Straight to the Point: Intra- and Intercluster LUT Connections to Mitigate the Delay of Programmable Routing

Stefan Nikolić
École Polytechnique Fédérale de Lausanne (EPFL)
Lausanne, Switzerland
stefan.nikolic@epfl.ch

Grace Zgheib
Intel Corporation
San Jose, USA
grace.zgheib@intel.com

Paolo Ienne
École Polytechnique Fédérale de Lausanne (EPFL)
Lausanne, Switzerland
paolo.ienne@epfl.ch

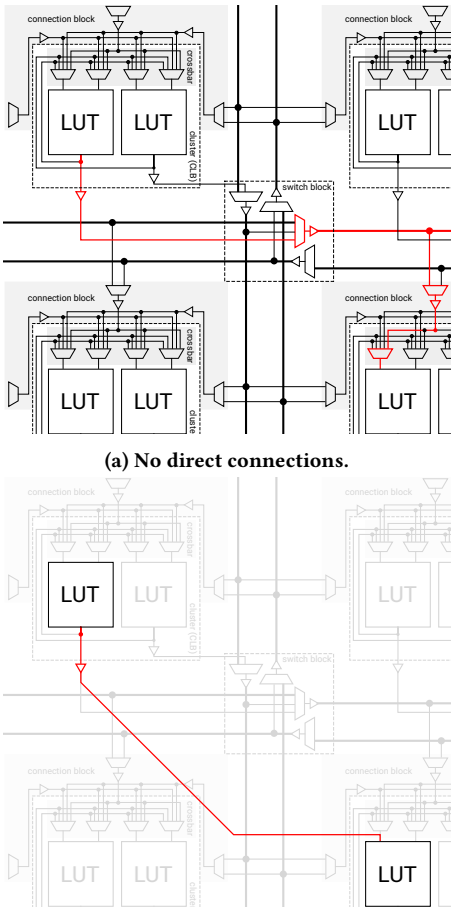
ABSTRACT

Technology scaling makes metal delay ever more problematic, but routing between *Look-Up Tables* (LUTs) still passes through a series of transistors. It seems wise to avoid the corresponding delay whenever possible. Direct connections between LUTs, both within and across multiple clusters, can eschew the transistor delays of crossbars, connection blocks, and switch blocks. In this paper we investigate the usefulness of enhancing classical *Field-Programmable Gate Array* (FPGA) architectures with direct connections between LUTs. We present an efficient algorithm for searching automatically the most interesting patterns of such direct connections. Despite our methods being fairly conservative and relying on the use of unmodified standard CAD tools, we obtain a 2.77% improvement of the geometric mean critical path delay of a standard benchmark set, with improvement ranging from -0.17% to 7.3% for individual circuits. As modest as these results may seem at first glance, we believe that they position direct connections between LUTs as a promising topic for future research. Extending this work with dedicated CAD algorithms and exploiting the increased possibilities for optimal buffering, diagonal routing, and pipelining could prove direct connections important to the continuation of performance improvement into next generation FPGAs.

ACM Reference Format:
Stefan Nikolić, Grace Zgheib, and Paolo Ienne. 2020. Straight to the Point: Intra- and Intercluster LUT Connections, to Mitigate the Delay of Programmable Routing. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '20), February 23–25, 2020, Seaside, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3373087.3375315

1 INTRODUCTION

FPGAs are a great way of giving users access to the latest technology and all the speed benefits it brings. However, the very programmable nature of FPGAs that lends them the flexibility necessary for achieving this distribution of development costs on a large number of users makes the circuits implemented on them several times slower than the equivalent specialized circuit fabricated in the same technology [11]. Much of this additional delay is contributed by the programmable routing structure and mitigating its delay without excessively compromising its flexibility is particularly appealing. Let us illustrate the cost of having programmable routing. Figure 1a shows a portion of an island-style FPGA fabric and a minimum length path that a signal needs to traverse from an output of one LUT to an input of another, residing in an adjacent cluster. Figure 1b shows what this path might have looked like if the architecture contained a direct connection between the two LUTs—quite different, indeed. Add to this the fact that due to routing congestion, the path of Figure 1a might not always be as short as depicted there, while the path of Figure 1b is indifferent to congestion and some motivation for direct connections between LUTs may arise.



(a) No direct connections.

(b) With direct connections.

Figure 1: Potential benefits of introducing direct connections between LUTs. The figure shows a minimum length path traversed by a signal generated by one LUT and consumed by another, residing in a neighboring cluster. In (a), the connection is in an architecture without direct connections between LUTs and, in (b), in an architecture with direct connections between LUTs.

FPGA '20, February 23–25, 2020, Seaside, CA, USA

ACM ISBN 978-1-4503-7099-8/20/02.
https://doi.org/10.1145/3373087.3375315

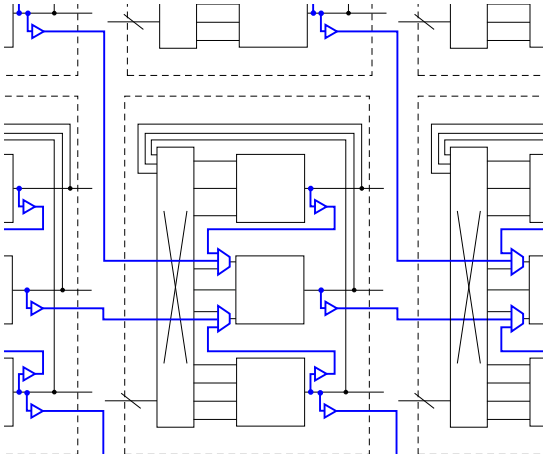


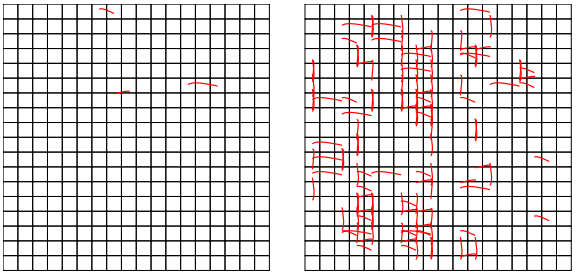
Figure 2: Direct connections considered in this work. Blue, thicker lines show direct connections between LUTs that can be both within the same cluster and in different clusters. Each direct connection is driving the particular LUT input through a multiplexer, which enables selecting the driver from multiple direct connections, as well as the programmable routing structure. The purpose of this style of connections is to retain all of the original flexibility of the architecture.

2 RELATED WORK

This idea is nothing new. Especially in the early days of FPGAs, there have been some very interesting architectures incorporating direct connections between LUTs, such as the *Xilinx XC4000* [9], with a two-level binary tree pattern of direct connections, *UTFPGA-1* from the University of Toronto, with a chain of three LUTs [5], and the *Triptych* [4], with a pattern mimicking the reconvergent fanouts that commonly occur in digital circuits. With the advent of the *Clustered Logic Blocks* (CLBs), which allowed short connections to be realized in a much faster manner than before while still keeping a high degree of flexibility, patterns of direct connections reduced mostly to chains, both for passing special carry-style signals [12] and general LUT-generated ones [7, 10, 16]. CLBs are themselves a rough, if very effective, approximation of inherent circuit connectivity that older architectures like the Triptych attempted to capture through the use of direct-connection patterns. To the best of our knowledge, there were no attempts to bridge this divide with something bolder than a simple chain. This work strives to reconcile the two approaches to profit from the inherent connectivity properties of digital circuits and by augmenting classical CLB-based island-style FPGA architectures with patterns of direct connections between LUTs, regardless of whether they reside in the same cluster or not.

3 WHICH DIRECT CONNECTIONS?

In this work we attempt to assess the value of introducing direct connections between LUTs both within the same cluster and in different clusters, with respect to critical path delay reduction. We assume that each of the direct connections can be optionally selected by configuring the appropriate multiplexer, as illustrated in Figure 2. This enables us to add direct connections on top of any standard architecture, without reducing its routing flexibility and, thus, making it possible to always use the same implementation of a particular circuit as the one found for the original architecture. Essentially, this limits the possible damage to the critical path delay caused by the introduction of the direct connections to the delays of the added multiplexers—any improvement has only this small penalty to overcome. Also, we thus retain the possibility to use the existing CAD algorithms (those in the *VTR* framework [14], in particular) without any modification. This is certainly suboptimal but we wish to focus on the exploration of the design space: we aim at answering the question "what pattern of direct connections serves best the critical path reduction?"



(a) Before permutation. **(b) After permutation.**

Figure 3: Influence of LUT permutation on usability of direct connections. The figure shows the same placement of the *sha* benchmark from the *VTR* set (a) before and (b) after permuting the LUTs inside the clusters. Each cell is a location on the fabric grid. Connections implemented as direct are shown in red.

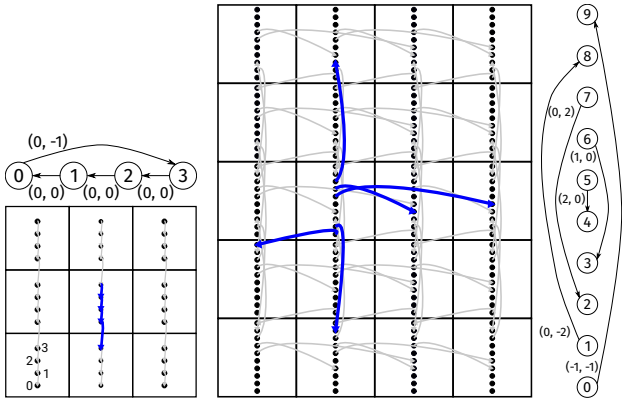
To benefit from this tight bound on negative improvement, we always keep the same packing and placement as produced for the underlying architecture. We only use the direct connections opportunistically, where the possibility to use such a connection is naturally created during the placement process. Apart from the placement of clusters in the programmable fabric, the number of such opportunities is highly dependent on the local placement of LUTs within the clusters—and thus potentially underwhelming. Indeed, Figure 3a shows in red the signals of a particular benchmark circuit implemented on one particular architecture that were successfully converted to use direct connections: as one could fear, a negligible number. Permuting the LUTs inside the clusters in an attempt to maximize the usage of direct connections results in the situation of Figure 3b. For the reasons apparent from these two figures, we slightly depart from a purely opportunistic approach by performing explicit LUT permutation between the placement and the routing stage.[1]

The previous paragraph anticipates our only modifications to an ordinary FPGA flow, more precisely described in the experimental methodology section. Our main contribution here, with this minimalist approach to exploiting direct connections, is to explore systematically the huge space of possible regular direct-connection patterns. We will proceed as follows: Section 4 describes more precisely the space of the direct-connection patterns we explore. Section 5 introduces a search algorithm that efficiently explores the vast space described in the previous section. The experimental setup, including circuit-level modeling and the above LUT permutation algorithm, is detailed in Section 6, while the experimental results themselves are presented in Section 7. Final conclusions and comments on future work are made in Section 8.

4 THE SPACE OF POSSIBLE CONNECTIONS

In this section, we define the space of direct-connection patterns that we consider as possible enhancements to classical architectures.

[1] Ideally, the routing algorithm would dynamically permute the LUTs where appropriate, but, due to the current limitations of academic tools [13], we are forced to resort to explicit permutation.



(a) Columnar chain. **(b) Pattern used in Figure 3b.**

Figure 4: Using static graphs to describe patterns of direct connections. Each square represents a cluster and each point represents a LUT. Blue edges emphasize the outgoing edges of one replica of the node set (one cluster).

4.1 Formalization of the Connection Patterns

Let us, for the moment, neglect disruptions to the regularity of a programmable fabric in the form of columns of hard-IPs, I/O blocks, etc. Furthermore, let us consider the fabric infinite. Then, an FPGA is nothing but a 2D integer lattice, with each of its points representing a cluster of logic. Let us now borrow the concepts of static and periodic graphs [8].

Definition 1. *A 2D static graph is an edge-weighted directed multi-graph* $S = (V, E \subseteq V \times V \times \mathbb{Z}^2)$.

V and E designate the node and the edge set, respectively. Multiple edges distinguished by their weight vectors can exist between two nodes, as can loops, making the graph a multi-graph.

Definition 2. *A 2D periodic graph P induced by a 2D static graph $S = (V, E)$ is the graph obtained by replicating V and placing it at the points of a 2D integer lattice, and for each $e = (u, v, \vec{w}) \in E$ and each $\vec{p} \in \mathbb{Z}^2$ adding an edge between the replica of the node u whose position vector is $\vec{p}$ and the replica of the node v whose position vector is $\vec{p} + \vec{w}$.*

If we represent each LUT of the cluster by a node of S, we can see that the classical island-style FPGA without any direct connections between LUTs would be represented by an edgeless graph. On the other hand, an FPGA with a chain connecting LUTs in the same column would be represented by the static graph of Figure 4a, for the case of a four LUT cluster. Another example of a slightly more complex pattern—the one for which the mapping of Figure 3b was produced—is shown in Figure 4b. Here we can see, for example, that the edge $(u, v, \vec{w}) = (5, 4, (2, 0))$ represents the connection between the fifth LUT of each cluster and the fourth LUT of the second cluster to the right, within the same row of the fabric.

The architectural search space considered in this paper coincides with the space of all patterns representable as a static graph on N nodes with M edges, and edge-weight bounded by w (that is, absolute values of both weight components are smaller or equal to w). Here, N is the cluster size while M and w are parameters. To give some numerical sense of the size of this design space, we could consider the following. There are N^2 ways to choose the endpoints of each edge and $2w + 1$ possible values for each component of its weight. Hence, we can choose the M edges from $((2w + 1)\,N)^2$ possible ones.[2] For $(N, M, w) = (10, 20, 4)$, which are not particularly large numbers, given the cluster sizes and routing segment lengths of modern FPGA architectures, this amounts to $\sim 10^{59}$. We will describe how we handle this problem size in Section 5.

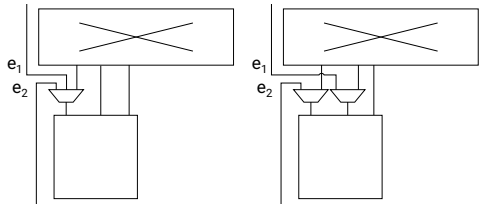


Figure 5: Multiplexer placement trade-offs. In the architecture on the left, the connections e_1 and e_2 are sharing a common multiplexer and, thus, can not be used simultaneously. This is addressed in the right architecture where both direct connections can be used and they are delayed by faster 2:1 multiplexers. However, another input of the LUT is now driven by a multiplexer, increasing the probability that the critical path delay would increase when the direct connection is not used.

4.2 Fully Specifying an Architecture

To fully specify an architecture, besides the description of the pattern of direct connections in the form of a static graph, we also need a description of the underlying FPGA architecture to which the pattern will be added. As mentioned before, in this work, each of the direct connections is optionally selected through an appropriate multiplexer and, when not used, the LUT input comes from the ordinary programmable interconnect structure—as if the direct connection did not exist at all (Figure 2). This kind of decoupling allows any classical FPGA architecture to be augmented by direct connections. However, it also means that the static graph in which nodes represent LUTs does not quite contain all the information necessary to specify an architecture, because it ignores which sets of direct connections targeting a specific LUT share a multiplexer. This actually makes an important difference because of the trade-off between usability of direct connections and delay to LUT inputs (Figure 5). We could easily represent multiplexer sharing by including the LUT input pins in the node set of the static graph but this would greatly increase the already huge search space. Therefore, we simply distribute the connections between the inputs of their target LUT in a round-robin fashion, thus keeping the inputs as equally delayed as possible. All source nodes are assumed to correspond to the output of the multiplexer selecting between the combinational and the registered output of the LUT.

5 SEARCHING A HUGE DESIGN SPACE

In this section we describe the reasoning about our approach to searching the space defined earlier, as well as the details of the search algorithm itself.

5.1 Our Search Space: A Naive View

Given an FPGA architecture, our task is to select some M edges that will enter the edge set of the static graph describing the pattern of direct connections that we intend to augment it with. Let us, for the moment, disregard the fact that our primary goal in doing all this is to reduce the critical path delay. Then, the value of adding a new edge to the edge set is in increasing the number of circuit connections covered by the direct connections. Each potential static

[2] Of course, this is a very loose bound because it does not account for isomorphisms. The intention is only to give some rough sense of the scale of the problem.

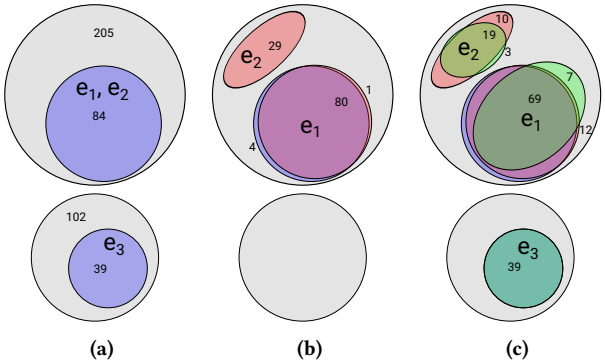


Figure 6: Analogy between our best pattern search and the Maximum Coverage problem [18]. Sets of connections of a particular placement of the *sha* benchmark coverable by the edges $e_1 = (1, 9, (0, -1))$, $e_2 = (2, 8, (0, -1))$, and $e_3 = (8, 3, (0, -3))$ are shown. The grey sets include all coverable connections, while the blue ones include the edges covered by the pattern composed of the single respective edge, after a particular permutation of LUTs. Connections covered by the patterns composed of e_1 and e_2, and all three edges are shown in red and green, respectively. Coverage achieved by the combinations of multiple edges is larger than each of their individual coverages, even if they overlap entirely (e.g., e_1 and e_2), but smaller than their sum, even if the individual coverage sets are disjoint (e.g., e_2 and e_3).

graph edge can be assigned a set of circuit connections it can cover. This set is dependent on the topology of the circuit to be mapped on the FPGA and its particular placement. Because we allow arbitrary LUT permutations, the set of connections that can potentially be covered by an edge of the static graph is the set of all connections between clusters whose relative position corresponds to the weight of the edge, irrespective of the precise location of the endpoint LUTs. In Figure 6a, these sets are represented in grey, for the edges $e_1 = (1, 9, (0, -1))$, $e_2 = (2, 8, (0, -1))$, and $e_3 = (8, 3, (0, -3))$, and a particular placement of the *sha* VTR benchmark. Because edges e_1 and e_2 have the same weight, the sets of connections they can cover are identical, while the set of edge e_3 is disjoint from them, due to its different weight. We could attempt to choose the M edges so as to maximize the union of the sets of connections they can cover. Because the grey sets of Figure 6a are either identical or disjoint, we could trivially achieve this by greedily choosing edges with disjoint sets. Note, however, that this would be misleading, as it would appear that choosing an edge with the weight equal to the weight of some edge already in the static graph would bring no benefit, due to their sets of coverable connections coinciding. Intuitively, of course, we would expect improvement in some cases, as there could be more than one connection between any two clusters, whereas a single static graph edge of the appropriate weight could cover only one of them. Likewise, it would appear that the coverage of edges with different weights is entirely additive, which is not really the case, as we will soon see.

5.2 Our Search Space: One Step at a Time

A more realistic view of the extent to which a particular edge may contribute to covering the connections of the circuit to be mapped can be obtained by measuring the coverage achieved by the pattern composed only of that edge. To perform such a measurement, we already need to consider LUT permutation. When the pattern is composed of any single one of these edges, or more generally, when the total degree of the static graph is bounded by one (that is, no two edges share a node and there are no loops), the set of covered edges constitutes a matching [3]. We search for maximal matchings with the following simple greedy algorithm:

(1) Mark position of all LUTs as *free*.
(2) Sort the circuit connections in decreasing length.
(3) For each connection (u, v), between LUTs in clusters at an offset $\vec{w}$, if there is an edge $(u_p, v_p, \vec{w})$ in the static graph, and u and v are either free or fixed to u_p and v_p, respectively, cover (u, v) and fix positions of u and v to u_p and v_p, respectively.

After applying this algorithm to each of the three single-edge patterns, we obtain the blue sets of Figure 6a. Whereas the grey sets corresponded to the unions of all sets of connections covered by the pattern under all possible permutations, the blue ones are the sets covered under one particular permutation. Hence, for instance, if the pattern includes only a single edge of weight $(0, -1)$, a subset of the 205 connections with 84 elements can be covered. Note that the covered sets of e_1 and e_2 again coincide, while that of e_3 is again disjoint from them. As it appears, apart from the numerical changes of the set cardinality, we are back to where we begun—we have the same coverage model that appears to be trivial to solve, but counters our intuition.

5.3 Our Search Space: Combining Steps

So let us combine e_1 and e_2 into one common pattern. The degree of this pattern is still bounded by one, so our algorithm still works, and the coverages it produces are overlaid in red, in Figure 6b. Now the situation already starts becoming more intuitive: The sets of connections actually covered by e_1 and e_2 are disjoint, as we can cover each connection only once. The combined coverage is now larger than either of the individual coverages (that were in fact the same), so there is indeed benefit from selecting a second edge of the same weight, as we intuitively predicted in the beginning. However, the combined coverage is smaller than the sum of the two individual coverages, indicating that coverage is really not additive.

What happens if we add e_3 to the pattern as well? Its coverable connection set (grey) was disjoint from the others, so will the combined coverage simply be the sum of its coverage and that of e_1 and e_2 combined? The new coverage sets are overlaid in green, in Figure 6c. Both the sets of e_1 and e_2 shrunk with the introduction of e_3. This is because e_3, which had precedence due to sorting of circuit connections based on length, fixed the positions of its endpoints in such a manner that e_2 could no longer cover some of the connections it covered previously. In turn, e_2 took over some of the connections from the intersection with e_1, causing a change in its set as well. Hence, even the coverage sets that originally appeared to be disjoint can have "induced" intersections, caused by conflicting permutation requirements.

5.4 Our Search Problem: An Analogy

The last subsections give us a taste of the challenges that our search problem faces. *Maximum Coverage* is a well-known intractable problem where the goal is to select a fixed number of sets out of several sets which may have some elements in common; in doing so, one wants to maximize the cardinality of the union of the selected sets [18]. A standard greedy approach to solving the problem is known to give the best achievable polynomial-time approximation, unless **P** = **NP** [6]. Without any rigorous analysis of the relations between the problem of finding the optimum set of direct connections and maximum coverage, relying on the intuition

```
search(N, M, w, cA, cB, cC){
    g_best = ([0, N), {})
    for m in [0, M) {
        G = get_all_unique_expansions(g_best, w)
        filterA(G, cA)
        filterB(G, cB)
        filterC(G, cC)
        g_best = pick_best(G)
    }
}

filter(G, c) {
    // A prototypical filter
    for g in G
        for b in benchmarks
            scores[b][g] = compute_score(g, b)
    for b in benchmarks {
        sort(scores[b])
        for g in G
            ranks[g] += index(scores[b], g)
    }
    sort(ranks)
    return ranks[0:c]
}
```

Figure 7: Pseudocode of our greedy pattern search algorithm.

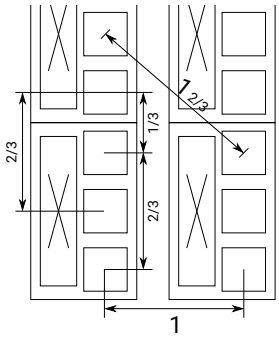


Figure 8: Connection length calculation. Several examples of connection length calculation are shown for a cluster size of three with LUTs vertically stacked. The distance between two vertically adjacent LUTs is equal to $1/N \times L_T$, where N is the cluster size and L_T the height of the tile. The distance between two horizontally adjacent LUTs is equal to L_T. N and L_T are 3 and 1 in this example, respectively.

developed by the observations in the previous example instead, we adopt the greedy approach to solving our search problem.

5.5 The Greedy Algorithm

The pseudocode of our search algorithm is shown in Figure 7. The algorithm starts from an edgeless graph on N nodes and performs M rounds of listing all unique expansions of the previous best-scoring pattern and choosing the new best one among them. During expansion listing, edges are first added to the static graph without weight assignment. All N^2 such expansions are generated and the resulting graphs are split into classes of isomorphic ones. The next stage is weight assignment, where, for each tried weight, a class representative that results in the minimum length of the added connection is chosen. This is based on the assumption that the LUTs are stacked vertically, like in the Stratix FPGAs [12] (Figure 8).

The best pattern is chosen based on the postrouting critical path delay, as this is what we wish to optimize. Because there are thousands of new patterns appearing in each search iteration, running the complete CAD flow is infeasible and we need fast predictors to filter out most of the weak candidates. A prototypical filter is represented by the function *filter* of Figure 7. For each circuit in the benchmark set and each pattern, a score is computed. Patterns are then ranked for each of the circuits and c of them (where c is a

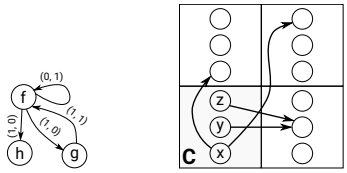


Figure 9: An example to demonstrate our coverage-based filters. The static graph on the left describes a pattern, while the fabric on the right shows a cluster, C, of a packed and placed circuit, along with three of its immediate neighbors. Only circuit connections originating in cluster C are shown.

parameter) with the minimum sum of ranks are kept for the next filtering stage. Details of the filters are discussed below. The first two are designed to be fast and based only on coverage. They also serve the purpose of anticipating the accumulated effect of adding multiple edges to the pattern, that may not be immediately reflected on the delay. The final filter is considerably slower, but also more accurate and tries to predict the critical path delay.

5.6 Pruning the Candidates: The First Filter

The first filter groups the pattern edges by their weights and treats the groups completely independently, attempting to quickly assess how much circuit connection coverage the chosen set of weights can achieve. For a given pattern P, edge weight $\vec{w}$, and a cluster C of the placed circuit, it works as follows:

(1) For each LUT u in P, count the number of edges $(u, v, \vec{w})$ and store these counts in $\vec{s_p}$.
(2) For each LUT u' in C, count the number of connections starting at u' and ending in a cluster at an offset equal to $\vec{w}$ and store these counts in $\vec{s_c}$.
(3) Sort the vectors $\vec{s_p}$ and $\vec{s_c}$ and compute the score $S(P, C, \vec{w})$ as $\sum_i min(\vec{s_p}(i), \vec{s_c}(i))$.

In the example of Figure 9, for $\vec{w} = (1, 0)$, vectors $\vec{s_p}$ and $\vec{s_c}$ are equal to $(2, 0, 0)$ and $(1, 1, 0)$, respectively. Hence, $S(P, C, (1, 0))$ is 1 in this case. The score $S(P, C)$ is obtained simply as $\sum_{\vec{w}} S(P, C, \vec{w})$, while the final score $S(P)$ is obtained by summing $S(P, C)$ for all C. In the running example, all the remaining count vectors are equal to $(1, 0, 0)$ so the score $S(P, C)$ is equal to 3. A dual version of the filter considering incoming connections is applied in the same manner, and the two scores are added for each pattern.

This is a very simple filter that completely ignores the mutual influence of various pattern edges through conflicting permutation requirements, as well as distribution of edges of different weights among LUTs, therefore largely overestimating the possible coverage. It has one important virtue, however—it is very fast to compute. Moreover, because we are keeping the same packing and placement for a given benchmark and random seed pair, we can precompute the score for each $(\vec{w}, \vec{s_p})$ pair on each particular placement. Despite its simplicity, this filter is effective in reducing the number of possible expansions from several thousands, with many clearly inferior to others and many essentially equivalent, down to a figure manageable by the further stages of the algorithm.

5.7 Pruning the Candidates: The Second Filter

The second filter attempts to fix some of the problems of the first one. The first filter essentially assumes that the static graph edges can be swapped between different LUTs dynamically, to maximize the number of covered circuit connections incident to the given cluster. Here, the static graph edges are really tied to their endpoint LUTs. An optimal permutation of the LUTs in a given cluster is computed,

considering that each LUT is one endpoint of its incident connections, and that the other endpoints can be freely permuted, so as to make the connections truly covered. This assumption means that the effects of conflicting permutation requirements are still largely neglected, but also that the scores can still be quickly computed. Computation of the score $S(P, C)$ proceeds as follows:

(1) For each LUT u in P, form the multiset of the incoming edge weights $m_i(u)$ and the multiset of the outgoing edge weights $m_o(u)$.
(2) Perform the same for each LUT u' in C.
(3) For each u in P and each u' in C, compute the score $s(u, u')$ as $|m_i(u) \cap m_i(u')| + |m_o(u) \cap m_o(u')|$.
(4) Populate the score matrix M_s with rows corresponding to all u in P and columns to all u' in C by computing the appropriate scores $s(u, u')$.
(5) Solve the assignment problem [15] on M_s to obtain the score $S(P, C)$.

The final score $S(P)$ is again computed as $\sum_C S(P, C)$. In the running example, $m_i(f) = \{(0, -1)\}$, $m_o(f) = \{(0, 1), (1, 0), (1, 0)\}$, while $m_i(x) = \{\}$ and $m_o(x) = \{(0, 1), (1, 1)\}$. Hence, $s(f, x) = 1$. The complete score matrix is

$$M_s = \begin{array}{c} \\ f \\ g \\ h \end{array} \begin{array}{c} \begin{array}{ccc} x & y & z \end{array} \\ \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \end{array}$$

with the entries participating in an optimum assignment highlighted. $S(P, C)$ is now equal to 2, which is less than 3 reported by the first filter, because the distribution of edges with different weights among the LUTs is now taken into account.

5.8 Pruning the Candidates: The Last Filter

So far, the filters had only focused on assessing coverage, without any relation to optimizing delay. The third filter is the first point when the timing directly enters the edge selection process. It relies on optimal permutation of a small subset of (near) critical nodes, using *Integer Linear Programming* (ILP), to obtain a critical path delay prediction, based on the postplacement assessment and the improvement achieved after LUT permutation. The details of choosing which nodes to permute are described in Section 6.3, where the process is referred to as *solving the critical core*.

Finally, because intracluster connections occur about as often as all intercluster connections combined, the first two filters would tend to select mostly zero-weighted edges. To prevent this, we split the search into two stages—first for nonzero-weighted edges, followed by only zero-weighted edges, with the previous set kept fixed. We will come back to the problem of intracluster connections in Section 7.5.

6 EXPERIMENTAL SETUP

Our evaluation flow is based on the VTR framework [14] and depicted in the flowchart of Figure 10. For comparison, the underlying architecture is passed through the traditional CAD flow of VTR, before being augmented by direct connections. The implementation flow of the pattern-enhanced architectures starts by reading the underlying architecture description and the description of the pattern, and generating a modified architecture that incorporates

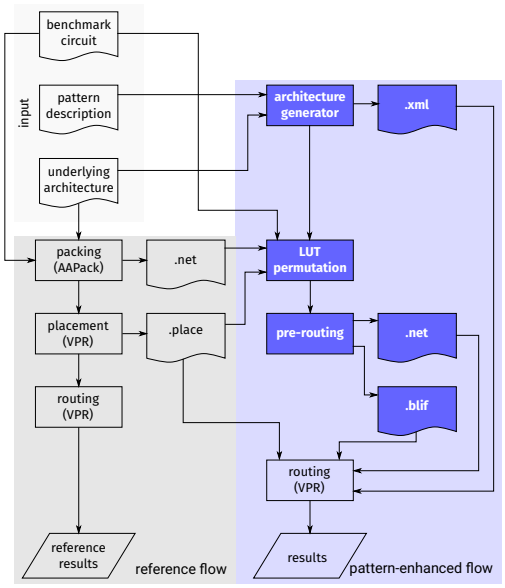


Figure 10: The pattern evaluation flow. Stages that are not part of the traditional VTR flow are shown in blue.

the pattern into the underlying architecture. The packing and placement files produced in the reference flow are passed to the LUT permutation algorithm along with all the postplacement timing information, the benchmark circuit netlist, and the modified architecture model. After LUT permutation, those circuit connections that are implementable as direct are prerouted, and the modified packing and circuit netlist files are generated. These two files are then passed to the router along with the unmodified original placement, to obtain the final implementation of the circuit in the new architecture.

To reduce the measurement noise, all benchmark circuits are placed with five different placement seeds; reported results are the median values of the five experiments. To further minimize noise, the *delay targeted routing* algorithm of Rubin and DeHon [17] is used for routing each architecture.

In all experiments, the 40nm *k6_N10_mem32K_40nm* architecture from the VTR project is used as the underlying (reference) architecture. In the following sections, we describe how this architecture is transformed to include the specified direct connections. To appropriately model the added circuitry, we perform SPICE simulations in the closest technology node we had access to—*45nm PTM HP* [19]—and scale the delays to match those reported in the underlying architecture. The details on modeling and the stages of the pattern-enhanced architecture CAD flow where it departs from the reference one are also discussed in the following sections.

6.1 Architecture Generation

VTR's XML format used to describe the underlying architecture strives to be compact by defining each block type (e.g., LUT) only once and then specifying the number of instances of that block. We need to find a mapping between the node set of the static graph representing the pattern of direct connections and these iteratively specified blocks. Hence, the original specifications are "unrolled" so that each instance becomes a uniquely identified block, corresponding to one node of the static graph.

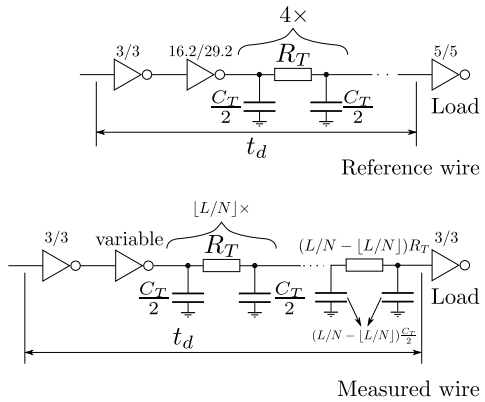


Figure 11: Experimental setup for direct connection delay measurement.

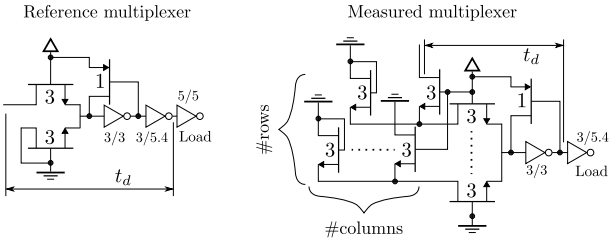
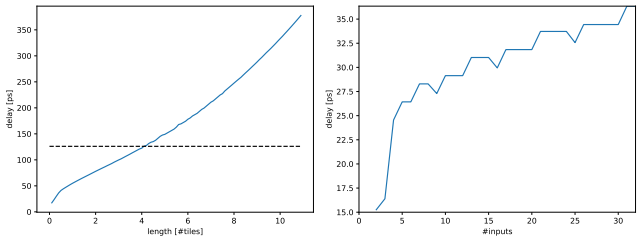


Figure 12: Experimental setup for multiplexer delay measurement.



(a) Direct connection delays. **(b) Multiplexer delays.**

Figure 13: Direct connection and multiplexer delays. The dashed line corresponds to the delay of the reference four-tile-long global-routing channel track of the underlying architecture.

Each direct connection receives a unique driver block that is tied to the output of its source node. The entire connection delay, along with that of the driver, is assigned to this block as the loads of connections are known in advance. The driver blocks implement a separate *repeater* blif primitive, which enables simple prerouting of a circuit using direct connections.

6.2 Circuit-level Modeling

We model direct connections as sequences of Π-type RC stages [2]. The per-tile-length RC parameters are those of the global routing tracks of the underlying architecture. Each direct connection is modeled by $\lfloor L/N \rfloor$ full-tile stages plus one stage with RC parameters scaled by $L/N - \lfloor L/N \rfloor$, where L is the length of the connection calculated as shown in Figure 8 and N the size of the cluster. The drive strength of the first driver stage is set to that of the first stage of the global wire driver of the underlying architecture, while the second stage drive strength is swept in both directions, starting from that of the second stage of the global wire driver. Once the delay stops decreasing, the sweep is terminated and the last considered size is taken. When that is advantageous (for short wires), we remove the second stage of the driver. The SPICE measurements on direct connections are made using the circuit shown at the bottom of Figure 11. The model of a single global wire of the underlying architecture is shown at the top of the same figure. The slightly reduced load applied to the direct connection (i.e., 3/3 inverter instead of a 5/5 one) reflects the fact that it drives only a single LUT input. We multiply all measured delays by the ratio between the delay of the reference wire reported in the underlying architecture and the corresponding delay obtained in our technology through SPICE simulation. The final delays are reported in Figure 13a.

Multiplexers are modeled as two-level structures [2]. The pass transistor and first-stage driver sizes are taken from the switch block of the underlying architecture. The second-stage driver NMOS and PMOS are assumed to be 3 and 5.4× wider than the minimum width transistor, respectively. We use the second stage as the load, assuming that a single stage driver is sufficient, given that the multiplexer is driving only one LUT input. Figure 12 shows the circuit used for measuring the multiplexer delay (on the right) and the circuit of a 2:1 one-level multiplexer with a two-stage driver used for reference delay measurement (on the left). We scale all the delays by the ratio of the delay of the *BLE* [2] output multiplexer reported in the underlying architecture and the delay measured on the reference circuit. The resulting delays are reported in Figure 13b.

```
permute(pattern, clusters, placement, timing_graph){
    for c in clusters {
        randomly_permute(c)
    }
    E = find_candidate_cons(clusters, placement, pattern)
    core = extract_critical_core(timing_graph, E, size)
    permutations, fixed_nodes = solve_core(core)
    permutations = anneal(clusters \ fixed_nodes)
    return permutations
}

extract_core(timing_graph, E, size) {
    for e in E {
        timing_graph.delay(e) = min_direct_con_delay(e)
    }
    do_sta(timing_graph)
    covered = E
    V = nodes_incident_to(E)
    while |V| > size {
        e = least_critical(covered, timing_graph)
        covered.remove(e)
        timing_graph.delay(e) = programmable_delay(e)
        do_incremental_sta(timing_graph)
        update(V)
    }
    return V
}
```

Figure 14: Pseudocode of the LUT permutation algorithm.

6.3 LUT Permutation

As mentioned in Section 3, the initial placement of LUTs within the clusters, agnostic of our direct connections, does not offer sufficient opportunity to use them. To overcome this issue, we employ the LUT permutation algorithm whose pseudocode is shown in Figure 14. Firstly, we find the circuit connections that might possibly be covered by a direct connection—that is, all connections between clusters whose relative distance equals the weight of some edge of the pattern (function *find_candidate_cons*). All other connections are irrelevant for the permutation process. Our primary goal in permuting the LUTs is to cover as many critical circuit connections as possible by the fast direct connections of the pattern. Since we are making decisions after placement, the timing predictions are already quite accurate and the number of (close to) critical nodes

is usually fairly limited. Hence, we can extract the critical portion of the circuit—the *critical core*—and optimize its coverage exactly. For this, we proceed as follows: Initially, we assume that all potentially coverable connections we just identified are part of the critical core. We then reduce the core size greedily, by removing the edge of the maximum slack, until the core size is less than a given parameter (function *extract_core*). And, finally, ILP is used to find a legal permutation of the clusters containing the core nodes so that the critical path delay is minimized. In order to keep the runtime reasonable, a timeout of one minute is imposed on the solver but, in practice, the optimum is often reached much faster. Because the core contains only a small fraction of the entire circuit, the resulting permutation does not maximize the overall usage of direct connections, which is important for reducing pressure on general routing. To achieve that, we conclude by performing a low temperature simulated annealing, with a setup similar to that of VPR's non-timing-driven placement algorithm. A concrete example of the difference created by LUT permutation with respect to the possibility of using direct connections was shown in Figure 3b.

6.4 Prerouting

After completing LUT permutation, all circuit connections that match an edge of the static graph are implemented using the appropriate direct connection. This necessitates LUT input permutation. When several connections share a multiplexer, the one with the least postplacement slack is selected, while the others are left to be implemented using general routing. The packed netlist is modified accordingly, as is the benchmark circuit netlist, by instantiating *repeater* subcircuits between sources and targets of the prerouted connections. Thus, instead of truly prerouting the circuit, we leave VPR with only one choice for routing each prerouted connection.

6.5 Further Assumptions and Limitations

In order to preserve the legality of the circuit implementation after explicit permutation of LUTs, the routing algorithm must no longer consider all LUT outputs to be equivalent. Due to VPR's current lack of support for selectively disabling the permutation of outputs [13], this calls for preventing any route-time output permutation whatsoever. Allowing LUT output permutation by the router for the reference architecture alone would be too unfair to the pattern-enhanced architectures. On the other hand, disabling permutations altogether would show unrealistically large benefits of direct connections, because there is clearly a trade-off between choosing the permutation that optimizes direct-connection utilization and the one that optimizes congestion and wire length in the programmable routing structure. Missing a better alternative at the moment, we generate 30 different random permutations of each packing for the reference architecture, prohibit any further output permutation by the router, and choose the permutation resulting in the median critical path delay as the representative one.

Because our permutation algorithm is currently not capable of determining local routability, we require that the underlying architecture has a fully populated crossbar and that all LUT inputs are permutable. This means that we are unable to consider fracturable LUTs for the time being. Similarly, because the algorithm is not aware of a priori fixed LUT positions in certain clusters, we currently do not support carry chains. Hard-IPs remain fully supported. All these limitations could cause the direct connections to appear slightly more appealing than they would have, had the architectures

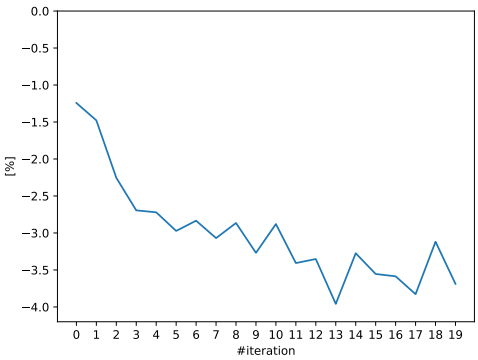


Figure 15: Evolution of the relative change of the geomean postrouting critical path delay. Despite some unsurprising noise, the overall trend seems clearly towards a monotone improvement.

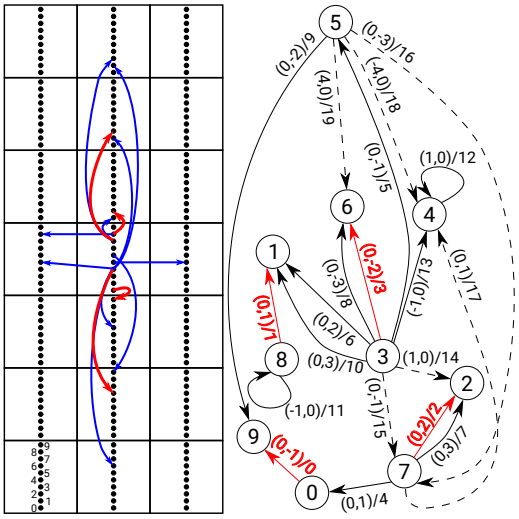


Figure 16: Pattern obtained through the search process. Dashed edges are added after the iteration which results in the maximum achieved delay improvement (iteration #13) and are thus not considered further in our analysis. Connections in the fabric depiction on the left correspond to the edges of this best found pattern originating in one particular cluster. The four red edges are responsible for achieving 68% of the total achieved gain.

more representative of the current state of the art been used; still, we do not believe that this changes our final conclusions.

The search space is limited to patterns on 10 nodes, which conforms to the cluster size of the underlying architecture, and up to 20 connections of Chebyshev length smaller than or equal to 4 (that is, $(N, M, w) = (10, 20, 4)$). The search algorithm filters are set to pass 100, 10, and 3 expansions, respectively. The size of the critical core is set to 100 nodes and *glpk* version 4.64 [1] is used to solve it. In all experiments, the routing channel width is set to 300. A subset of VTR benchmarks is used for all experiments, with the two largest excluded due to the prohibitive runtime requirements.

7 EXPERIMENTAL RESULTS

In this section we discuss the outcome of the search described in the preceding sections.

7.1 Intercluster Connections: Convergence

As mentioned in Section 5.6, the much larger number of intracluster connections appearing in an average circuit would overwhelm the

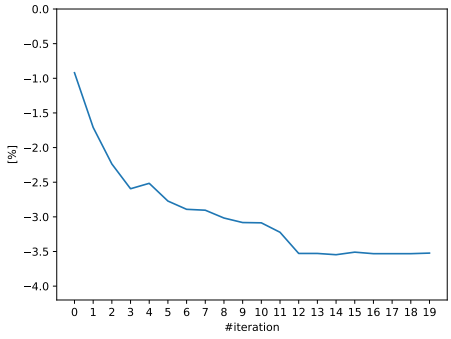


Figure 17: Evolution of the relative change of the geomean postplacement critical path delay prediction after critical core solving.

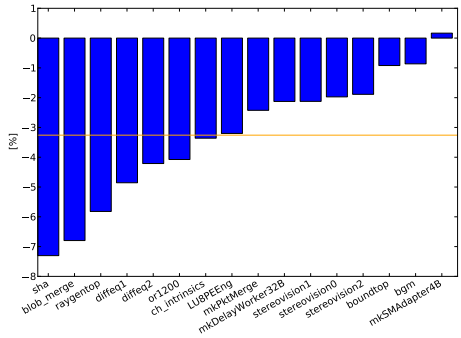


Figure 18: Relative change of the postrouting critical path delay per benchmark. The orange line shows the decrease in the geomean critical path delay. Practically, no circuit is worsened by the presence of the direct connections and the benefit is up to 7%.

coverage-based filters, requiring the search to be split into two phases. The first phase considers only intercluster connections, which have higher potential to reduce delay when used successfully. Figure 15 shows the evolution of the relative change of the postrouting critical path delay over the iterations of this first phase of pattern search. The pattern itself is shown in Figure 16, with edges labeled as *weight/iteration*, where *iteration* is the iteration of the search algorithm when the particular edge was added.

Clearly and perhaps unexpectedly, Figure 15 is not perfectly monotonic and has considerable noise almost certainly due to low predictability of the routing process. For reference, it is interesting to inspect Figure 17, which shows the evolution of the critical path delay change after critical core solving (third filter of the search algorithm), as predicted from VPR's postplacement data. This curve is not monotonic either, due to the heuristic nature of critical core extraction and the limited time given to the ILP solver, but it is much smoother than the postrouting curve of Figure 15. This indicates that, as noisy as it is, the evolution of Figure 15 is not a result of chance but truly driven by a sound, albeit imperfect, predictor. Note that during the search, the benchmarks *bgm*, *LU8PEEng*, *stereovision0*, and *stereovision2* were temporarily removed to reduce the runtime, while *mkPktMerge* and *stereovision1* were removed to enhance stability. The critical paths of these latter benchmarks contain no connections between LUTs and are thus not directly improvable by the patterns considered here.

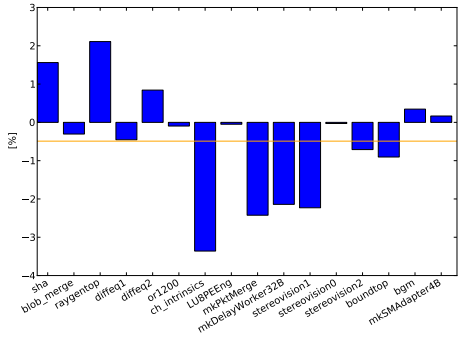


Figure 19: Influence of the permutation on delay changes. This graph shows the delay changes resulting from fixing the permutations that led to improvements of Figure 18, but not using any of the direct connections. The plot suggest that a part of the previously observed improvement is caused by the different permutations. The magnitude roughly corresponds to the difference between postplacement and postrouting delays (Figures 17 and 15).

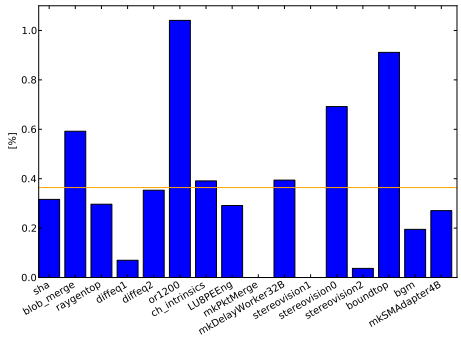


Figure 20: Maximum delay overhead. The graph shows the impact of the added multiplexer delay on the circuit implementation found for the underlying architecture. We can always opt for these implementations.

7.2 Intercluster Connections: Delay Impact

Because our ultimate goal is to minimize the postrouting critical path delay, we take the pattern at the 13^{th} iteration (with 14 edges), corresponding to the minimum of the curve of Figure 15, as the final one. Its edges are represented by solid lines in Figure 16. The per-benchmark delay changes for this pattern are shown in Figure 18.

The postplacement predictions of Figure 17 suggest that the obtained improvements did not originate primarily from noise, but implementing connections as direct. Nevertheless, it is important to know how much of this improvement comes from fixing a different LUT permutation for the pattern-enhanced than for the reference architecture. To measure that, we fix the permutations that led to the improvements of Figure 18 and report the postrouting delay change without using the direct connections in Figure 19. As we can see, part of the observed improvement indeed comes from the chosen permutation. This explains, for instance, why the circuits *mkPktMerge* and *stereovision1* which do not even have connections between LUTs on their critical paths appear as improved. Moreover, the magnitude of the improvement contributed by the chosen permutation corresponds well with the mismatch between the observed postrouting delay of Figure 15 and the postplacement delay prediction of Figure 17. Despite this noise, the main source of improvement clearly does not lie in different permutation fixing. We can conclude that the actual improvement due to the introduction of direct connections is 2.77% instead of the 3.26% reported in Figure 18, with the 0.49% difference contributed by permutation fixing.

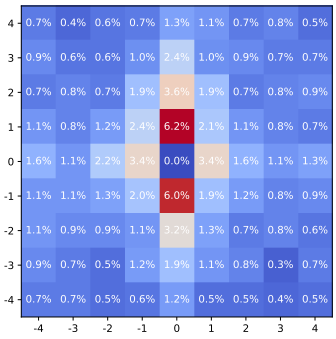


Figure 21: Distribution of target-cluster locations of intercluster connections bounded by $w = 4$. The x-axis corresponds to the horizontal offset and the y-axis to the vertical offset. The data comes from the *sha* benchmark. Dominant frequency of vertical connections and rough symmetry of the heat map help explain the choices made by the search algorithm.

In this paper we have taken a specific avenue in adding direct connections. We have introduced it in Section 3 with the visual help of Figure 2: we used direct connections connected through multiplexers at the input of the LUTs; the rationale is that this would bring a minimal penalty to the normal routing process whenever direct connections are not helpful. It seems appropriate to verify such penalty. Figure 20 shows the relative postrouting delay changes for the benchmark circuits implemented in the pattern-enhanced architecture with unmodified packing of the reference (i.e., without any LUT permutation) and no use of direct connections. As we can see, the penalty is indeed pretty insubstantial.

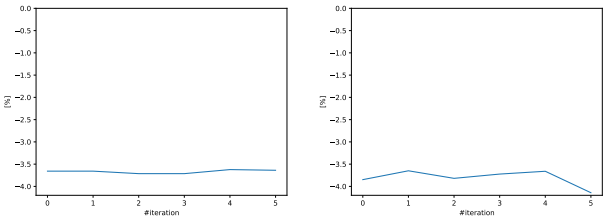
7.3 Intercluster Connections: The Pattern

Let us comment briefly on the found pattern itself. It is interesting that most of the selected edges are vertical. This results from both the placement of the circuits and the search algorithm favoring vertical edges in case of ties. The latter decision is due to their potentially shorter length (see Figure 8). To illustrate the distribution of the postplacement intercluster connections, irrespective of any direct connection, we have shown in Figure 21 a heat map of the average fraction of signals going to each location in the neighbourhood of a cluster. For this we use the *sha* benchmark which appears fairly representative of the whole set. The map indicates, for instance, that 6.2% of the signals leaving any cluster, on average, connect to a LUT in the cluster just above it. As we can see, there are indeed more connections that are vertical than those that are horizontal, while the diagonal ones are even rarer. Another interesting observation is that the heat map is fairly symmetrical, which could explain why pairs of opposed vectors often occur as consecutive pairs of edge weights chosen by the search algorithm.

Finally, the area overhead of the added connections is equivalent to 612 minimum width transistors, with 374 contributed by the drivers, and 238 by the multiplexers. This represents a mere 1.13% increase of the cluster area of the underlying architecture.

7.4 Intercluster Connections: A Trade-Off

We may note that 68% of the entire achieved gain came from the first four edges added to the pattern. They are shown in red in Figure 16. By sacrificing some of the performance gain, the required minimum width transistor investment reduces to 147 or merely 0.27% of the cluster area. Not only is the pattern formed by these four edges perfectly symmetrical in terms of edge weights, but the edges were also chosen at each step to be the shortest ones with the given weight, while maintaining the node degree bound of one. If we recall the discussion of Section 5.2, this feature greatly simplifies the problem of permutation. Hence, it is probably not only the good selection of edge weights, but also the possibility for an imperfect mapping algorithm to actually use them that makes this pattern successful.



(a) Postplacement **(b) Postrouting**

Figure 22: Relative change of the geomean critical path delay while adding local connections to the pattern of Figure 16.

7.5 Intracluster Connections

As mentioned before, starting from intercluster connections seemed more reasonable both in terms of easier filtering of best connection candidates and in terms of optimization potentials. In fact, the latter aspect turned out to be a bit of a bummer after looking at the quantitative data: Contribution of the intracluster connections to the geomean critical path delay of the benchmark set used during the search is a mere 2.84%, compared to a more significant 33.6% for intercluster. A simple application of Amdahl's law suggests to move on. Anyway, we did an experiment to add local direct connections to the best previous pattern, by including in the ILP formulation only those that have both endpoints in the critical core after it has already been extracted: this way, the solver can still return the previous solution but can also opt for covering a local connection instead, if beneficial. As we can see from Figure 22, there is very little improvement in the geomean critical path, without any visible trend towards this improvement increasing with the addition of further local direct connections. We conclude that direct connections within a cluster are less appealing than we anticipated.

8 CONCLUSIONS

In this work, we demonstrate empirically that the FPGA performance can benefit from the introduction of direct connections between LUTs, without any compromise to flexibility. Although the 2.77% average improvement of the critical path delay may not seem large, considering that we did not even use dedicated CAD tools and that many circuit-level optimizations, such as optimal buffering and diagonal routing were not performed, we believe that this result heralds an exciting topic for future research. The delay improvement of up to 7% on some circuits, at a cluster area increase of merely 1.13%, and with essentially no circuit suffering any delay penalty, suggests that direct connections can bring immediate improvement even with the current setup. We also present an efficient algorithm for searching the vast space spanned by direct-connection patterns. Our next step towards maximizing the profit from the introduction of direct connections is to develop dedicated CAD tools.

REFERENCES

[1] GLPK (GNU linear programming kit). "https://www.gnu.org/software/glpk/".

[2] V. Betz, J. Rose, and A. Marquardt. *Architecture and CAD for deep-submicron FPGAs.* Kluwer Academic, 1999.

[3] J. Bondy and U. Murty. *Graph Theory.* Springer Publishing Company, Incorporated, 1st edition, 2008.

[4] G. Borriello, C. Ebeling, S. Hauck, and S. M. Burns. The Triptych FPGA architecture. *IEEE Trans. VLSI Syst.*, 3(4):491–501, 1995.

[5] P. Chow, S. O. Seo, D. Au, B. Fallah, C. Li, and J. Rose. A $1.2\mu m$ CMOS FPGA using cascaded logic blocks and segmented routing. In *Proceedings of the International Workshop on Field Programmable Logic and Applications*, Oxford, UK, Sept. 1991.

[6] U. Feige. A threshold of ln n for approximating set cover. *J. ACM*, 45(4):634–652, July 1998.

[7] B. Gaide, D. Gaitonde, C. Ravishankar, and T. Bauer. Xilinx adaptive compute acceleration platform: Versal™ architecture. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2019, Seaside, CA, USA, February 24-26, 2019*, pages 84–93, 2019.

[8] F. Höfting and E. Wanke. Polynomial time analysis of torodial periodic graphs. In *Automata, Languages and Programming, 21st International Colloquium, ICALP94, Jerusalem, Israel, July 11-14, 1994, Proceedings*, pages 544–555, 1994.

[9] H.-C. Hsieh, W. S. Carter, J. Ja, E. Cheung, S. Schreifels, C. Erickson, P. Freidin, L. Tinkey, and R. Kanazawa. Third-generation architecture boosts speed and density of field-programmable gate arrays. In *Proceedings of the IEEE 1990 Custom Integrated Circuits Conference, May 13-16, 1990, Boston, MA, USA*, pages 31.2/1–31.2/7. IEEE, 1990.

[10] M. D. Hutton, V. Chan, P. Kazarian, V. Maruri, T. Ngai, J. Park, R. H. Patel, B. Pedersen, J. Schleicher, and S. Shumarayev. Interconnect enhancements for a high-speed PLD architecture. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA 2002, Monterey, CA, USA, February 24-26, 2002*, pages 3–10, 2002.

[11] I. Kuon and J. Rose. Measuring the gap between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215, 2007.

[12] D. M. Lewis, V. Betz, D. Jefferson, A. Lee, C. Lane, P. Leventis, S. Marquardt, C. McClintock, B. Pedersen, G. Powell, S. Reddy, C. Wysocki, R. Cliff, and J. Rose. The Stratix routing and logic architecture. In S. Trimberger and R. Tessier, editors, *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA 2003, Monterey, CA, USA, February 23-25, 2003*, pages 12–20. ACM, 2003.

[13] J. Luu. *Architecture-Aware Packing and CAD Infrastructure for Field-Programmable Gate Arrays.* Ph.D. thesis, University of Toronto, Toronto, 2014.

[14] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed, K. B. Kent, J. Anderson, J. Rose, and V. Betz. VTR 7.0: Next generation architecture and CAD system for FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 7(2):6:1–6:30, June 2014.

[15] J. R. Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1), 1957.

[16] H. Parandeh-Afshar, G. Zgheib, P. Brisk, and P. Ienne. Reducing the pressure on routing resources of FPGAs with generic logic chains. In *Proceedings of the ACM/SIGDA 19th International Symposium on Field Programmable Gate Arrays, FPGA 2011, Monterey, California, USA, February 27, March 1, 2011*, pages 237–246, 2011.

[17] R. Rubin and A. DeHon. Timing-driven pathfinder pathology and remediation: quantifying and reducing delay noise in VPR-pathfinder. In J. Wawrzynek and K. Compton, editors, *Proceedings of the ACM/SIGDA 19th International Symposium on Field-Programmable Gate Arrays, FPGA 2011, Monterey, CA, USA, February 27-March 1, 2011*, pages 173–176. ACM, 2011.

[18] V. V. Vazirani. *Approximation algorithms.* Springer, 2004.

[19] W. Zhao and Y. Cao. Predictive technology model for nano-CMOS design exploration. *ACM Journal on Emerging Technologies in Computing Systems*, 3(1), 2007.