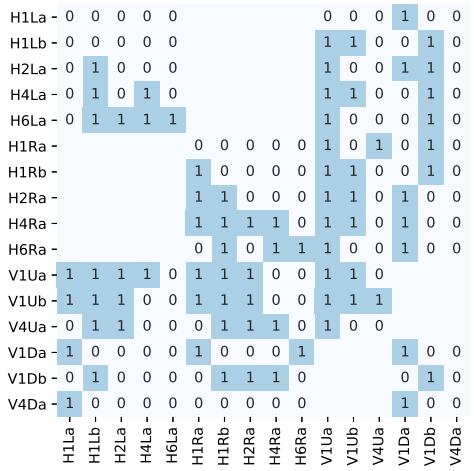
In this section, we assess the effectiveness of the proposed avalanche search method against the simple greedy algorithm of Section 7. Instead of introducing explicit ε costs without a physical meaning to the greedy algorithm, we use the timing costs of the switches equally visible to all nets, regardless of criticality (Equation (4)). 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 performance was done on all MCNC circuits, but for the pin-bound *dsip*, *des*, and *bigkey*. We note that the results reported here differ slightly from the ones reported in the original paper [11], because in the original paper, periodic rip-up did not affect all uncongested connections, leading to some missed opportunities for increased concentration. With periodic rip-up fully enabled (Section 15.3), a more compact pattern with better delay was obtained, at the expense of lower routability. This will be addressed in Section 12.

11.1 Direct Comparison with Greedy

Avalanche search converged after 36 iterations, accumulating 78 switch types, while greedy search converged only after 228 iterations, accepting 438 switch types (Table 1). This demonstrates that projected delay contributions of individual switch types alone are insufficient to deter the router from using them. The large number of switch types in the greedy pattern resulted in both a large increase of the tile width and the average fanin and fanout of channel wires. This in turn led to a large increase of average wire delays and the routed critical path delay (Table 1).



(a) Avalanche.



(b) Truncated Greedy.

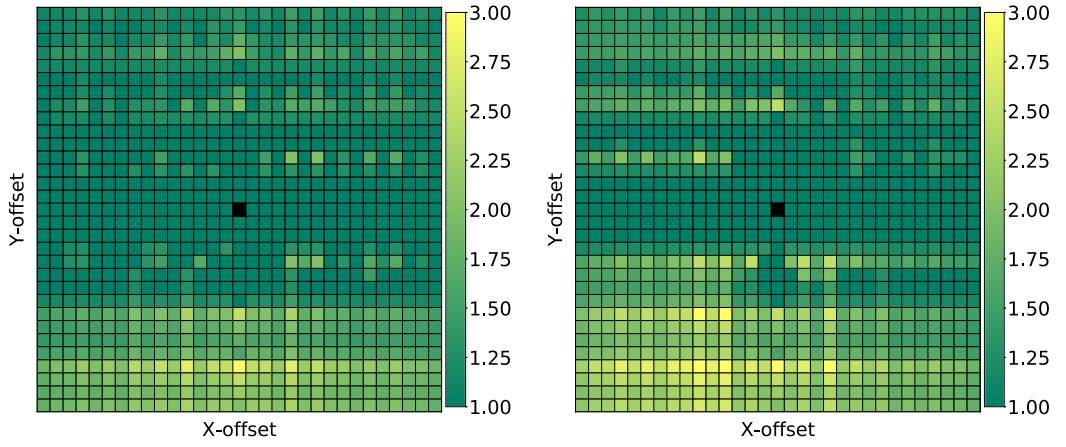
Fig. 12. Adjacency of wire types: avalanche (a) and truncated greedy (b). Entries with no number are prohibited by construction. Rows correspond to drivers and columns to targets.

11.2 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 54th iteration when the pattern also contained 78 switch types. The exact distribution of fanouts and fanins enables a tighter packing of the multiplexers of the avalanche pattern, leading to a slightly lower tile width. Fanouts and fanins still predominantly determine the wire delays, however, which are very close between the two patterns and, on average, slightly lower for the truncated greedy (Table 1).

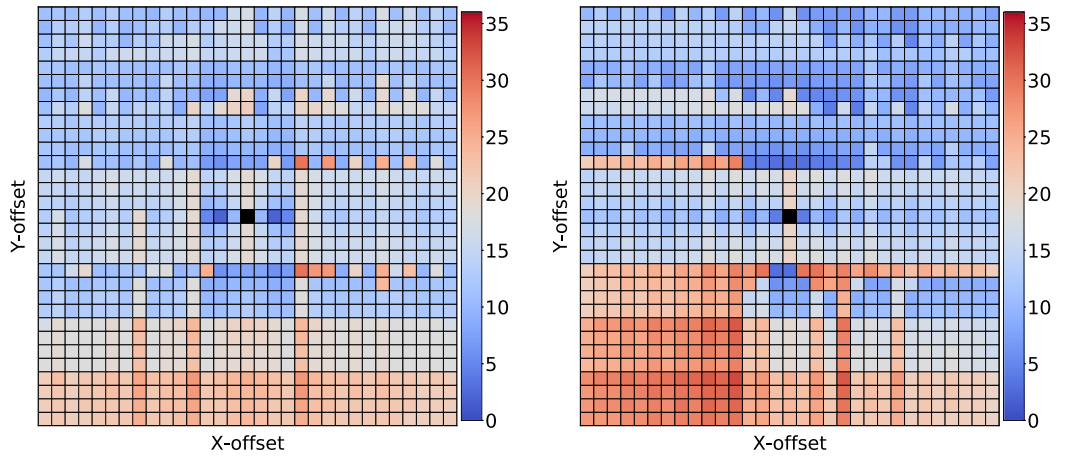
11.2.1 Adjacency. Adjacency between different wire types is illustrated in Figure 12. Avalanche search resulted in more varied connectivity between wire types of different lengths. This can be seen by observing that, e.g., the fanouts of H1Ra and H1Rb are complementary in the avalanche pattern, whereas they have two switch types in common in the truncated greedy. Similarly, fanouts of V1Ua and V1Ub share three switch types in the avalanche pattern, whereas they share eight out of nine switch types in the greedy pattern. This suggests that the greedy search selects multiple switch types between the same lengths of wires, commonly connected by the router, where only a subset of them would suffice. As a result, with the same number of switch types, fewer different wire lengths can be connected.

11.2.2 Grid Distances. Consequences of selecting multiple switch types between the same wire lengths, instead of introducing more variety, can be seen in Figure 13. Each entry of the matrices represents the minimum number of distinct channel 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 switch types 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 that disregards the impact of switch load on wire delay (Figure 14). 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 small FPGAs. In a production setting, larger circuits should be used. We will discuss this further in Section 14.5.2.



(a) Avalanche. (b) Truncated Greedy.

Fig. 13. Hop-distances from the center of the FPGA to other tiles, normalized by the distances computed on a pattern containing all allowed switch types. Dark green is best.



(a) Avalanche. (b) Truncated Greedy.

Fig. 14. Percentage increase of the delay needed to reach other tiles from the center, compared to a hypothetical switch-pattern containing all allowed switch types with no impact on wire delay. Dark blue is best.

11.2.3 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 (Figure 15). 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, which, combined with oversimplified intracluster interconnect [8], may make the delays inside the cluster dominant.

11.2.4 Routability. To see how the two patterns compare under increased stress, we generate 10 synthetic circuits with about 10,000 LUTs using *Gnl* [27]. The Rent's exponent was set to 0.7—the maximum used in the ISPD'16 routability driven placement contest [28]. We take the distribution of different LUT sizes in the circuits from Hutton et al. [29]. Then, we place the circuits on

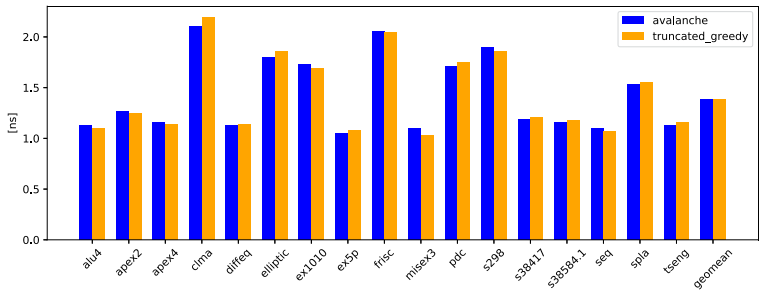


Fig. 15. Routed delays for the avalanche and the truncated greedy pattern.

Table 2. Percentage of Congested rr-graph Nodes after 300 VPR Routing Iterations on Gnl Circuits

circuit	1	2	3	4	5	6	7	8	9	10
avalanche	0.906%	0.274%	0%	0.521%	0.323%	0.187%	0.149%	0.262%	0.040%	0.002%
trunc. greedy	1.061%	0.632%	0.484%	1.340%	0.597%	1.175%	0.982%	0.509%	0.275%	0.984%

Table 3. Number of VPR Iterations Needed to Route Each of the 10 Gnl Circuits

circuit	1	2	3	4	5	6	7	8	9	10
Gnl-extended avalanche	142	61	27	106	26	55	46	55	30	82

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.

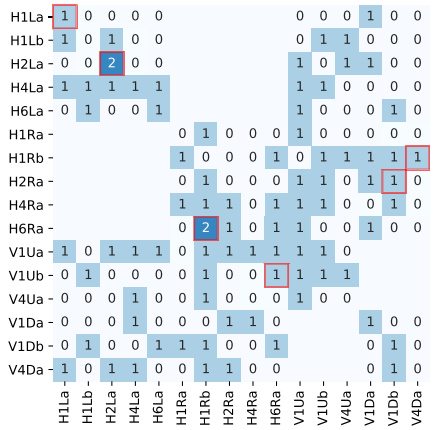
Table 2 shows the percentage of congested rr-graph nodes at the end of the 300 routing iterations for each circuit. The pattern obtained through avalanche search managed to legally route one of the 10 circuits, while no circuit was routable on the greedy pattern. The difference in the percentage of remaining congested nodes also showcases the higher routability of the switch-pattern obtained through avalanche search. Nevertheless, not being able to route 9 out of 10 circuits is not acceptable for any meaningful pattern. We describe a remedy to this in the next section.

12 MULTI-STAGE SEARCH

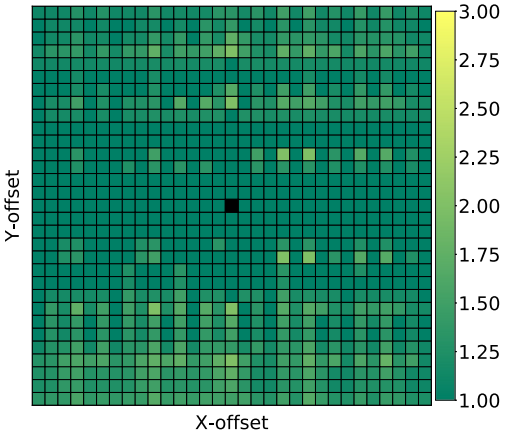
In Section 11.2.4, we have seen that searching for a pattern on a set of benchmark circuits that has a lower connectivity demand than some circuits for which the final architecture is intended can result in those more complex circuits failing to route. However, switch types needed by the simpler circuits are most likely also needed by the more complex ones. Hence, running the search on the smaller circuits first and using the resulting pattern to initialize the search on more complex circuits is a reasonable way to reduce the search runtime. We demonstrate that in this section by presenting the results of continuing the search from the pattern of Figure 12(a) on the Gnl benchmarks described in Section 11.2.4.

12.1 Convergence

In each iteration, 1 of the 10 Gnl circuits was routed to derive usage statistics, and the circuits were changed between iterations in a round-robin fashion. This additional run converged after three search iterations, adding six more switch types to the pattern. On this extended pattern, all 10 Gnl circuits routed successfully in less than 300 router iterations (Table 3). Instead of running 39 search iterations on the more complex circuits, which take about 7× more time to route, it was possible to run the majority of these iterations on the smaller circuits, drastically reducing the runtime. We will discuss runtime in more detail in Section 15. Here, we would just like to note that even



(a) Adjacency. Red switch types were added on top of those of the pattern of Fig. 12(a).



(b) Hop-distances. Addition of access to V4Da greatly improved the distances in the lower half-plane.

Fig. 16. Pattern obtained after continuing the avalanche search on the Gnl benchmarks of Section 11.2.4.

though the 10,000-LUT benchmarks are still rather small and running the entire search on them in a production setting would certainly be feasible, scaling the search to large modern circuits that can take several hours to route even in absence of avalanche costs could be more difficult without adopting this approach of gradually increasing the complexity of the used circuits.

12.2 Pattern Changes

The final pattern obtained after the additional search run on the Gnl benchmarks is shown in Figure 16(a), with switch types added on top of the pattern of Figure 12(a) highlighted in red. It is interesting to note that besides providing access to V4Da that results in significant reduction in hop count needed to reach cells below the center (Figure 16(b)), the new pattern also adds more options to switch between different LUT-heights (entries with a value of 2; see Figure 4). This may suggest that a few such *inter-plane* connections greatly help to reduce congestion, as mentioned by Chromczak et al. [8].

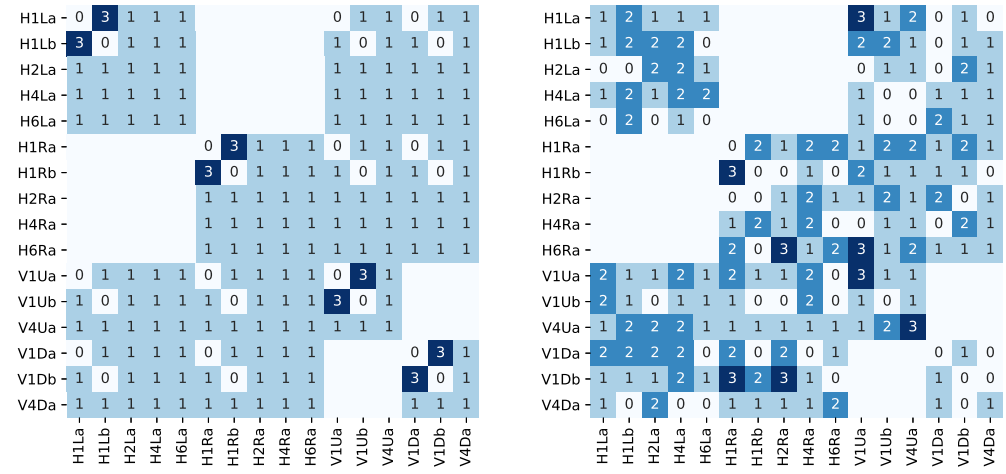
The 8% increase of the pattern size led to a slight increase of the wire delays. This caused the geomean routed critical path delay of the MCNC benchmarks to rise by 0.6% to 1.39 ns.

13 COMPARISON WITH SIMULATED ANNEALING

In Section 14, we analyze in more detail various aspects of the avalanche search algorithm. However, before delving into details, we first compare avalanche search to a method inspired by previous work. Namely, Lin et al. successfully used simulated annealing for simultaneously optimizing channel composition and the switch-pattern [17]. In this section, we investigate how a similar method compares with the proposed avalanche search.

13.1 Initial Pattern

We initialize the search with the default pattern produced by the physical modeling flow [12], which represents our best effort at manually capturing inter-wire-type connectivity of a modern tapless architecture [7], 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 [8]. The initial pattern contains 180 switch types organized as shown in Figure 17(a).



(a) Manual [12] (b) Manual [12] annealed

Fig. 17. Adjacency of wire types: initial manually designed pattern [12] (a) and its annealed version (b).

The optimal hop-distances that it achieves are not sufficient to counter the wire delay increase due to a high load (Table 1). As a result, the geomean routed delay is 5.8% larger than for the avalanche pattern (Table 1).

13.2 Setup

We use two very simple moves generated with equal probability: including or removing one of the 564 considered switch types. The self-normalizing two-term cost function of Marquardt et al. [26] 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:

$$\Delta cost = 0.5\frac{\Delta A(tile)}{\text{prev. } A(tile)} + 0.5\frac{\Delta \text{geom. rtd. crit. path delay}}{\text{prev. geom. rtd. crit. path delay}}. \tag{5}$$

To save runtime, wire delays are measured only when the switch-pattern differs from that of the previously measured architecture in at least five switch types, while floorplan is optimized only on temperature change. The same three MCNC circuits driving the avalanche search of Section 11 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.

13.3 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 (Figure 10). This makes convergence of the optimization difficult, as visible in Figure 18. In the present experiment, 30 new switches were added, while both adjacency regularity (Figure 17(b)) and hop-distance optimality were broken. The increased wire delays (Table 1) 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

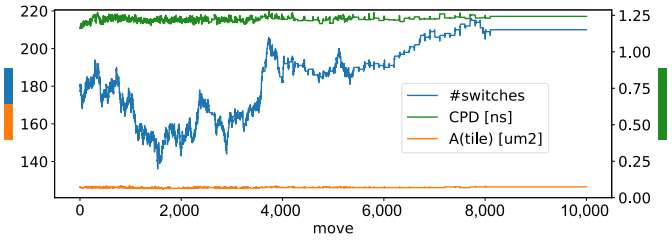


Fig. 18. Convergence of the simulated annealing optimization.

annealing it was just sufficient to rectify that. If applied to one fixed channel composition, then 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 [30], we believe that our avalanche search method—essentially relying on the same principles as PathFinder—removes the need for similarly elaborate heuristics to explore interconnect architectures.

14 ANALYSIS OF SOME FURTHER ASPECTS OF AVALANCHE SEARCH

In this section, we present results of several additional experiments that aim to increase the understanding of how avalanche search functions and what is the impact of different elements of the algorithm. Unless stated otherwise, experimental setup of Section 10 was used in all experiments.

14.1 Parameters

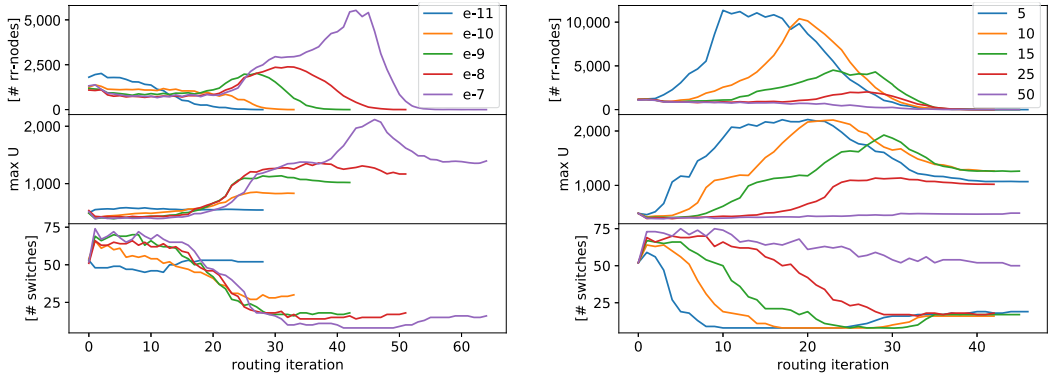
The functional form of the avalanche costs (Equation (2)) involves three parameters: the starting avalanche cost, $s(u)$, and the two parameters dictating the rate of cost decrease with respect to usage, a_p and a_h. For the search method to be effective, these parameters must be assigned reasonable values. In this section, we present results of several experiments intended to help the understanding of the impact of parameter tuning on the effectiveness of the algorithm. We also give some remarks on how to choose good parameter values. Because different switch types are already distinguished by their timing cost, we chose to fix all $s(u)$ to a single parameter s.

14.1.1 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 that do so is very small. This depends on the size and structure of the circuits being routed, making it difficult to choose a single value for a_p and a_h.

To resolve this issue, we first record the maximum usage during the first routing iteration, when the avalanche costs are temporarily reset to zero, much like VPR typically neglects congestion in the first iteration [16]. This allows all nets to initially choose the timing-optimal resources. Let the maximum recorded usage be M_U^1. We compute a_p and a_h as:

$$a_p = a_h = \frac{s}{M_U^1 \times (iter\_to\_zero + 1)}. \tag{6}$$

In other words, 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



(a) Dependence on the starting avalanche cost.

(b) Dependence on the rate of avalanche cost decrease.

Fig. 19. Dependence of concentration on avalanche parameters. The top graph of figure (a) 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. Graphs of figure (b) are analogous to those of figure (a) for the starting avalanche cost fixed at 10^{-9} and *iter_to_zero* $\in \{5, 10, 15, 25, 50\}$.

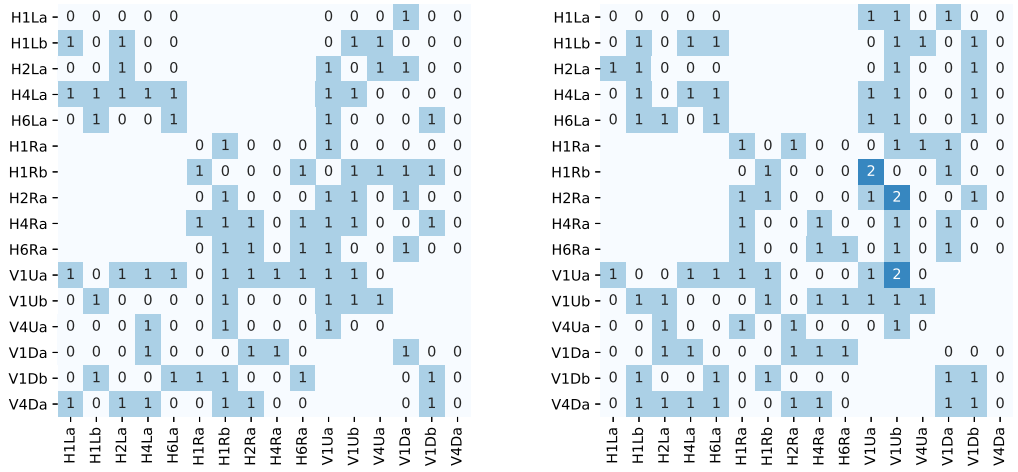
the first iteration of Algorithm 2, a_p and a_h do not change until the end of the search. Benefits of independently setting a_p and a_h are still to be investigated.

14.1.2 Starting Cost. Figure 19(a) shows the effect of various starting costs on concentration and congestion resolving when simultaneously routing the *alu4*, *ex5p*, and *tseng* MCNC circuits [31], 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 that disappears once congestion is penalized sufficiently for nets to move to switch types 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 switch types 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 switch types 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 of types not required by other nets—lead to very similar results in this respect, by the end of the routing process.

While larger values of s, such as 10^{-7}, may lead to additional reduction of the obtained switch-pattern size, in the experiments in this article, we use $s = 10^{-9}$, since it provides a reasonable tradeoff between concentration and runtime.

14.1.3 Rate of Decrease. Figure 19(b) shows the results of sweeping *iter_to_zero* under the setup of Section 14.1.2, but with s fixed at 10^{-9}. Smaller values quickly reduce the cost of switch types that 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



(a) Simultaneous routing (same as Fig. 12(a)).

(b) Independent parallel routing.

Fig. 20. Adjacency of wire types: simultaneous routing (a) and independent parallel routing (b). When routing all circuits at once and using the total usage to evolve the avalanche costs and select the switch types to enter the pattern, large circuits may have an unfair advantage. Routing many circuits at once may also not be feasible due to exceeding runtime. Figure (b) shows a pattern obtained from an avalanche search where all circuits are routed independently in parallel, with independently autotuned avalanche parameters. After each iteration, usages are normalized for each circuit and their average is taken as a basis for switch type selection.

does not allow the higher-usage switch types to attract nets to route through them. It appears that a good tradeoff 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 all experiments presented in this article, we use $iter\_to\_zero = 25$.

More comprehensive analyses could lead to parameter values that produce better quality solutions or reduce runtime. Nevertheless, at the moment, it does not seem that avalanche search is particularly sensitive to the values of parameters.

14.2 Circuit-level Parallelization

In Section 9.2, we proposed to route multiple circuits at once so usage information and avalanche costs can be shared among them. The rationale was that different nets of multiple circuits can together negotiate a more compact pattern than when routed individually. We test that hypothesis in this section.

14.2.1 Average Normalized Usage. We run two experiments. In the first one, each circuit is routed independently, using individually autotuned avalanche parameters. Once all circuits are routed, the final avalanche costs of all switch types are averaged out among all circuits, while the usages are first normalized by the total usage in the particular circuit and then averaged. In this way, large circuits are no longer given an unfair advantage over the small ones in determining switch type selection. The average costs and average normalized usages are then used to determine which switch types are adopted in the pattern, as described in Section 9.

The results of this experiment are shown in Figures 20 and 21. The search relying on independently routing the *alu4*, *ex5p*, and *tseng* circuits converged in 57 iterations (as opposed to 36 when routing all circuits together) and accumulated 87 switch types (as opposed to 78 when routing

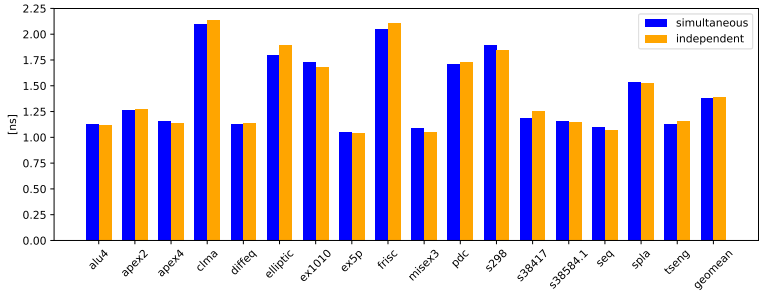


Fig. 21. Influence of simultaneously routing multiple circuits during pattern search on routed delays.

all circuits together). The smaller size of the pattern obtained through routing all circuits at once demonstrates the benefit of sharing the usage and cost information between circuits during routing. However, as Figure 21 shows, the routed critical path delays on the larger pattern are only negligibly (0.25%) larger. This means that when running avalanche search on larger modern circuits where even routing one circuit may be a challenge (see Section 15), let alone multiple of them at the same time, combining normalized usages can be a reasonable alternative. Of course, as an intermediate step, providing some mechanism to (periodically) pass cost information between different parallel threads routing different circuits independently could be useful.

It is interesting to note that even though the potential dominance of the larger circuits is now less likely, that did not improve the geomean routed delay of the three circuits used for the search; it is in fact negligibly higher (by 0.28%).

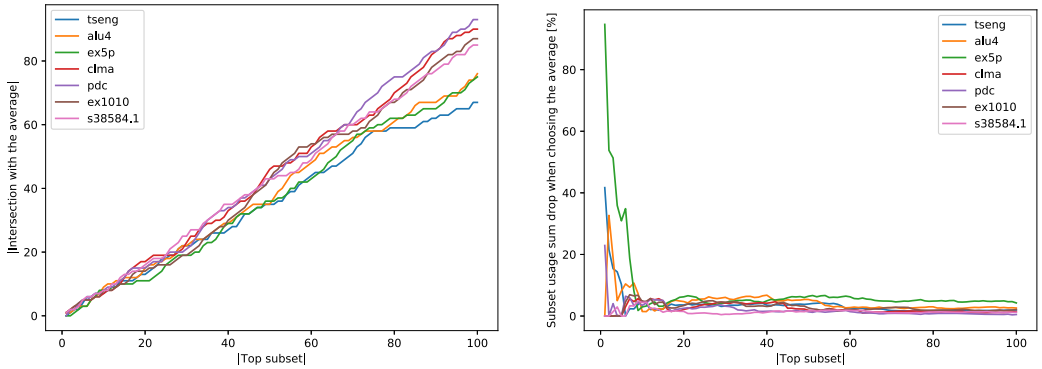
14.2.2 Total Usage. The second experiment attempts to more closely approximate the behavior of simultaneously routing all circuits, while still routing them independently. Namely, the avalanche parameters are taken from the search that routes all circuits together and, after every iteration of the search, the current and historical usages of each switch type are summed among all circuits. This total usage and the costs computed from it are then used for selecting the switch types. Hence, the only difference between this approach and routing all circuits simultaneously is that usage information is not shared between the circuits during the search iterations.

This search converged in 51 iterations, accumulating 92 switch types. For the interest of space, we do not show the resulting pattern, nor plot the routed delays, which are on geomean 2.63% worse than when routing all circuits at once.

14.3 Sensitivity to Circuit Choice

One of the main advantages of benchmark-driven FPGA architecture design is that the obtained architecture can be tailored to some extent to the circuits of interest, represented by the selected benchmark set. However, this is also one of the main disadvantages of the approach, since if the benchmark set does not appropriately represent the intended use of the architecture, then the architecture will either completely fail to implement some circuits of interest or fail to do so with appropriate performance. We have seen an instance of that already in Section 11.2.4. In this section, we attempt to provide a deeper understanding of dependence of switch type usage statistics—which forms the basis for the avalanche search algorithm introduced in this article—on the choice of the circuits used to derive these statistics.

In Section 14.2, we have determined that average normalized usage is an effective metric for choosing the switch types to adopt in the pattern. We will use it here to assess the impact of circuit choice on usage statistics. Namely, the rationale is that the less dependent usage is on exact



(a) Intersection of the maximum-usage subset with the average.

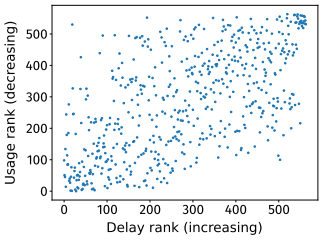
(b) Usage drop when choosing the average.

Fig. 22. Maximum-usage subset overlaps with the average of all circuits. Figure (a) shows the size of the intersection of the maximum-usage subset of each individual circuit and the same-size subset with maximum normalized average usage over all circuits for subset sizes ranging between 1 and 100. Overlap between the two subsets is reasonably high (line close to $y = x$) for all circuits, with some local variation. This variation increases with subset size, since larger subsets capture more switch types with low usage, mostly used by the few critical paths. Figure (b) shows the relative total usage drop if the average subset is chosen instead of the subset determined for each circuit individually. Since the top few switches differ among the circuits, large drops are observed for very small subset sizes. However, for subsets larger than about 10 switch types, cumulative usage drop is consistently below 10% for all circuits.

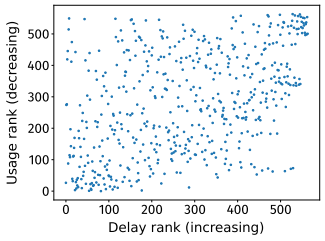
choice of the circuits on which it is observed, the closer the observation made on any individual circuit will be to the average computed on several circuits.

Let S_c be the set of all switch types ordered by the decreasing usage achieved on circuit c. Let S_m be the set of all switch types ordered by the decreasing average normalized usage on a set of circuits C. Furthermore, let S_c^n and S_m^n be the subsets of the aforementioned sets containing the first n of their elements. As a measure of similarity between usage statistics obtained on individual circuits from C and their average, we use the size of the intersection $|S_c^n \cap S_m^n|$ for various subset sizes n. We plot this metric for seven different individually routed MCNC circuits and n ranging between 1 and 100 in Figure 22(a). Usage statistics come from a single avalanche search iteration. For most values of n, the curves for most circuits are close to $y = x$. This means that there is significant similarity in the sets of most used switch types chosen by the router on different circuits. Hence, the sensitivity of the search outcome to the circuits used to run it is not particularly high. However, we can see that as n increases, most curves start moving away from $y = x$. This is because larger subset sizes capture switch types with significantly lower usage, mostly catering to the needs of the critical paths, where similarity between the circuits is lower.

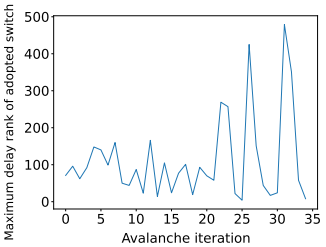
Let $U(S) = \sum_{e \in S} U(e)$ be the total usage of all switch types in a set S. We have seen that a maximum-usage subset of each circuit in Figure 22(a) significantly overlaps with the maximum-average-usage subset. Now, we would like to quantify how large a drop in total usage each circuit $c \in C$ would experience if S_m^n is chosen for it in place of S_c^n. Figure 22(b) plots $\frac{U(S_c^n) - U(S_m^n)}{U(S_c^n)} \times 100\%$, for the same circuits and values of n used in Figure 22(a). The total usage of both subsets is computed solely with usage information of the respective circuit. For very small n, the drop is significant, because the few most used switch types differ between circuits. However, as n increases beyond 10, total usage drop is consistently below 10%. This experiment further suggests that dependence of the usage statistics on the circuits on which they are observed is not particularly high and that the major differences could arise only in the long tail of switch types with low usage.



(a) Correlation between the rank of a switch type in sorting by increasing approximate delay and decreasing usage after the first avalanche iteration.



(b) Correlation between the rank of a switch type in sorting by increasing approximate delay and decreasing usage after the 16th avalanche iteration.



(c) Maximum rank of an adopted switch type in sorting by increasing approximate delay, for each avalanche iteration.

Fig. 23. Correlation between switch type adoption and its approximate delay.

14.4 Influence of Approximate Switch Type Delay on Adoption

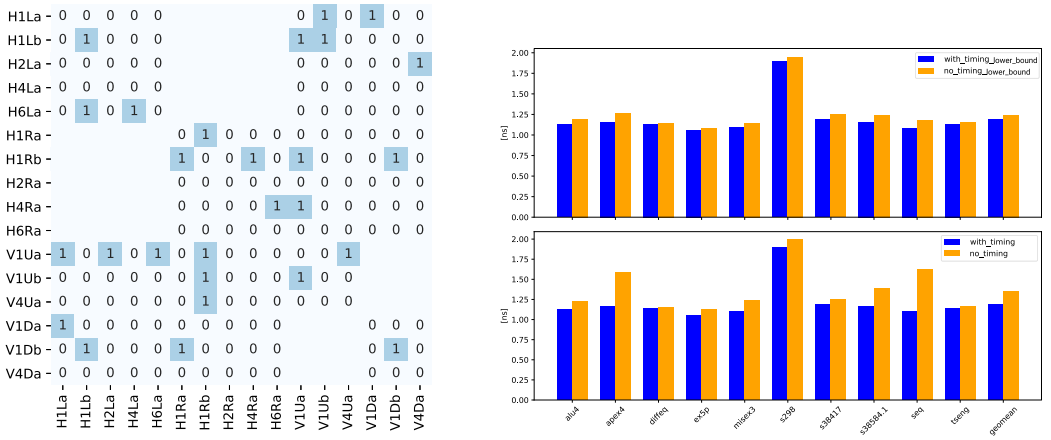
As was described in Section 9.1, different switch types are differentiated in the rr-graph by a delay that strives to approximate the impact of adding the particular switch type to the pattern on loading the wire that is driving it. While this is a way of informing the router on how expensive it would be to add a particular switch type to the pattern, potentially motivating it to use another, less detrimental switch type, we must confirm that the choices made by the router are not predominantly based on this delay penalty. If that were the case, then avalanche costs would become irrelevant, as the pattern would largely be determined by the *a priori* assigned delays.

In Figure 23(a), we plot the correlation between switch type usage and its intrinsic delay reported to VPR in the first iteration of the avalanche search. The x-coordinate of each point representing a switch type is determined by its delay rank, with switch types with lower delay being closer to the origin. The y-coordinate of each point representing a switch type is determined by its usage rank, with higher usage being closer to the origin. Ties are broken by names.

If switch type adoption was solely dependent on switch type delay, then all points would be along the $y = x$ line. We can clearly see that this is not the case and that the router is capable of escaping any potential local minima set by the preassigned delay penalties. However, there is some correlation between delay and usage, since points are scattered more closely to the $y = x$ line than to $y = 564 - x$; this was the intention of conveying the physical information through the approximate delays. The correlation weakens as the iterations of the avalanche search progress, which can be seen by the larger spread away from $y = x$ in Figure 23(b), depicting the same situation at iteration 16. Perhaps more illustrative of this phenomenon is Figure 23(c), depicting the maximum delay rank (i.e., the slowest) of the switch types adopted in each iteration of the avalanche search. We can see that some of the slowest switch types are adopted towards the end of the search. This could be because some critical path needed the particular switch types to form a connection between wires that optimally implement it, or the particular switch type was needed to resolve some outstanding congestion hotspot.

14.5 Routability-driven Search

In Section 8.5, we commented on the importance of allowing the router to route critical paths with switch types that otherwise have low usage and would not be adopted in the final pattern. We updated the node cost function in PathFinder to enable better selectivity of the critical paths and to allow them to see the high-cost switch types with low usage as sufficiently cheap. In this section, we inspect whether this was successful and whether the switch types that entered the pattern of Figure 12(a) were not merely required for making it possible to route the circuits used in the search.



(a) Adjacency. (b) Routed delays.

Fig. 24. Results of an avalanche search ignoring timing information.

To check this, we reran the search neglecting the timing information (insofar as it is not embedded in the base costs of the wires that are dependent on the delay [16]) by setting the *max_crit* parameter of VPR to 0. This search converged in 23 iterations, which is significantly less than the 36 that were needed when timing was considered. More importantly, only 27 switch types entered the pattern, as opposed to 78 when timing was considered. This significantly smaller pattern is shown in Figure 24(a). As can be expected, most of the connectivity incident to the long wires has disappeared, because the benchmarks used in the search are small enough that their congestion can largely be resolved while using only the short wires. Under these circumstances, long wires serve only to enhance the speed of the implemented circuits and can thus be disregarded if timing is not of concern. For instance, H6Ra had a fanout of six in the pattern obtained with timing considered (Figure 12(a)), whereas in this smaller pattern it has no fanout at all.

It is interesting to note that between the eight wires of length one, there are 17 switch types, making their average fanout (F_s [6]) very close to 2, which Lemieux and Lewis have determined to be the minimum required when two different switch-patterns are used in the FPGA grid in a checkered fashion [32]. Additional switch types may have been chosen to compensate for the lack of two different patterns or simply because the pattern was not fully minimized.

14.5.1 Impact on Performance. Routing results obtained on the smaller pattern are shown side-by-side with the results of the pattern of Figure 12(a) at the bottom of Figure 24(b). The smaller pattern does not provide enough connectivity to successfully route all MCNC circuits. For the subset that can be routed, the geomean critical path delay is 13.86% higher than on the pattern of Figure 12(a), despite the fact that the lower capacitive load and smaller tile area resulted in wires being faster. This clearly demonstrates the utility of considering the critical paths when performing the search.

There could be two main reasons why the delays are so significantly higher on the smaller pattern: (1) pairs of wires needed to optimally implement critical connections of the circuits cannot be connected due to the lack of the appropriate switch and (2) due to lower routability, nets need to detour more for the congestion to be resolved. To determine which issue contributes more to delay deterioration, we also report the lower-bound critical path delays obtained after the first iteration of VPR at the top of Figure 24(b). Significant deterioration is present there as well, but it is less

pronounced: The geomean is 4.84% higher on the smaller pattern. Hence, it is the poor routability that is the main culprit.

The question of how much freedom should be given to the critical paths to enlarge a minimal routable pattern remains to be answered in future work. In this section, we demonstrated that at least some level of freedom is highly useful, but a more optimal point may lie somewhere in between. One possible solution would be to first find a minimal pattern that supports all of the routability requirements, using a routability-driven search and then extend it in a subsequent timing-driven search, until a predefined budget of additional switch types or exploration iterations is surpassed.

14.5.2 What Do the Results Tell Us? Compaction of the switch pattern presented in this section, which resulted in some of the longer wires being unable to connect to other channel wires, once more points to the fact that the circuits used in the exploration are not large and complex enough to saturate the channel capacity of a modern plane-based FPGA. As anticipated in Section 1.3, this makes it hard to derive general rules of the sort "an FPGA implemented in a 4 nm technology should have between 24 and 32 switches per 16 wires in a plane" from the currently available results. Nevertheless, the compaction also demonstrates the effectiveness of the proposed exploration method in minimizing the pattern size where an opportunity for that exists. This was indeed the main intention of the article—to develop a method for automatically designing switch-patterns that are appropriate for the conditions created by the underlying technology and the requirements of the target circuits, even in situations when general design rules with which a human designer is familiar no longer hold. As we have seen in Section 13, simulated annealing does not possess this feature—at least not in the adopted implementation. Similarly, the stark difference between the performance of the patterns obtained in routability- and timing-driven search demonstrates the capacity of the method to select switch types important for delay optimization. Oracles that can quickly assess routability of a given pattern but not its performance (Section 4.3) would not be useful in this context, other than for pruning away some solutions. In the next section, we discuss in detail the reasons why the results so far presented were limited to small circuits and suggest possible remedies that could help in alleviating this limitation in the future.

15 RUNTIME SCALABILITY

Implicitly representing the entire search space in the rr-graph during avalanche search removes the need to route thousands of explicitly constructed solutions. This means that increased runtime of each routing run can be tolerated. Nevertheless, it is important to assess how large this increase is and where it comes from, so it can be mitigated when necessary. The total routing time spent in the single 37-iteration exploration run of Section 11 was about 5 hours. Of that, about 4.5 were spent by the actual PathFinder extended with avalanche costs, while the remaining half an hour was taken by lookahead computation and allocating the data structures in VPR. When combined with SPICE simulations, rr-graph generation, packing, placement, and placement manipulation, the entire search took about 10 hours. While this may not seem like an exceedingly long time, it is important to give it a context: routing the same three circuits used in exploration (which have a combined size of about 2,700 LUTs), but on an architecture containing only the final pattern, with no switch-splitting nodes and no usage tracking, took a mere 5.5 seconds. This $\sim 80\times$ average runtime increase per iteration is very significant. For instance, each iteration of avalanche search gives a possibility to evaluate 80 different patterns in the black-box-in-the-loop approach, although, as we have seen, even 10,000 moves were not sufficient for simulated-annealing-based exploration of Section 13 to converge to results comparable to those obtained after 37 iterations of the avalanche search (giving a budget of $\sim$3,000 move evaluations).

What is more important, however, is that with such a large increase in routing runtime (likely to deepen further with growing circuit size), it is infeasible to use in exploration circuits that normally take even minutes and let alone hours or days to route. Hence, in this section, we give a detailed analysis of the origins of this runtime increase and suggest several remedies for which we believe that they could be successful at alleviating the problem.

15.1 Routing Graph Size

Intuitively, increasing connectivity in the rr-graph could be expected to reduce the routing time, as a more flexible rr-graph makes it easier to eliminate congestion. This is indeed true, but only up to a certain point: If the rr-graph already offers sufficient flexibility to eliminate congestion, then any further increase in its size will only lead to deterioration in runtime, as it will take longer to find a shortest path for each connection. This was observed by Moctar et al., who determined that routing circuits on an architecture with a fully populated cluster input crossbar represented within the rr-graph takes about 2.5× more time than routing the same circuits on an architecture where this input crossbar is 40% populated (Figure 4 in Moctar et al. [33]). Complexity of Dijkstra's shortest path algorithm is $\Theta(|E| + |V| \log |V|)$ [34]. Assuming that the size of these rr-graphs was dominated by the number of edges describing the intracluster interconnect, and assuming that the population of both architectures was sufficient to easily eliminate congestion, we could expect that runtime is roughly linear in $|E|$. Indeed, the ratio between the edge counts of the fully populated and the 40%-populated crossbars is $1/0.4 = 2.5$, which corresponds to the runtime ratio observed in the work of Moctar et al.

Since avalanche search relies on embedding the entire search space in the rr-graph (Section 5), it is inevitable that the rr-graph used in exploration is much larger than the final one. For example, the final rr-graph obtained from the exploration of Section 12 contained in each switch-block 16 nodes (representing the 16 channel wires originating from it) and 84 edges. The graph representing the entire design space, however, contained 16 + 564 nodes and $2 \times 564 = 1{,}128$ edges—a 36× increase in $|V|$ and a 13× increase in $|E|$. Using the same rough assessment based on the complexity of Dijkstra's shortest path algorithm as above, a runtime increase between 13× and 44× can be expected. To measure this impact, we performed an experiment where we recorded the time taken to complete the first iteration of the inner loop of PathFinder (line 11 of Algorithm 1), when congestion costs are neglected, in three different cases: (1) on the final pattern, (2) on an rr-graph containing all possible switches, but without nodes to split them, and (3) on an rr-graph with all switches split by additional nodes. The runtimes were, respectively, 1.8 s, 9.7 s, and 21.9 s, leading to a runtime increase of 12.2× and 5.4× when additional switches are and are not split by nodes, respectively. This suggests that the increased edge count is the dominant problem and that providing support for weighting both nodes and edges, to avoid the need to split switches, could lead to a $\sim 2\times$ runtime improvement.

15.1.1 Possible Remedy: Suppressing Low-usage Switches. One way to reduce the rr-graph size is to suppress low-usage switch types after the usage is first measured in the first iteration of PathFinder before the avalanche costs are initialized. For example, if it is known that the size of all multiplexers should be some fixed number m, as is often the case in commercial architectures [7], then selecting the intrinsically most-used $k \times m$ switch types for each multiplexer, where k is some small constant, can lead to a drastic reduction in rr-graph size. The remaining switch types can periodically be brought back to ensure that some of them did not become more preferable, due to prior adoption decisions.

15.1.2 Possible Remedy: Randomized Instance Sparsification. Another orthogonal approach could be to remove switches of a certain type from some switch-blocks but retain them in

others. Provided that a sufficient number of instances is available for each type, it is plausible that the collected statistics would show little change compared to the situation when each switch type is represented in every switch-block. Removing switches from one region would likely alter the paths in another, but the hope is that if sparsification is done in such a way as to ensure that each multiplexer receives a sufficient number of inputs, average behavior would be similar to representing all switches everywhere.

15.1.3 Possible Remedy: Partition, Sample, and Mix. The method of simultaneously routing multiple circuits depicted in Figure 11 may seem completely impractical following the analysis of this section. However, it also offers a possible remedy for the problem of rr-graph size increase. Rather than combining multiple complete circuits together on a common FPGA, placements of large circuits can be partitioned into manageable pieces, a certain subset of these pieces can be selected through random sampling, possibly across multiple circuits, and then combined on a common FPGA. In this way, switch types necessary for short-haul connections can be quickly determined, before proceeding recursively, as in Section 12, to extend the pattern with others required by the long-haul connections. Alternatively, long-haul connections passing over a particular piece could be determined through the use of a global router and approximated in the piece by appropriate input-to-output connections.

Besides rr-graph size increase, there are several other reasons why routing in the presence of avalanche costs is significantly slower than the usual routing. We list them in the following sections.

15.2 A*

FPGA routers typically use A^* to speed up shortest path finding [16]. The idea is that when a node u is being pushed to the heap, it could be easy to obtain a lower bound on the cost of the path needed to reach the target t from u. Then, instead of pushing u with the known cost f needed to reach u from the source s, it is pushed with the cost $f + g$, where g is the estimate for reaching the target. If g is really a lower bound, then the algorithm will never fail to miss the shortest path. At the same time, though, the addition of g will prevent some nodes from ever being popped from the heap, thus drastically reducing the portion of the rr-graph that needs to be explored. The closer the chosen lower bound to the actual cost of the remaining path is, the more effective this pruning will be.

15.2.1 Congestion Lookahead. In case of congestion costs, the minimum value of congestion is 0. Hence, a lower bound estimate (lookahead) can only be computed when congestion itself is completely neglected and only base costs of routing resources are taken into account [16]. This means that pruning will be more effective towards the beginning of the routing, because there the actual congestion costs are closer to 0. Fortunately, the number of congested nodes generally drops as the routing iterations progress (Figure 26), so only a relatively small portion of the rr-graph and the circuit's nets will be affected by the poor performance of the lookahead.

15.2.2 Avalanche Lookahead. Similarly to congestion costs, the lowest value that avalanche costs can attain is 0. Hence, when computing a lookahead that stores the values of g in a fixed look-up table, as is typically done by VTR-8 [16], we have to set all avalanche costs to 0; otherwise, it would not be a true lower bound (*admissible*). However, contrary to congestion costs that are (close to) 0 in the beginning of the routing process, avalanche costs drop to zero towards the end of it. For most iterations, such a lookahead is ineffective in presence of avalanche nodes.

Let us illustrate this by the example of Figure 25, showing a portion of the rr-graph with the source and the sink nodes designated as s and t, respectively. There are two possible paths between

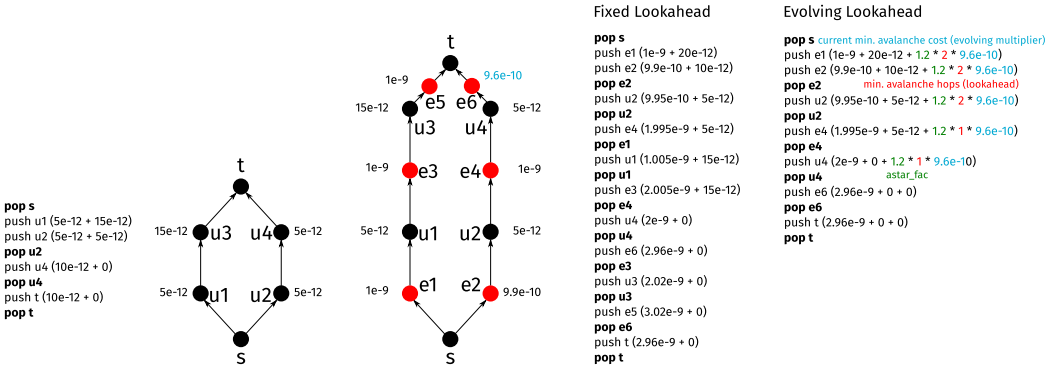


Fig. 25. Example of lookahead ineffectiveness. Left figure shows a portion of the rr-graph without avalanche nodes and a sequence of heap operations needed to find the shortest path from s to t, while the right one shows the same portion of the graph with avalanche nodes (red), along with the corresponding heap operation sequence. Costs are annotated next to nodes. Ratios between the avalanche costs and the base costs of the wires are realistic. Lookahead values are on the right side of the "+" signs. "Fixed Lookahead" is the usual lookahead used by VTR [16], while "Evolving Lookahead" is the modification proposed in Section 15.2.3.

them, each composed of two wires. The path on the left, composed of the nodes $u1$ and $u3$, has a total cost of $20e-12$, when there is no congestion present, while the path on the right, composed of $u2$ and $u4$, has a total cost of $10e-12$. In absence of avalanche nodes and congestion (left figure), the lookahead is exact and only the nodes of the shorter path get popped. Once the avalanche nodes are inserted (right figure), their cost overshadows the admissible lookahead and nodes from both paths need to be popped. The increase in the number of nodes along the shortest path necessarily increases the number of pops needed to find it. However, this alone would cause an increase from 4 to 7 pops (1.75×), while the actual number of pops required to find the shortest path in the right figure is 11 (2.75× more). The difference comes from lookahead ineffectiveness. This is only a toy example to illustrate the mechanism of pruning. In practice, when nodes have higher out-degrees, the difference between having an effective pruning function g and not is much higher. For example, Swartz et al. have demonstrated that A^* can produce a speedup of about 50× on MCNC circuits [15]. In larger circuits with longer average paths, the impact could be even higher. For example, as we have mentioned in the previous section, the first iteration of PathFinder when routing one of the Gnl circuits with all avalanche nodes in place, but their cost reset to zero (making the admissible lookahead effective), took 21.9 s. Raising the avalanche costs up to $1e-9$ increased this time to 3,899 s—an almost 180× increase. After the first iteration of avalanche search, when 12 switch types have been adopted to the pattern, the same runtime reduced to 319 s, bringing the lookahead ineffectiveness gap down to $\sim$ 15×. Once the cost of the adopted switch types drops to zero, lookahead becomes effective for paths routed through them. Hence, it is imperative to improve lookahead effectiveness in the initial iterations with few adopted switch types.

15.2.3 Possible Remedy: Evolving Lookaheads. One idea that could help in resolving this issue is to store in an additional look-up table the minimum number of avalanche nodes needed to connect a node of given type to the target tile. Then, when pushing nodes onto the heap, their cost could be increased by the appropriate entry from this table, multiplied by the cost of the currently cheapest avalanche node. This is illustrated on the right of Figure 25, for the *astar_fac* parameter multiplying g set to 1.2 (VTR-8 default [16]). The proposed lookahead now finds the shortest path with seven pops, which is the minimum in presence of switch-splitting nodes. Note that this approach is not easily applicable to congestion lookahead, because with very high probability, some node

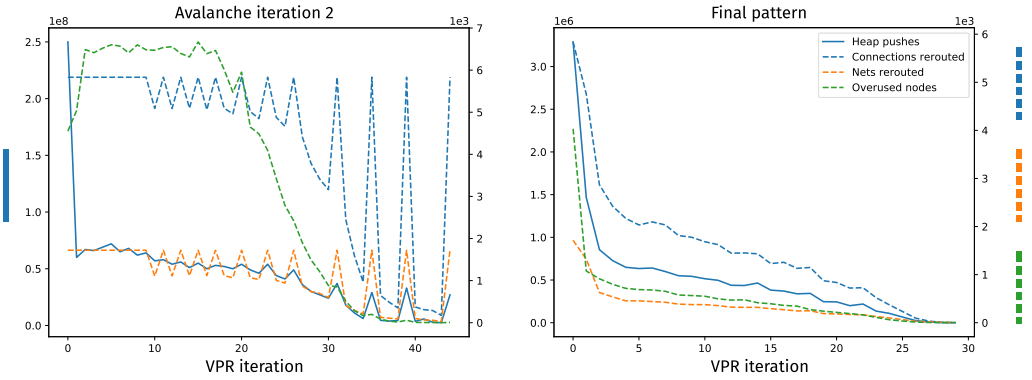


Fig. 26. Heap operations and rip-ups during the second iteration of the avalanche search (most congested). Bars on the sides of the graphs designate which y-axis corresponds to which curve(s) (right for the dashed, left for the solid).

will always remain unused, thus reducing the lowest congestion cost to zero. Avalanche nodes, however, have the highest cost when their type is unused.

Another potential advantage of avalanche costs is that they are tied to types and not instances, unlike congestion costs. This could perhaps allow runtime improvements by increasing preprocessing effort, as information would need to be stored only about relationship between the few types, rather than the numerous instances. At any rate, for the proposed method to be truly scalable, A^* must be made effective.

15.3 Periodically Forcing Rip-up

Figure 26 plots rip-up and congestion statistics of routing in two extreme iterations of the avalanche search: (1) second iteration, where the highest congestion was observed (left figure), and (2) last iteration, where the switch-pattern has been finalized and no potential switches and in turn no avalanche costs are present. Dashed orange and blue lines represent the number of nets and connections, respectively, that were ripped up and rerouted in the corresponding router iteration. We can see that in the right figure, where avalanche costs are not present, these curves almost monotonically decrease. This comes from the incremental-rerouting capabilities of VTR 8 [16], where only connections that use congested nodes or fail to meet timing are ripped up, contrary to the original VPR PathFinder implementation, which ripped up all connections in every routing iteration [6]. The corresponding curves in the figure on the left contain a number of peaks at an increasing distance from one another, as the routing iterations progress. These peaks represent iterations where rip-up of all connections has been forced. If this had not been done, then it would have been possible for some switch types to be used in the initial iterations where the avalanche cost differences were still small, without the paths using them later moving to a cheaper switch type. This would then potentially cause some nonessential switch types to be included in the final pattern. As the routing progresses, avalanche cost changes become smaller, so it becomes less useful to rip up the legal connections and hence this forcing is done less frequently.

15.3.1 Possible Remedy: Path-cost Bounding. Peaks in the number of heap pushes coincide with the forced rip-up peaks, which can be expected. However, this increase in the amount of work could potentially be reduced. Namely, if a legal connection is being routed, then its pre-rip-up avalanche cost is also an upper bound on the avalanche cost of the new shortest path that can implement it, because avalanche costs are monotonically nonincreasing. This upper bound could be used for more efficient pruning when searching for the new shortest path.

15.4 Congested Nodes

The green dashed curve of Figure 26 shows the number of congested nodes in each PathFinder iteration. In the figure on the right, we see that this curve is again almost monotonically decreasing, with the peak being in the first iteration, which neglects congestion altogether. In the figure on the left, however, the situation is drastically different. Since in the first PathFinder iteration avalanche costs are also neglected, congestion is comparable to that of the first iteration in the right figure. As soon as the avalanche costs start to be considered, they create concentration on switch types, which drives the congestion up. This congestion starts to get resolved only after the avalanche costs drop sufficiently for a large-enough number of switch types and at the same time, congestion costs increase enough to outweigh the avalanche costs. This occurs roughly around the middle of the routing run, which coincides with the intended avalanche cost reduction to zero for the most used nodes (see Section 14.1.1). As discussed in Section 8.2.2, fully resolving congestion may not always even be necessary and early stopping could benefit the runtime.

16 CONCLUSIONS AND FUTURE WORK

In this article, we introduced a new method for automated exploration of FPGA switch patterns, which removes the fundamental limitation of prior techniques: necessity to explicitly list and test numerous architectures in a place and route flow. The proposed method achieves this by leveraging the router itself to perform the exploration, instead of perceiving it merely as a black box used for evaluation of explicitly listed solutions. We hope that this will open up new interesting possibilities in the FPGA architecture domain. One of the most exciting aspects of the method is that it represents the design space implicitly in the routing-resource graph and should thus in principle be useful for exploring any aspect of programmable interconnect that can be represented within an rr-graph. For example, it should be possible to use the method to simultaneously explore intercluster and intracluster interconnect.

16.1 What Remains to Be Done?

In the current implementation, a large increase of the rr-graph size, coupled with ineffectiveness of A^* caused by the additional costs used for pattern minimization, quickly makes runtime on all but very small circuits impractical. However, we strongly believe that these issues can be resolved, and we suggested several possible remedies in Section 15. To make the method truly practical, it is imperative that these and/or other remedies are applied in future work.

16.2 Increasing Regularity

Of similar interest is the ability to design regular patterns that may be required in commercial architectures; for example, it may be necessary that all multiplexers have the same size [7]. It turns out that this problem can be reduced to a completely orthogonal problem of transforming the observed statistics into a pattern that satisfies the specified constraints. We refer the reader to another of our papers, which introduces one solution to this problem [35].

16.3 Reducing Regularity

Finally, in certain cases it may be required to design patterns that are less regular than specified here. For example, in an architecture implemented in an older technology, a plane division may not be enforced and it may be possible that multiplexers close to different LUTs in the same CLB have different input patterns. If this is the case, then simply more wire and switch types are to be introduced, according to the monolithic indexing scheme of Figure 4. Of course, increasing the number of switch types in such a way would increase the rr-graph size as well. We note here once

more that the proposed method may not be as useful for older technologies, as we believe it is for scaled ones, because in them, resistance of lower metal layers is not nearly as pronounced. Hence, individual switches have less impact on delay and well-tested patterns developed for earlier FPGA generations [10] are not likely to be outperformed.

In case regularity is to be reduced in the fashion of checkered architectures proposed by Lemieux and Lewis [32], this can be easily achieved by assigning the switch-blocks in the black tiles one set of switch types and those in the white ones a different set. This approach could be particularly useful for exploring architectures where switch-blocks in hard-IP tiles can differ from those in CLB tiles.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers whose constructive feedback greatly improved the manuscript.

REFERENCES

[1] J. Rose and S. Brown. 1991. Flexibility of interconnection structures for field-programmable gate arrays. *IEEE J. Solid-state Circ.* 26, 3 (1991), 277–82.

[2] Steven J. E. Wilton. 1997. *Architectures and Algorithms for Field-programmable Gate Arrays with Embedded Memory.* Ph.D. Dissertation. University of Toronto.

[3] Yao-Wen Chang, D. F. Wong, and C. K. Wong. 1996. Universal switch modules for FPGA design. *ACM Trans. Des. Autom. Electron. Syst.* 1, 1 (Jan. 1996), 80–101.

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

[5] Herman Schmit and Vikas Chandra. 2002. FPGA switch block layout and evaluation. In *Proceedings of the ACM/SIGDA 10th International Symposium on Field-Programmable Gate Arrays.* 11–18.

[6] Vaughn Betz, Jonathan Rose, and Alexander Marquardt. 1999. *Architecture and CAD for Deep-Submicron FPGAs.* Kluwer Academic Publishers.

[7] Morten B. Petersen, Stefan Nikolić, and Mirjana Stojilović. 2021. NetCracker: A peek into the routing architecture of Xilinx 7-Series FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays.* 11–22.

[8] Jeffrey Chromczak, Mark Wheeler, Charles Chiasson, Dana How, Martin Langhammer, Tim Vanderhoek, Grace Zgheib, and Ilya Ganusov. 2020. Architectural enhancements in Intel®Agilex™FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* 140–49.

[9] X. Tang, E. Giacomin, A. Alacchi, and P. Gaillardon. 2019. A study on switch block patterns for tileable FPGA routing architectures. In *Proceedings of the International Conference on Field-Programmable Technology (ICFPT).* 247–50.

[10] Oleg Petelin and Vaughn Betz. 2016. 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.* 1–10.

[11] Stefan Nikolić and Paolo Ienne. 2021. Turning PathFinder upside-down: Exploring FPGA switch-blocks by negotiating switch presence. In *Proceedings of the 31st International Conference on Field-Programmable Logic and Applications.* 225–33.

[12] Stefan Nikolić, Francky Catthoor, Zsolt Tőkei, and Paolo Ienne. 2021. Global is the new local: FPGA architecture at 5nm and beyond. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* 34–44.

[13] Larry McMurchie and Carl Ebeling. 1995. PathFinder: A negotiation-based performance-driven router for FPGAs. In *Proceedings of the ACM 3rd International Symposium on Field-Programmable Gate Arrays.* 111–17.

[14] Carl Ebeling, Larry McMurchie, Scott A. Hauck, and Steven Burns. 1995. Placement and routing tools for the triptych FPGA. *IEEE Trans. Very Large Scale Integr. Syst.* 3, 4 (Dec. 1995), 473–82.

[15] Jordan S. Swartz, Vaughn Betz, and Jonathan Rose. 1998. A fast routability-driven router for FPGAs. In *Proceedings of the ACM/SIGDA 6th International Symposium on Field Programmable Gate Arrays.* 140–49.

[16] Kevin E. Murray, Oleg Petelin, Sheng Zhong, Jia Min Wang, Mohamed Eldafrawy, Jean-Philippe Legault, Eugene Sha, Aaron G. Graham, Jean Wu, Matthew J. P. Walker, Hanqing Zeng, Panagiotis Patros, Jason Luu, Kenneth B. Kent, and Vaughn Betz. 2020. VTR 8: High-performance CAD and customizible FPGA architecture modelling. *ACM Trans. Reconfig. Technol. Syst.* 13, 2 (May 2020), 9:1–9:60.

[17] M. Lin, J. Wawrzynek, and A. E. Gamal. 2010. Exploring FPGA routing architecture stochastically. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 29, 10 (Sept. 2010), 1509–22.

[18] David Lewis, David Cashman, Mark Chan, Jeffery Chromczak, Gary Lai, Andy Lee, Tim Vanderhoek, and Haiming Yu. 2013. Architectural enhancements in Stratix V™. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 147–56.
[19] Vaughn Betz and Jonathan Rose. 1997. VPR: A new packing, placement and routing tool for FPGA research. In *Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications*. 213–22.
[20] Elias Ahmed and Jonathan Rose. 2000. The effect of LUT and cluster size on deep-submicron FPGA performance and density. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. ACM, 3–12.
[21] Vaughn Betz and Jonathan Rose. 1998. How much logic should go in an FPGA logic block? *IEEE Des. Test Comput.* 15, 1 (1998), 10–15.
[22] Kaichuang Shi, Xuegong Zhou, Hao Zhou, and Lingli Wang. 2022. An optimized GIB routing architecture with bent wires for FPGA. *ACM Trans. Reconfig. Technol. Syst.* 16, 1 (Dec. 2022).
[23] Oleg Petelin and Vaughn Betz. 2015. Wotan: A tool for rapid evaluation of FPGA architecture routability without benchmarks. In *Proceedings of the 25th International Conference on Field Programmable Logic and Applications*. 1–4.
[24] Guy Lemieux and David Lewis. 2002. Analytical framework for switch block design. In *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications*. 122–31.
[25] G. Wang, S. Sivaswamy, C. Ababei, K. Bazargan, R. Kastner, and E. Bozorgzadeh. 2006. Statistical analysis and design of HARP FPGAs. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 25, 10 (2006), 2088–102.
[26] Alexander Marquardt, Vaughn Betz, and Jonathan Rose. 2000. Timing-driven placement for FPGAs. In *Proceedings of the ACM/SIGDA 8th International Symposium on Field Programmable Gate Arrays*. 203–13.
[27] Dirk Stroobandt, Jo Depreitere, and Jan Van Campenhout. 1999. Generating new benchmark designs using a multi-terminal net model. *Integration* 27, 2 (1999), 113–29.
[28] Stephen Yang, Aman Gayasen, Chandra Mulpuri, Sainath Reddy, and Rajat Aggarwal. 2016. Routability-driven FPGA placement contest. In *Proceedings of the International Symposium on Physical Design*. 139–43.
[29] Michael D. Hutton, Jay Schleicher, David M. Lewis, Bruce Pedersen, Richard Yuan, Sinan Kaptanoglu, Gregg Baeckler, Boris Ratchev, Ketan Padalia, Mark Bourgeault, Andy Lee, Henry Kim, and Rahul Saini. 2004. Improving FPGA performance and area using an adaptive logic module. In *Proceedings of the 14th International Conference on Field Programmable Logic and Application*. Springer, 135–44.
[30] S. Brown and G. Lemieux. 1993. A detailed router for allocating wire segments in FPGAs. In *Proceedings of the ACM/SIGDA Physical Design Workshop*. 215–26.
[31] Saeyang Yang. 1991. *Logic Synthesis and Optimization Benchmarks User Guide, Version 3.0.* Technical Report. Microelectronics Center of North Carolina.
[32] Guy Lemieux and David Lewis. 2004. *Design of Interconnection Networks for Programmable Logic*. Kluwer Academic Publishers, USA.
[33] Yehdhih Ould Mohammed Moctar, Guy Lemieux, and Philip Brisk. 2012. Routing algorithms for FPGAs with sparse intra-cluster routing crossbars. In *Proceedings of the 22nd International Conference on Field Programmable Logic and Applications (FPL'12)*. 91–98.
[34] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.
[35] Stefan Nikolić and Paolo Ienne. 2023. Regularity matters: Designing practical FPGA switch-blocks. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 99–109.

Received 10 June 2022; revised 11 February 2023; accepted 4 May 2023