Detailed Placement for Dedicated LUT-Level FPGA Interconnect

STEFAN NIKOLIĆ, École Polytechnique Fédérale de Lausanne
GRACE ZGHEIB, Intel Corporation
PAOLO IENNE, École Polytechnique Fédérale de Lausanne

In this work, we develop timing-driven CAD support for FPGA architectures with direct connections between LUTs. We do so by proposing an efficient ILP-based detailed placer, which moves a carefully selected subset of LUTs from their original positions, so that connections of the user circuit can be appropriately aligned with the direct connections of the FPGA, reducing the circuit's critical path delay. We discuss various aspects of making such an approach practicable, from efficient formulation of the integer programs themselves, to appropriate selection of the movable nodes. These careful considerations enable simultaneous movement of tens of LUTs with tens of candidate positions each, in a matter of minutes. In this manner, the impact of additional connections on the critical path delay more than doubles, compared to the previously reported results that relied solely on architecture-oblivious placement.

CCS Concepts: • **Hardware → Programmable interconnect**; **Placement;**

Additional Key Words and Phrases: FPGA, placement, algorithm, direct connection, LUT, LP, ILP, timing-driven

ACM Reference format:
Stefan Nikolić, Grace Zgheib, and Paolo Ienne. 2022. Detailed Placement for Dedicated LUT-Level FPGA Interconnect. *ACM Trans. Reconfig. Technol. Syst.* 15, 4, Article 37 (December 2022), 33 pages.
https://doi.org/10.1145/3501802

1 INTRODUCTION

Introducing dedicated connections between ***Field-Programmable Gate Array*** (**FPGA**) resources in order to increase performance, by avoiding multiple levels of multiplexing in the programmable routing architecture, is an old idea [6, 13]. A problem in exploring such architectures is that there could be two different causes for failing to achieve the anticipated effect of the additional connections. One could, for instance, expect that a cascade of ***Look-Up Tables*** (**LUTs**) is reasonably useful in reducing the critical path delay of a typical circuit. Failure to observe any benefit could lead to a guess that the CAD tools do not provide adequate support for such cascades. Before dedicating effort to envisioning new algorithms, it would be useful to know that the problem does not lie in the simplicity of the cascade itself, because, for instance, it cannot cover multiple fanouts or fanins. Unfortunately, without an optimal algorithm for putting the cascades to use, one cannot

Authors' addresses: S. Nikolić and P. Ienne, École Polytechnique Fédérale de Lausanne, Station 14, CH-1015 Lausanne, Switzerland; emails: {stefan.nikolic, paolo.ienne}@epfl.ch; G. Zgheib, Intel Corporation, 101 Innovation Drive, 95134 San Jose, California; email: grace.zgheib@intel.com.

1936-7406/2022/12-ART37 $15.00
https://doi.org/10.1145/3501802

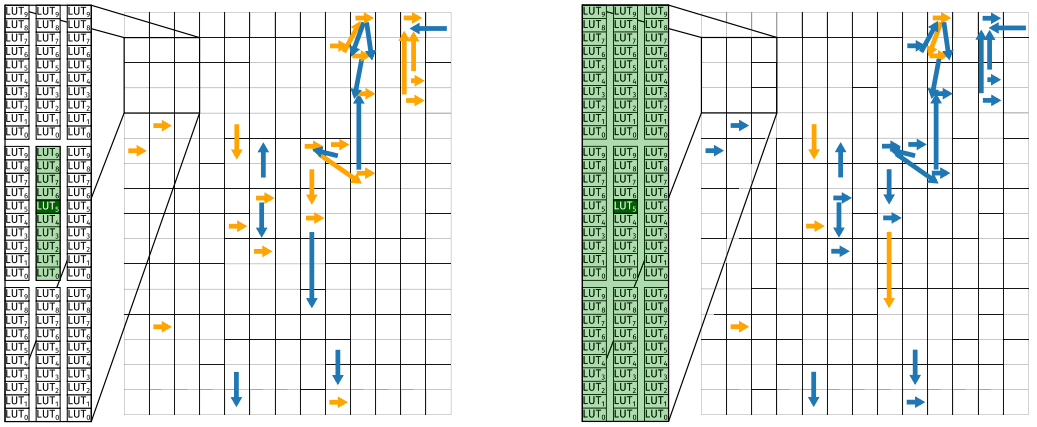


Fig. 1. Influence of movement freedom on delay minimization. Arrows depict a subset of connections of the *or1200* circuit, placed by standard VPR, on the architecture of Figure 2. The blue arrows mark the connections that are successfully implemented as direct after appropriately moving their endpoint LUTs, while those that remain programmable are shown in orange. In the left figure, LUTs can only move within their respective cluster (indicated by the light green region depicting the allowed positions for the central LUT shown in dark green), while in the right, they can also move to the adjacent clusters, resulting in a 900 ps smaller delay.

be sure which of these two potential sources led to the unexpected result. In other words, a lack of good algorithms makes it hard to assess the quality of the architectures, while the lack of a good architecture makes it hard to assess the quality of the algorithms.

In our previous work, however, we showed that there exists at least one architecture, which profits from the additional connections even if no special CAD tools are used to map circuits onto them [26]. The only CAD flow modification used to demonstrate the effectiveness of this architecture consisted of shuffling the LUTs within their respective clusters, to align them with the endpoints of the direct connections, after the circuit has been placed. This changes the perspective considerably, because it is no longer a question of which architecture with fast connections is useful, but if its usefulness can be increased by application of CAD tools that are aware of the existence of the fast connections.

Intuitively, one would expect that the latter question is easier to resolve than the former one and this is what inspired the present work. Before diving into details, it is useful to quantify our hopes. Previous work reports a 3% improvement of the average critical path delay of a subset of *VTR* circuits [18, 26]. If the delays of all the connections between the LUTs were reduced to the average of the delays of the direct connections in the proposed architecture, the improvement would rise to about 19%. This is clearly not achievable, but it shows that there is likely a fairly big margin for improvement. A more illustrative example is given in Figure 1. Each cell represents one 60-input cluster of 10 6-LUTs. The architecture also contains a number of direct connections between individual LUTs, shown in Figure 2. The arrows show a subset of connections of the *or1200* circuit placed on this architecture by standard *VPR* [18], oblivious of existence of the direct connections, with a resulting postplacement delay of 13 ns. Moving the LUTs within their respective clusters, in an attempt to improve this delay by aligning the connections depicted by arrows with the direct connections of the architecture, produces the figure on the left. The blue connections are the ones successfully aligned, resulting in a delay of 12.57 ns. Allowing the LUTs to move to the adjacent clusters as well produces the situation on the right, with the delay of 11.67 ns. These numbers were produced by actual optimization, as described in the following sections, and clearly demonstrate the benefits of moving LUTs across clusters.

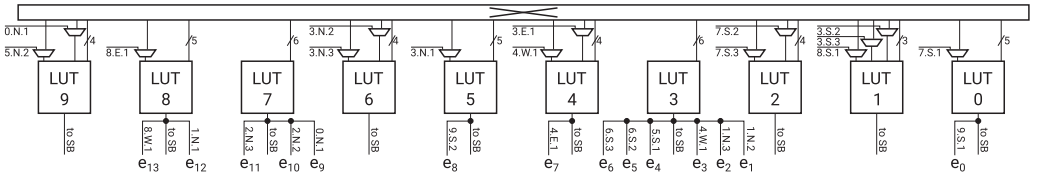


Fig. 2. The target architecture [26]. The other endpoint of each direct connection is labeled as $L.O.D$, where $L \in (0, 9)$ is the index of the LUT in its cluster, $O \in \{N, S, E, W\}$ is the connection orientation with respect to the shown cluster, and D the distance to the other cluster. In general, diagonal connections are also allowed, but are not present in this architecture.

In this article, we investigate the impact of dedicated placement on performance of architectures with direct connections between LUTs. Its original version appeared at the *30th International Conference on Field-Programmable Logic and Applications (FPL)* [27] and introduced a dedicated detailed placement algorithm targeting FPGA architectures with such direct connections, combining iterative solving of ***Linear Programs*** (**LPs**) to select nodes that should be moved from their original positions determined by a general placer oblivious of the direct connections, and ***Integer Linear Programs*** (**ILPs**) to appropriately position them. The main technical novelty that this extended version of the article brings is a greatly simplified and improved main algorithm that orchestrates the construction and solving of different LPs and ILPs. This includes a mechanism to more tightly couple the LPs and the ILPs together. Besides that, we introduce new experiments and present various aspects of optimizing the problem formulation, which have not been previously discussed. Finally, we give a broad overview of higher-level decisions that led to the proposed approach on dedicated placement for architectures with direct connections between LUTs. The choices we took while designing the algorithm are illustrated by numerous examples and compared with possible alternatives.

The rest of the article is organized as follows. In Section 2, we describe the architecture class targeted by the developed algorithm. We describe the high-level aspects of the problem at hand and why we chose to approach it as detailed placement in Section 3. A review of prior work on detailed placement algorithms, for FPGAs and ASICs alike, is given in Section 4. By analyzing the downsides of the existing solutions, we arrive at the one proposed in this article, which is to split the problem into LP-based selection of movable nodes, followed by their ILP-based placement. These two stages of the algorithm are presented in Sections 5 and 6, respectively. The complete algorithm is presented in Section 7, while the various optimizations that we apply to its constituent parts are presented in Section 8. Experimental results are presented in Section 9 and final conclusions drawn in Section 10.

2 TARGET ARCHITECTURES

In this work, we target FPGA architectures that have dedicated fast connections between individual pairs of LUTs. Such architectures have been extensively explored in our previous work [26]. An example architecture—the best one found in that previous study—which we are going to use for the majority of the subsequent experiments is shown in Figure 2. It illustrates a very important feature of this architecture class, without which the placement approach presented in this article would not be practicable. Namely, each of the dedicated connections adds potential for implementing a certain connection of a user circuit in a faster manner, skipping the various multiplexers of the programmable interconnect (see Figure 4), but in doing so, it strictly increases the number of ways in which the user connection can be implemented. This is because the output of each LUT has access to the programmable interconnect and each LUT input pin driven by a direct

Fig. 3. Position of the proposed detailed placement algorithm in a typical FPGA CAD flow.

connection can still be accessed from the programmable interconnect, through the small additional decoupling multiplexers (Figure 2). This somewhat reduces the potential performance gain that the dedicated connections bring, compared to what it would be if they were to directly access the LUT pins, without passing through additional levels of multiplexing. However, it guarantees that every legal implementation of a circuit on the underlying FPGA architecture without the dedicated interconnect will also be legal for the architecture augmented with the direct connections.

3 GENERAL APPROACH

We tackle the problem of placement for FPGA architectures with direct connections between LUTs by constructing a detailed placement algorithm which (1) selects a minimal subset of LUTs that allows the desired critical path delay reduction to be obtained by implementing some of the connections incident to the selected LUTs as direct; then (2) solves an ILP to determine the new positions of the selected LUTs such that the critical path delay is actually improved. Technical details of these two steps are explained in Sections 5 and 6, respectively, while their relation to the existing work on detailed placement algorithms is presented in Section 4.

In this section, we attempt to give a higher level view of the important decisions that formed our approach to the problem. Notably, we answer the question of why a *placement* algorithm is imperative in the first place, why it is necessary to move individual LUTs, and why we opted for a detailed placer which acts upon an already constructed placement oblivious of the existence of the direct connections between LUTs. The position of the proposed algorithm in the overall CAD flow is shown in Figure 3.

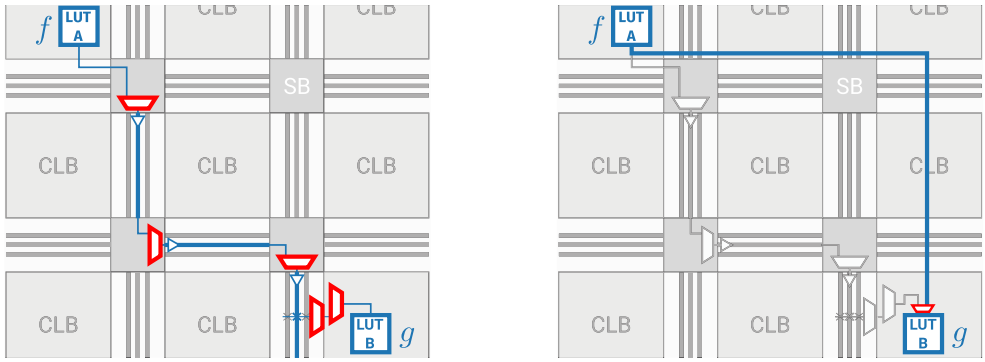
This section also includes a discussion of why arguably the most obvious solution to the placement problem from an academic standpoint—using simulated annealing—is not particularly well suited to the situation when the purpose of doing a placement is targeted implementation of critical connections of the circuit by direct connections of the FPGA.

3.1 Is this not a Routing Problem?

The purpose of this work is to develop adequate CAD support for FPGA architectures equipped with optional direct connections between LUTs, so that these fast dedicated connections may be used to implement the most critical connections of the user's circuit and increase its performance. In a standard FPGA-CAD flow [1] (Figure 3), it is typically the router, which determines the exact path through the programmable interconnect that will implement a particular connection of the circuit, once its endpoints have been fixed during placement. This is the case illustrated in Figure 4(a).

Given that our goal is to determine if some connections of the circuit can be profitably routed by the direct connections of the FPGA, a question could be raised if it is actually a routing and not a placement algorithm that is required. Similarly to the carry chains [19], the direct connections of the FPGA have uniquely defined endpoints. Hence, if a circuit connection (A, B) is to be implemented using a direct connection between points f and g, A (respectively, B) must be aligned with f (respectively g) during placement; otherwise there will be no way of accessing this particular direct connection. This is illustrated in Figure 4(b).

Unlike the case of carry chains, timing criticality of a set of connections between LUTs cannot be readily ascertained in the synthesis phase. At the same time, the number of possible



(a) Implementing a connection using programmable interconnect. Determined during routing.

(b) Implementing a connection using a direct connection of the FPGA. Must be determined during placement.

Fig. 4. Importance of placement for using direct connections. Which wires and multiplexers will implement a connection of the circuit using programmable interconnect (a) can be determined at route time. However, whether it is possible to use a particular direct connection of the FPGA instead is fully determined by the placement of the two endpoint LUTs of the circuit's connection (b).

topologies that the direct connections between LUTs can support vastly surpasses the columnar cascade of the carry chains. For these reasons, strategies such as a priori locking blocks together and moving them in unison during placement [19] would be too constraining for the problem at hand; not only could such a strategy fail to maximize the benefit of using direct connections but it could even damage circuit's performance, by prematurely fixing relative positions of a group of LUTs.

3.2 Necessity of Placing Individual LUTs

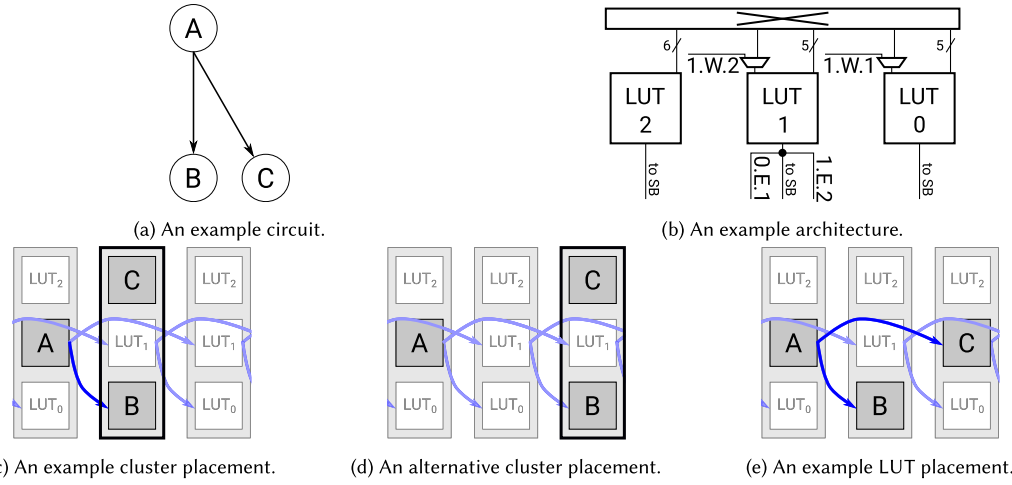
Treating individual LUTs as movable objects during placement can result in superior placement quality [3], and some modern placement algorithms demonstrate that this is practicable at scale [17]. However, a typical FPGA-CAD flow includes a packing stage before the actual placement [1], which groups LUTs together so that each group can be implemented by a logic cluster of the FPGA. Then, these clusters, instead of LUTs, become movable objects in the placement process, greatly reducing the time needed to complete it [4].

To actually use the direct connections of the FPGA, the endpoint LUTs of a circuit's connection must be aligned with the endpoint LUTs of a direct connection. This is often impossible to achieve by moving entire clusters of LUTs, as Figure 5 illustrates. Our goal is to optimize the critical path delay of a circuit, which often requires implementing a small but precisely selected subset of the circuit's connections using the direct connections of the FPGA, and this can be met only by appropriately placing individual LUTs and not clusters.

Hence, for the currently popular cluster sizes of about 10 LUTs, the problem we are facing could involve up to an order of magnitude more movable objects with an order of magnitude more candidate positions than what is usually tackled by a placer that follows a packing stage.

3.3 Global, Detailed, or Combined Placer?

General FPGA placers are very effective in optimizing the critical path delay of a circuit. To illustrate this, we measure the critical path delays of the subset of VTR circuits for which we previously computed the potential average critical path delay improvement due to direct connections (Section 1) at two instants of the VPR's placement process: (1) the very beginning—that is, when all



(a) An example circuit.
(b) An example architecture.
(c) An example cluster placement.
(d) An alternative cluster placement.
(e) An example LUT placement.

Fig. 5. Necessity of placing individual LUTs. Figure 5(a) and (b) respectively shows a portion of a circuit and a simple FPGA architecture on which it is to be implemented. Two alternative cluster placements are shown in Figure 5(c) and (d), both assuming $\{A\}, \{B, C\}$ for the initial packing of LUTs into clusters. Each cluster is represented by a vertical column of three LUTs, designated by the label of the circuit's LUT that it implements, or LUT_{0-2} when left unoccupied. The architecture's direct connections are depicted in blue: dark when used, light when unused. With this initial packing of LUTs, at most one connection of the circuit may be implemented using the direct connections of the FPGA, when only entire clusters are placed. If individual LUTs are able to move independently during placement, however, the outcome in Figure 5(e) can be obtained, with both connections of the circuit implemented as direct and its critical path optimized.

clusters are placed randomly and (2) at the end, when simulated annealing converges. The relative delays are plotted in Figure 6. On average, they improve by almost 45%.

The average additional improvement over that achieved by VPR, obtainable through the implementation of connections between LUTs as direct, lies in the interval between 3%—the value confirmed in our previous work—and 19%—the upper bound presented in Section 1. This means that the final combined improvement over the initial random placement will fall somewhere in the orange strip of Figure 6. Whatever the actually obtained value of improvement due to direct connections may be, it is clear that it will be dwarfed by the improvement initially achieved by the general placer. Hence, it is meaningful to neglect the impact of direct connections on critical path delay, until the critical path delay itself is reduced sufficiently for this additional optimization to become important. This enables gains in runtime, by allowing the initial part of the placement process to be performed at the cluster level.

3.4 Direct Connections at Low Temperature

Let us for the moment stay in the framework of simulated annealing used by VPR. An obvious solution to the problem of individual LUT placement that would partially mitigate the runtime increase would be to perform cluster placement until a certain temperature level, continuing with placement of individual LUTs afterwards, until convergence.

3.4.1 Runtime Surge Persists. This approach has two issues, however. The first one is that it would not really resolve the problem of runtime surge. To illustrate this, we plot in Figure 7 the evolution of the postplacement critical path delay over temperature update iterations during a placement of the *blob_merge* circuit. The orange line indicates the point where the critical path

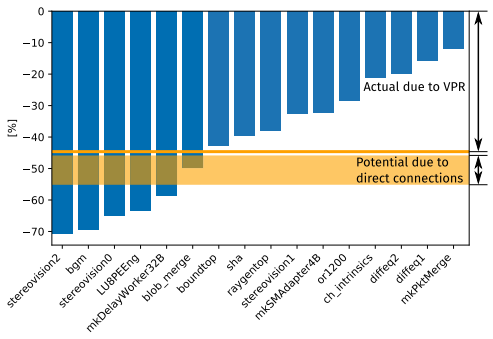


Fig. 6. Critical path delay improvement due to placement. The figure shows the relative critical path delay improvement achieved by VPR at the end of the placement process, compared to the initial random placement of a subset of VTR benchmarks. A highly optimistic estimate of the average potential additional delay reduction due to usage of direct connections of the target FPGA from Figure 2 (Section 1) is superimposed in orange.

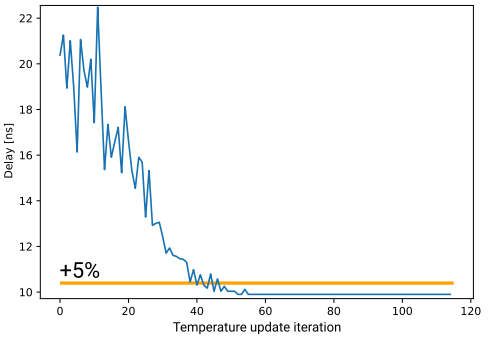


Fig. 7. Postplacement critical path delay evolution as a function of temperature update iteration while placing the *blob_merge* circuit. The orange line depicts the critical path delay, which is 5% worse than the one obtained at the end of VPR's placement process. To achieve this final 5% improvement—roughly comparable to what one could realistically expect from appropriately using direct connections—a significant portion of iterations is used.

delay is 5% larger than the final postplacement critical path delay that VPR was able to achieve. This value was chosen as a reasonable estimate of what impact direct connections could have.

As Figure 7 shows, a large portion of the temperature update iterations is spent on this final 5% delay reduction. Let us optimistically assume for the moment that, with some tuning of the exit criteria, the process would be able to end in 60 iterations. This would mean that about a third of the time would be spent on the final 5% of the delay reduction and this would be the time when placement of individual LUTs would have to be performed if the direct connections are to be used appropriately. Given that a typical number of moves per iteration depends on the number of movable objects with $\Theta(n^{4/3})$ [1], a tenfold increase in the number of movable objects when switching from placing clusters to placing LUTs would increase this third of the runtime almost 22×, increasing the overall time about 8×. Since runtimes of simulated-annealing-based placers are already not competitive by today's standards [29], this would likely be prohibitive in a production setting.

3.4.2 Difficulties in Utilizing Direct Connections. The much more significant issue with this approach is its inaptness for the problem of aligning endpoints of circuit connections with the direct connections of the FPGA. As an illustration, let us take a look at Figure 8. In the top part of the figure, a portion of an FPGA without any direct connections is shown, with a pair of LUTs that eventually need to be connected. For the sake of simplicity, let us assume that the delay of the implemented connection is some function of the Manhattan distance between the clusters in which the two endpoint LUTs reside at the given instant of the placement process (d_M in Figure 8). Let us also assume that each move performed by the placer is a swap of two randomly selected LUTs [1]. Each move will be reflected on the cost function, allowing the optimization to favor moves that improve it, as the temperature of the anneal decreases.

The bottom of Figure 8 depicts the same architecture augmented with one type of direct connections. To appropriately model the impact that this has on the delay of the implemented connection

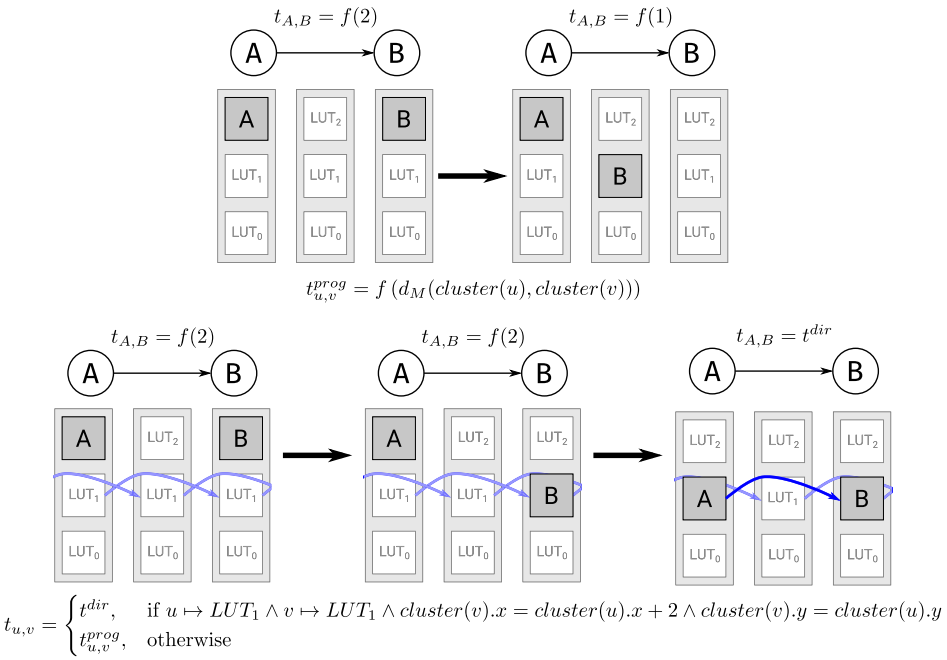


Fig. 8. Difficulty of appropriately utilizing direct connections with simulated-annealing-based placement. To implement a circuit connection as direct, both of its endpoints must be aligned with the endpoints of the direct connection. This is not easy to achieve using the standard moves of swapping pairs of randomly selected objects [1]. Once a connection is implemented as direct, changing it back to programmable would be expensive, which could lead to suboptimal usage of scarce direct connections.

of the circuit, we must introduce a discontinuity in the cost function. Namely, if A and B are positioned at LUT_1, two clusters apart horizontally, the direct connection may be used, resulting in a dramatic drop in delay. In all other cases, the delay is the same as it would have been in the original architecture without direct connections. This is illustrated by the formula at the bottom of Figure 8, with $cluster(u).x$ ($cluster(u).y$) designating the $x-$ ($y-$) coordinate of the cluster in which the node u resides and $u \mapsto LUT_1$ describing the fact that u is placed at LUT_1 of its respective cluster. Let us assume that a move of B was generated, resulting in the placement in the middle of the figure. In order for the LUTs to be properly aligned with the endpoints of the direct connection, a move bringing A to LUT_1 of its current cluster must be generated. If this happens, the sudden drop in cost function will make it unlikely for the connection to be broken again, provided that the temperature is low enough.

This illustrates two important issues: (1) if one endpoint of a circuit connection is aligned with an endpoint of a direct connection of the FPGA (B in the middle placement above), there are no guarantees that the other endpoint will be appropriately moved to complete the implementation of the circuit connection as direct, before the first endpoint moves again; and (2) once a connection is implemented as direct, unless the temperature is high, it is unlikely that it will move back to being programmable, which may prevent another, more critical connection of the circuit from using the particular direct connection of the FPGA.

While both of these issues could perhaps be partially mitigated by clever engineering of the cost function and adoption of directed moves [30], it is evident that an approach better suited to the landscape created by the direct connections would be highly beneficial. Needless to say,

appropriately capturing the discontinuities of the direct connections in other popular frameworks for large-scale placement, such as analytical [22], would be difficult as well.

Adopting a detailed placement approach can resolve most of the above issues. This amounts to starting from a general placement produced by a placer, which is unaware of the existence of the direct connections and then strategically repositioning some of the nodes to improve the critical path delay through appropriate use of the fast direct connections. Another benefit of this approach is that it can be largely oblivious to which general placement algorithm is used to produce the starting placement. This would have been much more difficult if the direct connections were not strictly increasing the flexibility of the interconnect, as discussed in Section 2. Of course, the starting general placement does impact the ability of the detailed placer to improve the critical path delay. A more detailed discussion of this issue is given in Section 9.2.3.

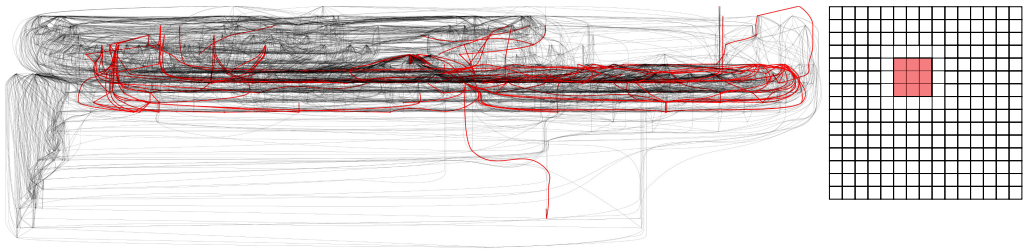
4 PRIOR WORK ON DETAILED PLACERS

There is an abundance of published work introducing detailed placers, both for FPGAs and ASICs [2, 8, 9, 15, 16, 23, 24]. Most of them operate on a sliding-window principle, where a fixed region of the chip is selected for optimization and then iteratively changed by sliding the window that determines it [20]. One basic distinction between the various algorithms is how they optimize inside the window. Some of them rely on heuristics [8, 15], while others use exact optimization methods, such as ILP [2, 16], SAT [24], or SMT [23]. The virtue of heuristics lies in their scalability, which allows them to target larger windows at once, possibly increasing the improvement margin. Exact methods are usually not as scalable, so they are confined to smaller windows, with possibly smaller improvement margin, but are guaranteed to actually meet it. We take a different approach to selecting which portion of the circuit will be optimized and describe it in more details in Section 5. In this section, we analyze the existing approaches to provide motivation for introducing the proposed one. We also express the reasoning that led us to decide on using ILP as the optimization technique.

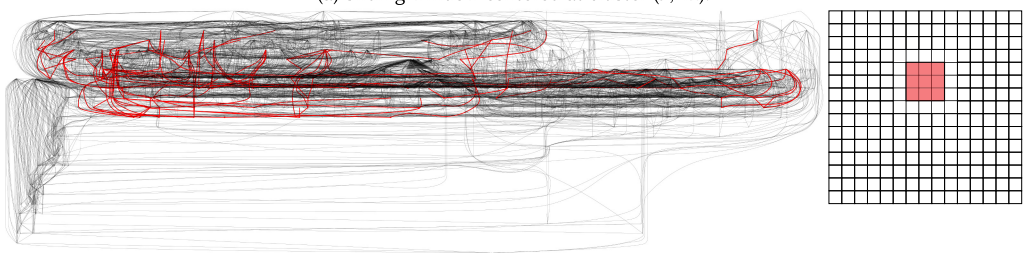
4.1 Movable Node Selection

The sliding-window approach to guiding local optimization is illustrated in Figure 9. One obvious downside of this approach is that it gives little control over which edges of the circuit's timing graph may suffer a change in delay as a result of moving the nodes inside the window. These edges are highlighted in red in Figure 9.

Detailed placers that iteratively optimize edges of the critical path itself, thus avoiding this problem, have also been proposed [9]. Such an approach naturally alleviates the potentially artificial spatial constraints imposed by the sliding windows, as illustrated in Figure 10(a). However, optimizing only one simple path at a time may not be sufficient to actually decrease the critical path delay. Hence, we adopt a related, but much more powerful approach, which selects a number of edges whose timing should be improved, regardless of their location in the timing graph, such that optimizing them maximizes the final critical path delay reduction. It then considers the endpoint nodes of the selected edges movable, regardless of the location of these nodes on the FPGA grid. An example of applying this method, described in Section 5, on the same circuit used to demonstrate the previous two approaches is shown in Figure 10(b). Clearly, the selected edges form a more complex topology in the timing graph than a simple path, while the movable nodes are distributed over a larger set of clusters than in either of the two previously described methods. Another example of spatial distribution of movable nodes obtained by the method proposed in Section 5 is shown in Figure 1. Given that each cell represents a cluster of 10 LUTs, that is, 10 potentially movable objects, a sliding-window approach would likely be limited to not more than a few cells in width and height [20].



(a) Sliding window centered at cluster (9, 12).



(b) Sliding window centered at cluster (10, 12).

Fig. 9. Sliding-window-based movable node selection [20]. The figure shows two different positions of a 3×3 cluster sliding window, on a portion of an FPGA containing a placement of the *blob_merge* circuit. All nodes inside the window are considered movable. Red edges in the timing graph of the circuit, shown on the left, connect different movable nodes. Clearly, the method gives little control over which edges may have their delays improved as a result of the moves.

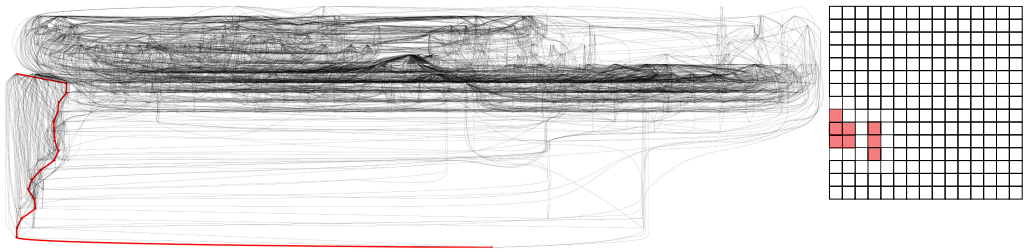
4.2 Movement Freedom

A sliding-window-based selection method typically assumes that each node in the window can move anywhere within the window, as illustrated in Figure 11(a). We take a similar approach by allowing each movable node to move anywhere within a square of half-width W, centered at its original cluster. This is illustrated in Figure 11(b).

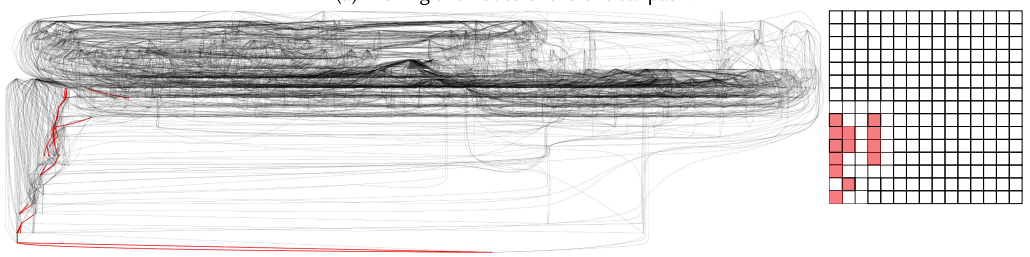
The fact that the sliding-window-based movable node selection assumes all nodes within the window to be movable has one important benefit: the subsequent optimization can guarantee that there are no overlaps between nodes. While the method that we use also guarantees that at the end of the optimization there will be no overlaps between the movable nodes, it leaves a possibility for a movable node to overlap with a stationary one, within its movement region. Such overlaps are only removed in a final postprocessing step, discussed in Section 7.2. The rationale is that if the set of movable nodes is appropriately selected, the benefit of optimally positioning them would most of the time far outweigh the penalty suffered from suboptimally moving some other, less critical nodes standing in their way. If all nodes within the movement regions of the originally selected movable nodes (Figure 11(b)) were to be considered movable at the same time, the problem would quickly become impractically large, unless the number of originally selected movable nodes is itself severely restricted.

4.3 Choice of the Optimization Method

Direct connections are very sparse in a typical architecture considered here, thus requiring a high level of precision in placing the LUTs if the right connections of the circuit are to be aligned with them. As we have seen in Section 3.4, this leaves heuristics with less space for doing a good-enough job at various points in the circuit that would accumulate to a large net improvement than they



(a) Moving the nodes of the critical path.



(b) Selecting a subset of nodes that maximizes delay reduction (Section 5).

Fig. 10. Movable node selection techniques without spatial constraints. Figure 10(a) shows the results of deeming the nodes on a critical path of the circuit movable [9], for the same example previously shown in Figure 9. Figure 10(b) shows results of applying the method introduced in Section 5. This method also imposes no constraints on the spatial distribution of movable nodes, but it enables optimization of subgraphs of the timing graph of arbitrary complexity, maximizing the chance that the critical path delay is actually reduced once the selected nodes are moved. We note again that only a portion of the FPGA is shown in the figure; the movable nodes are in fact not in the corner.

could have had if the direct connections were a more abundant resource. For this reason, instead of attempting to design elaborate heuristics, we choose the exact approach. In particular, we opt for ILP, as it allows straightforward modeling of the timing information. Yet, we formulate the placement problem itself in a way that can be easily converted to SAT.

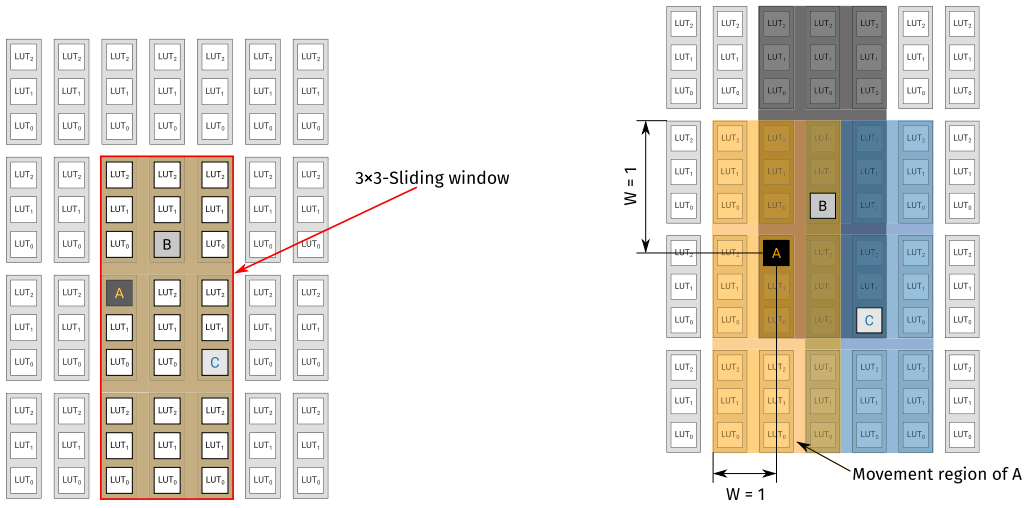
The necessity to precisely position individual LUTs increases the potential number of movable nodes as well as candidate locations for each of them by an order of magnitude when compared to the classical problem of placing entire clusters [9]. However, as we will see shortly, it is exactly the sparsity of the dedicated interconnect that will help us resolve this problem, by enabling more efficient ILP formulations than in the case of general detailed placement [24].

5 THE LP-BASED NODE SELECTOR

The first step in our placement flow is to determine the LUTs that will be moved from their initial positions. This problem is fundamentally linked to determining which connections of the circuit should have their delays reduced so that the reduction of the critical path delay is maximized.

5.1 Which Connections Should be Improved?

Let T be the critical path delay that should be met after the detailed placement. Our goal is to select a minimal number of edges of the circuit's timing graph, which should have their delays improved by their endpoint nodes being aligned with the endpoints of the direct connections of the FPGA, such that the postplacement critical path delay is reduced below T. The rationale is that the fewer edges there are to be improved, the fewer nodes will need to be moved and more likely



(a) Sliding window. (b) Adopted method.

Fig. 11. Movement freedom of movable nodes. Figure 11(a) represents a portion of the FPGA with a sliding window of 3×3 clusters used to select the movable nodes. Inside this window, all nodes are movable and each of them can be placed anywhere in the window. Figure 11(b) shows the approach we take in this work. Only a subset of nodes in each square region of the FPGA is selected for movement. Each of them can be placed at any position inside a square of half-width W, centered at its original cluster.

it is that the placement method of Section 6 will be able to find a solution in the allowed runtime budget.

Let $\tau_{u,v}$ be the initial postplacement delay of the connection $e = (u, v)$, as determined by the general placer. In the example of Figure 12, this is illustrated using a simple model based on Manhattan distance between the initial clusters of u and v, which we already saw in Section 3.4. In practice, any model used by the general placer can be used for obtaining the initial delays. Let us also assign to each edge $e = (u, v)$ a variable $imp_{u,v}$ describing how much its delay should be improved so that the critical path delay bound is met. Then, the final delay of the edge can be expressed as $t_{u,v} = \tau_{u,v} - imp_{u,v}$. In order to reduce the critical path delay below T, we need to find an assignment of *imp*-variables, which will appropriately reduce the delay of each edge. We can achieve this by solving a ***Linear Program*** (**LP**) of the following form, introduced by Hambrusch and Tu [12]:

$$\min \quad \sum_{(u,v)\in E} imp_{u,v}, \tag{1}$$

$$\text{s.t.} \quad ta_u \le T, \qquad \forall u \in V, \tag{2}$$

$$ta_v \ge ta_u + t_{u,v}, \qquad \forall (u, v) \in E, \tag{3}$$

$$0 \le imp_{u,v} \le I_{u,v}, \qquad \forall (u, v) \in E. \tag{4}$$

Here, ta_u represents the arrival time of node u, and constraints (2)–(3) model the timing constraints in the usual manner. Note that to actually minimize the number of edges selected for improvement, the objective should be

$$\min \quad |\{(u, v) \in E : imp_{u,v} > 0\}|, \tag{5}$$

but representing it would require introduction of integral variables, which would render solving the program on the entire timing graph prohibitive.

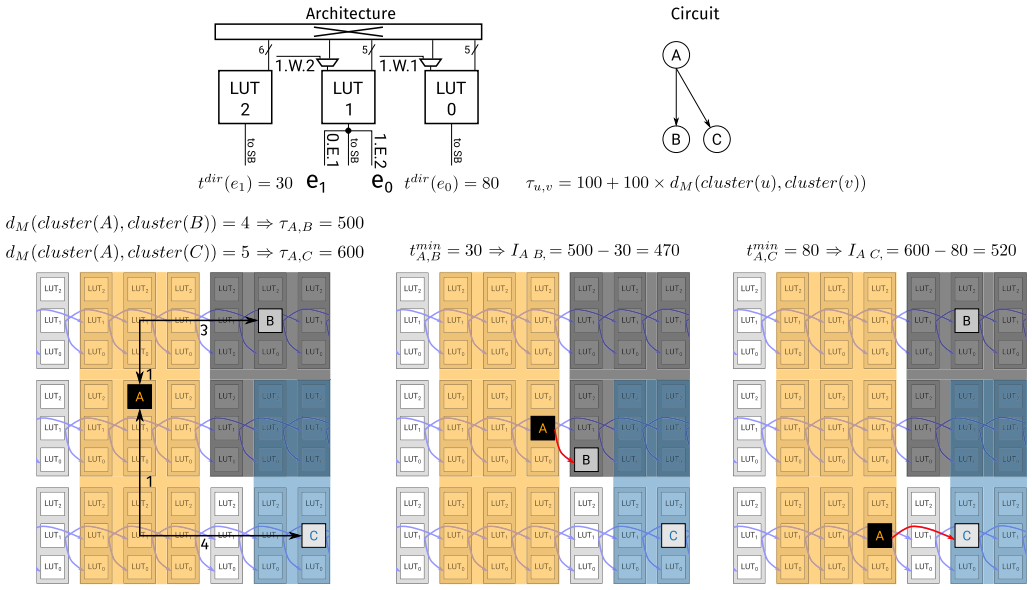


Fig. 12. Illustration of LP variables. Each edge in the timing graph of a circuit is assigned an *imp*-variable determining the amount by which its delay should be improved so that the target critical path delay is met. In order for the assignments of values to the *imp*-variables to reflect the restrictions on node movement, each of the variables is bounded from above by the difference between the initial postplacement delay and the minimum achievable delay, given the movement regions of its endpoint nodes.

The *imp*-variables must be nonnegative, as assigning a negative improvement to an edge with substantial slack could allow increasing the *imp*-variables of many other edges without changing the minimization objective. Representing the possibility of edges being slowed down due to node movement, to which negative *imp*-variables would correspond, is not needed at this level, where we only wish to determine which edges should be improved. This situation changes during actual movement of nodes and hence in Section 6, we model the full range of delay values an edge can attain.

Similarly, the minimum values that can be assigned to the t-variables should reflect the minimum achievable delay for the particular edge, given the movement regions of its endpoint nodes. Hence, the *imp*-variables are bounded from above by the difference between the initial postplacement delay τ and the minimum delay that the edge can realistically achieve (the I-variables in (4)), as illustrated in Figure 12. Since we are constructing a dedicated placer for architectures with direct connections between LUTs, we assume that the starting placement is of high quality and that the delay of each edge can only be improved if it is implemented by a direct connection. For example, if the initial clusters of A and C in Figure 12 were one more cluster apart, horizontally, $I_{A,C}$ would have been zero, for $W = 1$.

5.2 Determining Movable Nodes

To extract the set of movable nodes, which we denote as V_m, from the solution of the above LP, we simply introduce a threshold θ on the minimum delay improvement. Then, the set of edges, which should be improved and are thus candidates for implementation by the direct connections of the FPGA is $E_s = \{(u, v) \in E : imp_{u,v} \geq \theta\}$. To actually implement these edges with direct connections, nodes incident to them must be moved and thus enter V_m.

Controlling $|V_m|$ can be done only indirectly, by specifying the bound on the critical path delay, T. In general, the smaller the value of T, the more edges will have to be improved to meet it and $|V_m|$ will rise accordingly. The fractional nature of the *imp*-variables, however, allows improvement to be spread among more edges than necessary, meaning that a more relaxed T does not necessarily result in smaller $|V_m|$. We comment on this further in Section 8.1, while the explanation of choosing the critical path delay bound is given in Section 7.

6 THE ILP-BASED PLACER

In this section, we discuss various aspects of formulating the ILP that models the movement of the nodes selected by the process described in the previous section.

6.1 Naive ILP Formulation

Each LUT of the FPGA can be described by a triple (x, y, i), where x and y are the coordinates of its cluster and i the index within it. Let $P(u, W)$ be the set of positions within the square of half-width W, centered at the initial cluster of a movable node u (Figure 11). Each LUT inside $P(u, W)$ is a candidate for placing u. To each node $u \in V_m$, we can assign the following set of variables: $x_{u,p} \in \{0, 1\}, \forall p \in P(u, W)$. The variable $x_{u,p}$ is 1 iff node u is placed at position p. The following set of constraints describes a valid placement, where we again note that overlaps with nodes outside V_m are removed in a postprocessing step:

$$\sum_{u \in V_m} x_{u,p} \leq 1, \qquad \forall p, \tag{6}$$

$$\sum_{p \in P(u,W)} x_{u,p} = 1, \qquad \forall u. \tag{7}$$

The first set of constraints prevents overlaps of movable nodes and the second makes sure that each movable node is assigned a unique position. Let $E_\psi = \{(u, v) \in E \backslash E_s : u \in V_m \vee v \in V_m\}$ be the set of edges, which have at least one incident movable node and are thus affected by the placement, but have not been selected for improvement. The delay of each edge in $E_s \cup E_\psi$ is determined by the location of its endpoints:

$$t_{u,v} = \sum_{p_u \in P(u,W), p_v \in P(v,W)} \tau_{p_u,p_v} e_{u,v,p_u,p_v}, \tag{8}$$

$$e_{u,v,p_u,p_v} \in \{0, 1\}, \qquad \forall p_u, p_v, \tag{9}$$

$$e_{u,v,p_u,p_v} \leq x_{u,p_u}, \tag{10}$$

$$e_{u,v,p_u,p_v} \leq x_{v,p_v}, \tag{11}$$

$$e_{u,v,p_u,p_v} + 1 \geq x_{u,p_u} + x_{v,p_v}. \tag{12}$$

Here, τ_{p_u,p_v} are constants corresponding to the least delay of the connection with its endpoints placed at p_u and p_v, respectively. Constraints (10)–(12) are merely one way of linearizing a product of two binary variables [31]. With the timing graph modeled as in the selection LP (constraints (2)–(3)), we have a complete formulation of the placement ILP. If (u, v) is an edge from E_ψ, some variables may become constants, simplifying the corresponding constraints.

This formulation is generic and can be used to place circuits on architectures with or without direct connections. It is also rather intuitive and well known in literature. In a very similar form, it has been used for SAT [24] and SMT-based [23] timing-driven FPGA placement, as well as for ILP-based wirelength-driven ASIC placement [2].

The problem lies in the large number of position variables and quadratic length of delay assignment constraints (8) with respect to that number. Fixing W to 3—the length of the longest connection in the architecture of Figure 2—leads to $7 \times 7 = 49$ clusters and 490 potential positions for each movable node. Any edge can have up to $490^2 > 240{,}000$ addends in the delay assignment constraint (8). This is clearly an issue and we address it in the next section.

6.2 Exploiting the Sparsity of Dedicated Interconnect

Direct connections are sparse. If they were not, the width and count of the additional multiplexers and the increased loading of LUT outputs would greatly reduce their speed, slowing down the general routing as well. In the architecture of Figure 2, there are only 14 connections originating in one cluster.

Let $E_d(u, v) \subseteq P(u, W) \times P(v, W)$ be the set of direct connections that can implement (cover) the edge $(u, v) \in E_s$. The exact position of a LUT matters only when an edge $e \in E_s$ incident to it is being covered. In all other cases, knowing its cluster is sufficient, since placement-time delay models of general placers rarely differentiate between different exact positions of LUTs [21], which may be subject to change during routing, to reduce congestion [14].

Instead of listing all exact positions for all movable LUTs and inspecting which edges are covered, we can list the edge covering possibilities and derive the LUT positions from them. Let $C(u, W)$ be the set of clusters within the square of half-width W, centered at the initial cluster of a movable node u. A binary variable $x_{u,c}$ indicates that u is placed in cluster $c \in C(u, W)$. We can now model the edge delays as follows:

$$t_{u,v} = \sum_{(p_u,p_v)\in E_d(u,v)} \tau_{p_u,p_v} e_{u,v,p_u,p_v} + \overline{c}_{u,v} \sum_{c_u \in C(u,W), c_v \in C(v,W)} \tau_{c_u,c_v} e_{u,v,c_u,c_v}, \tag{13}$$

$$\overline{c}_{u,v} = 1 - \sum_{(p_u,p_v)\in E_d(u,v)} e_{u,v,p_u,p_v}. \tag{14}$$

If there is a direct connection that covers the edge (u, v) in the current subproblem, the appropriate τ from the first sum will determine the delay because the *coverage indicator* $\overline{c}_{u,v}$ will be 0. In all other cases, the indicator will be 1, causing the second sum to determine the delay. The τ constants in that sum are the delays between the two appropriate clusters, as modeled by the general placer. The e_{u,v,c_u,c_v} variables are products of the cluster position variables, linearized using constraints similar to constraints (10)–(12). Another level of linearlization is applied to products with the coverage indicators. Note that constraints (13–14) are merely an ILP-encoding of a generalized version of the delay model used in Figure 8. While the sparsity of direct connections created problems for convergence of common formulations of simulated-annealing-based placement, it allows for compact modeling of the problem as an ILP.

The maximum length of the delay assignment constraint for $W = 3$ and the architecture of Figure 2 is now $49 \times 14 + 49^2 \ll 490^2$. The first addend corresponds to at most 49 ways to choose the starting cluster for the direct connection and 14 ways to choose the exact direct connection leaving it, while the second addend amounts to the number of cluster pairs that determine the edge delay if it is implemented as programmable. Similarly, the number of exact position variables for each node $u \in V_m$ is reduced from $|P(u, W)|$ to $|\cup_{\{u,v\}\in E_s} \{p_u : \{p_u, p_v\} \in E_d(u, v)\}|$; that is, to only those positions implied by covering of some edge incident to the given node.

Our model is still not complete. The linearizations (10)–(12) are of course kept, and they cause any e_{u,v,p_u,p_v} to imply exact positions of u and v. What is missing is that each exact position $x_{u,p=(x,y,i)}$ implies the corresponding cluster position $x_{u,c=(x,y)}$. The following set of constraints

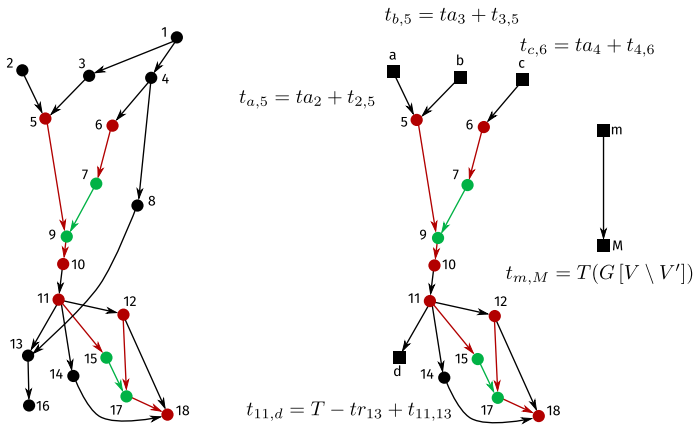


Fig. 13. Illustration of timing graph compaction through simplification of fixed-delay subgraphs. Green edges are selected for improvement ($\in E_s$) and their endpoint nodes are movable ($\in V_m$). Red edges are affected by the movement of nodes, but were not selected for improvement ($\in E_\psi$). The remaining edges, shown in black, are unaffected by node movement. The graph on the right shows how the starting graph on the left can be simplified, by excluding some of its fixed-delay subgraphs and replacing them with additional constraining nodes, depicted as squares. Edge (m, M) is used to store the delay of the longest path in the part of the timing graph not represented in the compacted version ($G[V \setminus V']$).

achieves that:

$$x_{u,c=(x,y)} \geq \sum_i x_{u,p=(x,y,i)}, \quad \forall u \in V_m, c \in C(u, W). \tag{15}$$

Finally, we need to make sure that each node is assigned exactly one cluster, using constraints similar to (7).

6.3 Delay-Based Model Compaction

Further compaction of the model can be achieved by excluding the irrelevant portions of the timing graph. Edges could be irrelevant either because their delay does not change during placement and they do not carry any information relevant for computing of arrival and required times of the nodes affected by placement, or because they can never become critical, under any feasible assignment of delays to the remaining edges.

6.3.1 Simplification of Fixed-Delay Subgraphs. An example of simplification of the fixed-delay subgraphs is shown in Figure 13, where the graphs are constructed merely for the purpose of illustration and have no further meaning. Edges selected for improvement (E_s), and the nodes incident to them (V_m) are shown in green, while other edges whose delay may change as nodes move (E_ψ) as well as the stationary nodes incident to them are shown in red. For sufficient timing information to be represented, all colored nodes from the graph in the figure are kept in its compacted version. Node 14 must also be kept, as it informs the solver that there is an alternative path between nodes 11 and 18, which may at some point become the determining factor for the arrival time of node 18. If that happens, no more effort should be spent on trying to further decrease this arrival time, by decreasing the delay of edge (15, 17). Instead, an attempt to optimize other edges should be made.

Let us now take a look at node 13. Since none of the movable nodes is reachable from it, it cannot affect the arrival time of any of them. It can, however, affect their required time and it is important

to represent this correctly in the compacted graph, as otherwise, critical subgraphs may be left out of optimization. To do this, it is not necessary to include any nodes from the transitive fanout of 13, because all the delays in it are unaffected by repositioning of the movable nodes. Instead, we can introduce an additional node, labeled as d in Figure 13, and connect it to 11 by an edge with a delay equal to the delay of the edge between 11 and 13 (fixed throughout the placement as neither 11 nor 13 are movable) increased by the difference between the original critical path delay and the required time of 13. This increase represents the total delay of the downstream portion of the graph unaffected by the movement of nodes.

A similar approach can be applied to, e.g., node 3, but with its arrival time being relevant to represent the fixed delay of the upstream portion of the graph. This can be generalized as follows:

(1) Find all nodes $V' \in V$, which can both reach and are reachable from nodes in V_m.
(2) For each v added to V' in step 1 and each $u : (u, v) \in E$, if u was not added to V' in step 1, add a new node n to V' and connect v to it by an edge with delay $t_{n,v} = t_{u,v} + ta_u$, where ta_u is the arrival time of u.
(3) For each u added to V' in step 1 and each $v : (u, v) \in E$, if v was not added to V' in step 1, add a new node n to V' and connect it to u by an edge with delay $t_{u,n} = t_{u,v} + T - tr_v$, where tr_v is the required time of v and T the original critical path delay.

Finally, we need to add another edge to the compacted graph which will represent the critical path delay of the portion of the original graph which was excluded from its compacted version. Otherwise, it may appear to the solver that the critical path delay can be reduced more than what is actually possible. This edge is labeled as (m, M) in Figure 13. Although the toy example of Figure 13 fails to illustrate it, given that $|E_s|$ typically does not exceed a few tens or perhaps a few hundred, the above technique can provide great reduction in the size of the timing graph.

In principle, it would suffice to include additional nodes only for the most constraining parents/children, in steps 2 and 3, but savings from this would not be very high, so we do not do that in our implementation. Similarly, the path $11 \rightarrow 14 \rightarrow 18$ could be replaced by a single edge connecting nodes 11 and 18 and carrying the delay of the entire path. While this approach could result in significant further compaction, generalizing it to subgraphs with more complex connectivity is not as straightforward, so we chose to stay with the simple procedure described above.

6.3.2 Filtering Slow Edges. A lower bound on critical path delay achievable by solving the placement ILP can be easily computed from the solution of the selection LP (Section 5). The maximum delay of each edge can be easily computed by considering the allowed positions of its endpoint nodes. We annotate the timing graph with these maximum delays and compute the slacks of all edges, given the lower bound on the critical path delay. All edges which have a positive slack are guaranteed not to be critical for any valid solution of the ILP and can thus be safely removed from the timing graph.

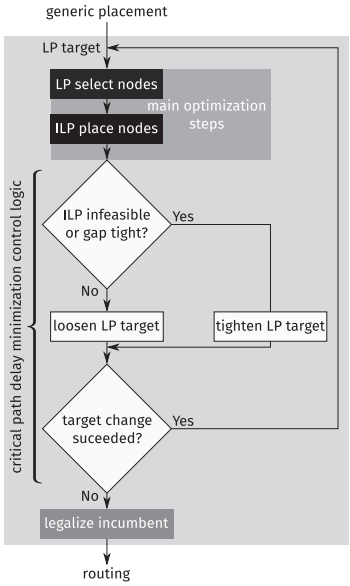
6.3.3 Filtering Slow Position-Pairs. Another very straightforward compaction method that we use is to compute the slacks of all edges on a timing graph annotated with the minimum achievable delay for each edge and then remove all e_{u,v,p_u,p_v} (e_{u,v,c_u,c_v}) position pairs for which τ_{p_u,p_v} (τ_{c_u,c_v}) exceeds the minimum delay of (u, v) increased by its slack.

7 THE COMPLETE ALGORITHM

In this section, we combine together the two stages of the placement flow, presented in Sections 5 and 6, and introduce a postprocessing step that removes overlaps between the movable and the stationary nodes.

ALGORITHM 1: Detailed Placer

1: $T_{incumbent} \leftarrow T_{start}$
2: $\text{gap}_{\text{incumbent}} \leftarrow \infty$
3: $T_{low} \leftarrow$ get_lower_bound()
4: $T_{high} \leftarrow T_{start}$
5: **while** $T_{high} - T_{low} > T_{\delta}^{min}$ **do**
6: $T_{target} \leftarrow (T_{low} + T_{high})/2$
7: movable, $T_{LP} \leftarrow$ lp_select_movable_nodes(T_{target})
8: status, placement, T_{ILP}, gap $\leftarrow$ ilp_place_nodes(movable, $T_{LP} \leq T \leq T_{incumbent}$, $T_{lb} = T_{LP} - 2T_{\delta}^{min}$)
9: **if** $(T_{ILP} < T_{incumbent}) \vee ((T_{ILP} = T_{incumbent}) \wedge (\text{gap} < \text{gap}_{\text{incumbent}}))$
10: update_incumbent(T_{ILP}, gap, placement)
11: **if** (gap < 1) $\vee$ (status = infeas)
12: $T_{high} \leftarrow T_{target}$
13: **else**
14: $T_{low} \leftarrow T_{target}$
15: legalize(incumbent)



7.1 Composing the Detailed Placer

We use a simple binary search to minimize the target critical path delay specified when selecting the movable nodes (Section 5). The lower bound of the search range is determined by performing a timing analysis on the timing graph of the circuit with the delay of each edge replaced by the minimum it can attain, given the movement regions of its endpoint nodes. This is represented by line 3 of Algorithm 1. The upper bound is set to the critical path delay of the starting placement produced by the general placer, assuming that all connections are implemented as programmable. The search stops when the two bounds differ less than T_{δ}^{min}, which we set to 30 ps in the subsequent experiments.

The main loop consists of solving an improvement LP with the current target bound, to obtain the set of movable nodes (line 7; see Section 5) followed by solving the related placement ILP to actually position the movable nodes (line 8; see Section 6). The solver is instructed to minimize the critical path delay, T_{ILP}, as much as possible, given the allowed runtime budget. The solution is constrained to have a critical path delay at most as large as the smallest one encountered so far, $T_{incumbent}$. The lower bound on achievable critical path delay used for pruning the edges which can never become critical (see Section 6.3) is set to the critical path delay obtained after thresholding the LP solution, T_{LP} (see Section 5), reduced slightly to leave some margin for round-off.

If the obtained critical path delay, T_{ILP}, is lower than the current best, $T_{incumbent}$, or they are equal, but T_{ILP} is proven by the solver to be closer to the optimum for the current set of movable nodes, the incumbent solution is updated (line 10). When the solution is proven to be within 1% of the optimum, the algorithm considers that the current placement problem was successfully solved and that an attempt to achieve more critical path delay reduction should be made. Hence, the binary search range is constricted from the right, on line 12. The same happens when the problem is proven infeasible. This means that the incumbent solution cannot be improved with the current selection of movable nodes. To resolve this, the set of movable nodes should likely be increased, which is achieved by reducing the target critical path delay so that more edges need to be improved

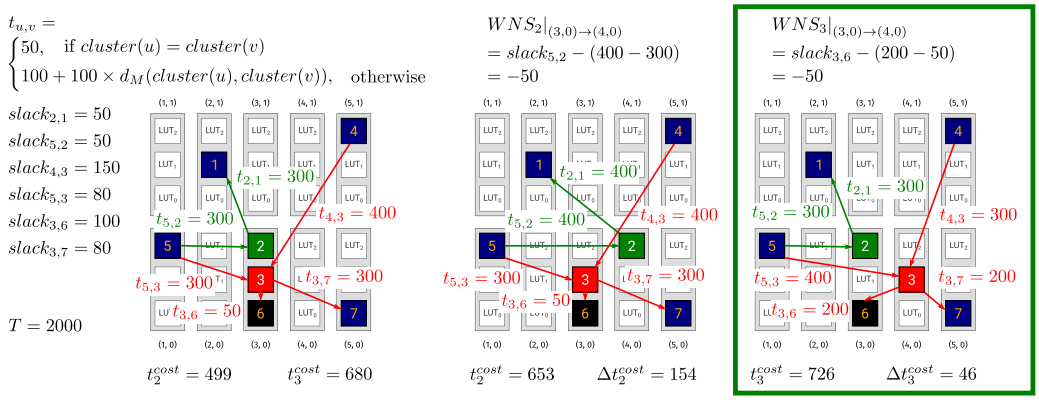


Fig. 14. Determining movement cost between a pair of clusters during legalization. The figure illustrates the cost of moving nodes 2 and 3 from the cluster (3, 0) to the cluster (4, 0). The post move worst negative slack is the same for both nodes. Hence, the timing cost difference between the starting situation and after the move is used to break the tie. The difference is lower for node 3; hence, it will be the moved by the legalizer.

to meet it. If the solver failed to provide a definitive answer in the allowed runtime budget (even if it did find a new incumbent solution, but failed to prove it optimal) the problem is deemed too difficult to be solved in the allowed runtime budget and the binary search range is constricted from the left (line 14) in a hope that a looser target critical path delay would result in an easier placement problem.

Once the binary search converges, any overlaps which may have occurred between the movable and the stationary nodes must be removed. This is done by the postprocessing step on line 15, discussed in more detail in the next section.

7.2 Legalizer

For removing overlaps between the movable and the stationary nodes, we adapt the algorithm of Darav et al. [7]. Since our main goal is to optimize performance of the processed circuit, the legalizer must be timing-aware itself, not to undo the critical path reduction achieved by the detailed placer, unless that is necessary for achieving a legal placement.

7.2.1 Pricing LUT Movement. When faced with a decision about which LUT should be moved between clusters A and B, as a primary factor, we use the postmove worst negative slack of all connections incident to the LUT we are attempting to move, and choose the LUT for which the magnitude of this slack is the smallest. This is illustrated in Figure 14. In case of ties, we compute the difference in the timing cost of the LUT before and after the move and pick the LUT with the smallest increase in this cost. The timing cost is adopted from VPR's timing-driven placer [21]:

$$crit_{u,v} = 1 - \frac{slack_{u,v}}{T}, \tag{16}$$

$$t_u^{cost} = \sum_{(u,v)\in E} t_{u,v} \times crit_{u,v}^{\alpha} + \sum_{(p,u)\in E} t_{p,u} \times crit_{p,u}^{\alpha}, \qquad \alpha \in \mathbb{R}^+. \tag{17}$$

Here, T designates the current critical path delay. For the selectivity parameter α, we use 8 in the subsequent experiments. We first run the legalizer without performing any slack updates, relying on the values obtained in the first static timing analysis after the detailed placement converged.

If the legalized critical path delay exceeds the one computed by the ILP solver during detailed placement by more than 100 ps, we rerun the legalizer, committing each move to the timing graph as soon as it is decided and updating the slacks accordingly, to prevent suboptimal local move decisions from having a large cumulative effect.

7.2.2 Bounding Overlaps. Success of targeted application of the placement ILP to a limited set of movable nodes spread over wide regions of the starting placement relies on the observation that in many circuits, the gain from appropriately positioning a small number of critical nodes far exceeds the loss created by suboptimally moving other, less critical nodes that stand in their way. However, this only holds if not too many nodes need to be moved from their original positions during postprocessing. Otherwise, the timing information that was presented to the ILP solver, which assumes that all stationary nodes will retain their original positions, may be too significantly disturbed in the overlap removal process, leading to an inevitable loss in the achieved delay reduction. To prevent this from happening, we need to control the amount of overlap occurring in each cluster. This is easily achieved with the help of the following constraints:

$$\sum_{u \in V_m} x_{u,c} \leq \omega_c, \qquad \forall c. \tag{18}$$

The constant $\omega_c \in \mathbb{N}$ sets a limit on the number of movable nodes, which can be placed in cluster $c = (x, y)$. We determine it as follows:

$$s = |\{V \setminus V_m\} \cap c|, \tag{19}$$

$$\omega_c = N - s + \min(s, \delta). \tag{20}$$

Constant s holds the number of stationary nodes in cluster c, while N is the cluster size in the underlying FPGA architecture. The allowed overflow is determined by the parameter δ, which we set at 2 in the subsequent experiments, as we observed that this value does not limit achieved post-placement delay reduction and rarely leads to an increase in this delay after legalization. The $N - s$ part of Equation (20) specifies that all positions, which were originally unoccupied, or were occupied by the movable nodes, can be filled by the movable nodes. The $\min(s, \delta)$ part guarantees that overlaps within the cluster can be resolved by removal of stationary nodes in the postprocessing step. As mentioned before, overlaps between movable nodes, which are assigned exact positions are impossible due to constraints (6). However, some of the movable nodes may not be assigned an exact position but merely a cluster, in which case they could overlap with other movable nodes. Since the movable nodes are necessarily critical (albeit for the LP target critical path delay), as otherwise the minimum improvement solution of the improvement LP (Section 5) would not affect them, they should be positioned with care. Allowing them to overlap with other movable nodes, by letting the amount of overlap exceed the number of stationary nodes in the cluster, would leave their positioning to the fast but suboptimal legalizer.

8 OPTIMIZATION

In the previous sections, we described the basic form of the proposed algorithm. It first solves an LP to determine which edges in the circuit's timing graph should have their delays reduced by moving their endpoint nodes to align them with the endpoint LUTs of the direct connections available in the FPGA architecture. Then it solves a related ILP to perform the actual placement. We focused on simplifying the ILP model to the extent that would allow for its solving in reasonable time. Until now, the solution of the improvement LP determined the formulation of the placement ILP, but we have done little to formulate the improvement LP itself in such a manner that its solution is more likely to produce a feasible ILP. In this section we focus on extending the formulation of the

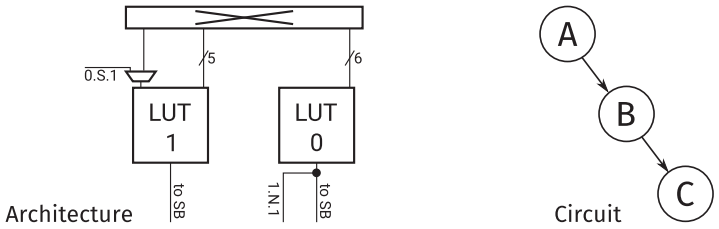


Fig. 15. Pitfalls of the basic selection LP. An architecture is shown on the left, and a piece of a circuit on the right. A solution selecting both circuit connections for improvement is valid, but not supported by the architecture in which each LUT has only one incident direct connection.

improvement LP to more tightly couple it to the placement ILP. We also extend the formulation of the placement ILP itself, so as to make it easier to solve.

8.1 Specialization of the Improvement LP to the Architecture

Keeping the set of edges selected for improvement (the number of movable nodes, $|V_m|$) reasonable is necessary for the related placement ILP to be solved in a reasonable amount of time. That was the reason for which the formulation of the improvement LP presented in Section 5 minimized the total delay improvement. Let us for the moment disregard the aforementioned fact that minimizing the total improvement does not necessarily translate to smaller $|V_m|$, nor does this necessarily translate to an easier-to-solve ILP. Let us assume instead that the obtained ILP can be solved in the allowed amount of time. The ILP can still be infeasible, for various different reasons and we would like to predict and ideally prevent this already at the LP level. For example, simultaneous improvement of two different connections might imply two nodes being placed at the same position or one node being placed at two distinct positions at once.

Aside from the initial placement of the circuit and the allowed movement regions, the FPGA architecture strongly influences feasibility of the placement ILPs constructed from the solutions of the improvement LPs. Figure 15 shows a simple architecture and a piece of a circuit. With the current LP formulation, there is nothing that would prevent the solution from including both edges of the circuit, although it is clear that the architecture will not be able to improve both of them. We cannot enforce exclusivity in choosing between these two edges without introducing integer variables, but we can use additional constraints to increase the chance of obtaining solutions that the architecture can support. To begin with, we can introduce a bound on the sum of the improvements of the two edges, equal to the maximum of the two individual bounds. This still does not prevent the solution from including both connections, but covering only one of them during the placement process will suffice for this short path to meet what is expected of it in terms of overall delay reduction. For that reason, we introduce pairwise improvement bounds, for each pair of edges sharing a common node. In general, this will not be the maximum of the two individual bounds, but the largest total improvement achievable within the movement regions of the three incident nodes. To further improve feasibility, we include bounds on the total improvement of the incoming, the outgoing, and all the edges incident to each individual node.

8.2 Solving Successive ILPs

During the binary search for the smallest achievable critical path delay, the placer may have to solve many ILPs. However, since all of them are describing a detailed placement problem of the same circuit, on the same FPGA architecture, and with the same starting general placement, they will inevitably be related. We can use this fact to make the solution of the ILPs simpler, as well as improve the chances that they are feasible, by slightly adjusting the LP formulation.

8.2.1 Enforcing ILP Solution Overlaps. During experimentation, we observed that the number of covered edges rarely substantially decreases between two consecutive incumbent solutions. Moreover, in most cases that we have inspected, there was a substantial overlap between two consecutive incumbent solutions in terms of which edges were covered in them. We can use this fact to help the solver find feasible solutions more easily. Let E_s^i be the set of edges selected for improvement in the problem that led to the incumbent solution and $E_c^i \subseteq E_s^i$ the set of edges that are actually covered in the incumbent solution. We denote the set of edges selected for improvement in the current problem as E_s, like before. Similarly, E_c denotes the set of edges covered in a valid solution of the current problem. Then we add the following two constraints to the ILP:

$$|E_c| \geq \eta \left|E_c^i\right| \times \frac{|E_s|}{\max\left(|E_s|, \left|E_s^i\right|\right)}, \quad \eta \in (0,1), \tag{21}$$

$$\left|E_c \cap E_c^i\right| \geq \zeta \left|E_c^i\right| \times \frac{\left|E_s \cap E_s^i\right|}{\left|E_s^i\right|}, \quad \zeta \in (0,1). \tag{22}$$

The first constraint specifies the minimum number of covered edges, with respect to the number of covered edges in the incumbent solution, appropriately scaled down if the number of edges selected for improvement is smaller than in the problem which produced the incumbent solution. The second constraint specifies the minimum amount of overlap between the set of covered edges in the incumbent solution and the solution of the current problem. Note that $|E_c^i|$ and $|E_s^i|$ are constants obtained by inspecting the incumbent solution, while $|E_s|$ and $|E_s \cap E_s^i|$ are also constants obtained from the solution of the improvement LP. Hence, the right-hand side of both constraints is constant. The left-hand side is a single sum, encoded using the coverage indicators $c_{u,v} = 1 - \overline{c}_{u,v}$, where $\overline{c}_{u,v}$ was introduced in Section 6.2. In the subsequent experiments, parameters η and ζ are set to empirically determined values of 0.7 and 0.3, respectively. In general, there is no requirement that $\eta + \zeta = 1$.

8.2.2 Using ILP Solutions to Improve LP Formulation. Anticipating which edges selected for improvement will not actually improve due to conflicts with improvement of other selected edges is difficult at the LP level. On the other hand, whenever the ILP returns a feasible solution, it is possible to inspect it for selected edges, which did not actually get covered and discourage their repeated selection in the LP solution. To do that, we extend the objective of the LP to

$$\min \sum_{(u,v)\in E} \alpha_{u,v} imp_{u,v}, \quad \alpha_{u,v} \in \mathbb{R}^+ \cup \{0\}. \tag{23}$$

We temporarily set $\alpha_{u,v}$ to 0 for all edges covered by the incumbent solution (E_c^i), to prevent unnecessary restriction of possible overlap between it and the solution of the next problem (Section 8.2.1). For the remaining edges of the timing graph, the coefficients are initially set to 1. Each time a solution to the ILP is found, for every edge $(u, v) \in E_s$, we multiply $\alpha_{u,v}$ by 0.9 if it is covered by the solution and 1.1 if it is not.

In this manner, we encourage the solution of the improvement LPs to include edges that were repeatedly shown to be successfully coverable and discourage it from selecting the ones, which were repeatedly shown to be difficult to cover. Note that since we do not modify the bound on achievable improvement of any edge, but merely the way in which the invested improvement enters the objective, if the target critical path delay is small enough to require selection of edges which over time grew expensive, there is nothing that would prevent this from happening. Needless to say, the exact values of the scaling parameters can be changed as required.

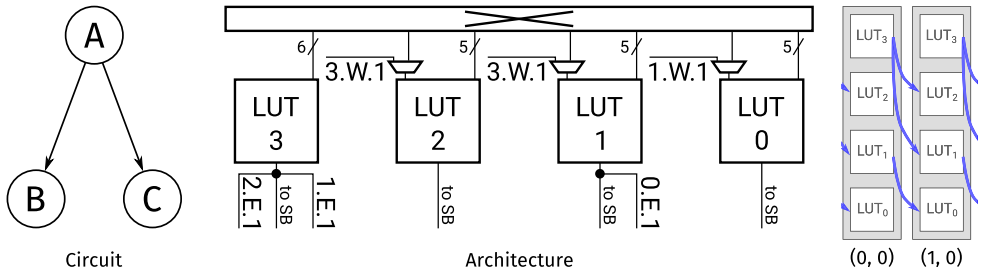


Fig. 16. Example used to illustrate formulation tightening through node degree matching.

8.3 ILP Formulation Tightening

In Section 6.2, we have exploited a specific characteristic of FPGA architectures with direct connections to encode the placement ILP much more efficiently. We can exploit characteristics of a particular architecture further, but this time to produce additional constraints that will tighten the formulation of the ILP.

The approach that we use is to classify LUT positions based on the number of incoming and outgoing direct connections they have. Then, we introduce constraints, which keep count of the incoming and outgoing direct connections that each movable node has under the current assignment of values to the variables. Finally, we use these counts in implications that help exclude the positions, which cannot accommodate the required number of direct connections. To illustrate this, let us take a look at the example in Figure 16.

Let us for the sake of simplicity neglect the timing information, assume that all nodes of the circuit are movable and that the objective is to maximize the number of covered edges. This can be described with the following ILP:

$$\max \quad c_{A,B} + c_{A,C},$$

$$c_{A,B} = e_{A,B,(0,0,3),(1,0,2)} + e_{A,B,(0,0,3),(1,0,1)} + e_{A,B,(0,0,1),(1,0,0)},$$
$$c_{A,C} = e_{A,C,(0,0,3),(1,0,2)} + e_{A,C,(0,0,3),(1,0,1)} + e_{A,C,(0,0,1),(1,0,0)},$$

$$e_{A,B,(0,0,3),(1,0,2)} \leq x_{A,(0,0,3)}, x_{B,(1,0,2)}; \quad e_{A,B,(0,0,3),(1,0,2)} + 1 \geq x_{A,(0,0,3)} + x_{B,(1,0,2)},$$
$$e_{A,B,(0,0,3),(1,0,1)} \leq x_{A,(0,0,3)}, x_{B,(1,0,1)}; \quad e_{A,B,(0,0,3),(1,0,1)} + 1 \geq x_{A,(0,0,3)} + x_{B,(1,0,1)},$$
$$e_{A,B,(0,0,1),(1,0,0)} \leq x_{A,(0,0,1)}, x_{B,(1,0,0)}; \quad e_{A,B,(0,0,1),(1,0,0)} + 1 \geq x_{A,(0,0,1)} + x_{B,(1,0,0)},$$
$$e_{A,C,(0,0,3),(1,0,2)} \leq x_{A,(0,0,3)}, x_{C,(1,0,2)}; \quad e_{A,C,(0,0,3),(1,0,2)} + 1 \geq x_{A,(0,0,3)} + x_{C,(1,0,2)},$$
$$e_{A,C,(0,0,3),(1,0,1)} \leq x_{A,(0,0,3)}, x_{C,(1,0,1)}; \quad e_{A,C,(0,0,3),(1,0,1)} + 1 \geq x_{A,(0,0,3)} + x_{C,(1,0,1)},$$
$$e_{A,C,(0,0,1),(1,0,0)} \leq x_{A,(0,0,1)}, x_{C,(1,0,0)}; \quad e_{A,C,(0,0,1),(1,0,0)} + 1 \geq x_{A,(0,0,1)} + x_{C,(1,0,0)},$$

$$x_{A,(0,0,3)} + x_{A,(0,0,2)} + x_{A,(0,0,1)} + x_{A,(0,0,0)} + x_{A,(1,0,3)} + x_{A,(1,0,2)} + x_{A,(1,0,1)} + x_{A,(1,0,0)} = 1,$$
$$x_{B,(0,0,3)} + x_{B,(0,0,2)} + x_{B,(0,0,1)} + x_{B,(0,0,0)} + x_{B,(1,0,3)} + x_{B,(1,0,2)} + x_{B,(1,0,1)} + x_{B,(1,0,0)} = 1,$$
$$x_{C,(0,0,3)} + x_{C,(0,0,2)} + x_{C,(0,0,1)} + x_{C,(0,0,0)} + x_{C,(1,0,3)} + x_{C,(1,0,2)} + x_{C,(1,0,1)} + x_{C,(1,0,0)} = 1,$$

$$x_{A,(0,0,3)} + x_{B,(0,0,3)} + x_{C,(0,0,3)} \leq 1,$$
$$x_{A,(0,0,2)} + x_{B,(0,0,2)} + x_{C,(0,0,2)} \leq 1,$$
$$x_{A,(0,0,1)} + x_{B,(0,0,1)} + x_{C,(0,0,1)} \leq 1,$$
$$x_{A,(0,0,0)} + x_{B,(0,0,0)} + x_{C,(0,0,0)} \leq 1,$$
$$x_{A,(1,0,3)} + x_{B,(1,0,3)} + x_{C,(1,0,3)} \leq 1,$$
$$x_{A,(1,0,2)} + x_{B,(1,0,2)} + x_{C,(1,0,2)} \leq 1,$$
$$x_{A,(1,0,1)} + x_{B,(1,0,1)} + x_{C,(1,0,1)} \leq 1,$$
$$x_{A,(1,0,0)} + x_{B,(1,0,0)} + x_{C,(1,0,0)} \leq 1.$$

While solving the continuous relaxation of the above program the solver could yield the following fractional solution

$$e_{A,B,(0,0,3),(1,0,2)} = e_{A,C,(0,0,3),(1,0,2)} = e_{A,B,(0,0,1),(1,0,0)} = e_{A,C,(0,0,1),(1,0,0)} = 0.5,$$
$$c_{A,B} = c_{A,C} = 1,$$
$$x_{A,(0,0,3)} = x_{A,(0,0,1)} = 0.5,$$
$$x_{B,(1,0,2)} = x_{C,(1,0,2)} = x_{B,(1,0,0)} = x_{C,(1,0,0)} = 0.5,$$

with all other variables at 0. Of course, this solution is not feasible for the ILP itself, since it implies that all nodes partially occupy two positions each. It would be good to make this solution infeasible for the relaxation too. To do so, let us start by introducing covered fanout counting variables, fo_u, and covered fanin counting variables, fi_u, for each movable node. In the running example, these would be:

$$fo_A = c_{A,B} + c_{A,C}; \quad fi_A = 0, \tag{24}$$
$$fo_B = 0; \quad fi_B = c_{A,B}, \tag{25}$$
$$fo_C = 0; \quad fi_C = c_{A,C}. \tag{26}$$

Let us focus on fo_A, since no other variable in the current example is interesting, as will soon become apparent. Let the binary variable $x_{u,i} = \sum_{p_u \in \{p=(x,y,j) \in P(u,W): j=i\}} x_{u,p_u}$ designate that the movable node u is placed at LUT_i in one of the clusters within its movement region. The following implication always holds: $(fo_A > 1) \implies x_{A,3} = 1$. To encode this, we can first introduce another binary variable $fob_{u,\theta}$, indicating that $fo_u \geq \theta$. We can assign a valid value to this variable with the help of the following two constraints [31]

$$\theta fob_{u,\theta} \leq fo_u, \tag{27}$$
$$(\mu - \theta + 1) fob_{u,\theta} + \theta - 1 \geq fo_u, \tag{28}$$

where μ is the largest number of connections originating at u, which could potentially be covered (upper bound on fo_u). In the running example the constraints would be

$$2 fob_{A,2} \leq fo_A, \tag{29}$$
$$fob_{A,2} + 1 \geq fo_A. \tag{30}$$

In the previous fractional solution, $(c_{A,B} = c_{A,C} = 1) \implies fo_A = 2$. Hence, constraint (30) implies that $fob_{A,2} = 1$. To complete the implication that the fanout constraint has on valid placement positions, we merely need to add the following constraint

$$fob_{A,2} \leq x_{A,3} = x_{A,(0,0,3)} + x_{A,(1,0,3)}, \tag{31}$$

which makes the previous fractional solution invalid in the continuous relaxation of the program as well. As can be seen in Figure 2, in a typical architecture, the fanin and the fanout of LUTs is rather small, which means that not many values of the θ-threshold need to be considered. This also allows for combining the fanin and fanout constraints. For example, encoding $(fib_{u,2} \wedge fob_{u,1}) \implies x_{u,4}$ would constraint a movable node with at least two covered incoming edges and one covered outgoing edge to LUT_4, as it is the only one which can support that in the architecture of Figure 2. It is important to note, however, that depending on the value of μ, degree matching may not be as effective as the above example illustrates. For instance, if μ and θ are 6 and 2, respectively, $fob_{u,2}$ can be as low as 1/5.

In principle this degree-matching approach could be recursively extended to counting the covered fanins/fanouts of predecessors and successors up to a certain distance [25], to further constrain the set of valid positions in the continuous relaxation. We have not tried this in practice yet.

9 RESULTS

In this section, we present the results of applying the proposed placement algorithm on the target architecture.

9.1 Experimental Setup

We inherit the delay modeling from our previous architectural study that produced the architecture of Figure 2 [26]. We also retain the experimental methodology, along with its limitations. Notably, we do not support carry chains, fracturable LUTs, nor sparse crossbars at the moment. One important restriction of the previous methodology is now lifted, however. We extended VPR to support cluster output equivalence specification after placement, independently for each cluster. As a result, there are no longer any constraints on route-time LUT permutation for the reference architecture, while for the one with the direct connections, only those LUTs that actually use a direct connection are kept fixed; the others may be freely permuted by the router. To further improve realism, we allow each cluster output in both architectures to reach all four adjacent routing channels. Thus, we avoid the situation where different pins have access to different channels, which is not representative of industrial architectures, such as Agilex [5]. At the moment, we do not have a sound method for legalizing the number of inputs to each cluster, so we increase the number of physical cluster inputs to 60 for both architectures (the maximum for a ten 6-LUT cluster). As this is not uncommon in industrial architectures [10, 17], we do not believe that it has any impact on the validity of the results.

All experiments were performed on a 20-core (40-thread) Xeon-based server with 256 GB of RAM, using CPLEX 12.10 with a timeout of 10 minutes for the solver. The reported results are medians of five different starting placements and each circuit was routed by the *delay-targeted* routing algorithm of Rubin and DeHon [28], implemented on top of VTR 7.0 [18], with the channel width fixed at 300 tracks.

9.2 Delays

In this section, we present the impact of applying the proposed algorithm to the architecture of Figure 2 on the critical path delay of the implemented benchmark circuits.

9.2.1 Postplacement Delays. For the cases when the LUTs are allowed to move only within their initial clusters ($W = 0$) and in a region of 3×3 clusters centered at the initial clusters ($W = 1$), the delays obtained through solving the sequence of placement ILPs (Section 7) are shown in the columns labeled as *covered* in Table 1. The > 400 ps difference between the average delays of 10.09 ns and 9.68 ns is significant and translates to about 3× greater relative average improvement over the reference, when LUTs are allowed to change clusters.

We may note that the 1.94% average improvement for the $W = 0$ case is noticeably lower than what was previously reported [26]. This could be an artifact of an inferior movable node selection method, although the lower bounds in Table 1 suggest a more fundamental cause. The cause is in fact of architectural nature: because we use a 60- instead of a 40-input cluster architecture, a much denser packing is obtained, bringing some of the intercluster routing delay into the clusters. Since the architecture has no local direct connections, when $W = 0$, the placer cannot do anything to improve their delay, while when $W > 0$ it can. To verify the hypothesis, we reran the experiments

Table 1. Delays in Nanoseconds

	postplacement							postrouting				
		lower bound		covered								
circuit	ref.	$W = 0$	$W = 1$	$W = 0$	$W = 1$	legal.	$-\Delta$[%]	ref.	w/ dir.	mux	$-\Delta$[%]	w/o dir.
raygentop	4.70	4.70	4.70	4.70	4.70	4.70	0.00	4.87	4.88	0.02	−0.21	4.88
ch_intrinsics	3.15	3.15	2.84	3.15	3.15	3.15	0.00	3.28	3.27	0.03	0.30	3.27
mkDelayWorker32B	6.83	6.83	6.55	6.83	6.58	6.58	3.66	7.09	7.04	0.03	0.71	7.36
mkSMAdapter4B	5.16	5.11	4.97	5.11	5.02	5.02	2.71	5.38	5.26	0.05	2.23	5.63
bgm	23.56	23.56	22.21	23.56	22.66	22.66	3.82	23.66	23.04	0.20	2.62	26.33
boundtop	6.10	6.01	5.57	6.01	5.73	5.73	6.07	6.05	5.82	0.05	3.80	6.37
stereovision0	3.74	3.74	3.31	3.74	3.52	3.52	5.88	3.74	3.57	0.06	4.55	4.06
diffeq1	20.45	19.48	18.24	19.81	19.19	19.19	6.16	21.16	20.01	0.12	5.43	21.86
diffeq2	15.69	14.92	13.46	15.02	14.48	14.48	7.71	16.14	15.14	0.11	6.20	16.68
blob_merge	9.90	8.76	6.79	9.44	8.90	9.16	7.47	9.89	9.21	0.11	6.88	10.56
or1200	13.08	12.66	10.76	12.77	11.69	11.75	10.17	13.12	12.20	0.23	7.01	15.66
LU8PEEng	105.05	101.07	91.47	101.49	95.57	95.63	8.97	104.86	96.45	0.98	8.02	110.17
sha	11.89	11.02	9.15	11.25	10.65	10.83	8.92	11.88	10.86	0.15	8.59	12.59
geomean	10.29	9.99	9.03	10.09	9.68	9.72	5.54	10.46	10.01	0.09	4.30	11.07

Each entry corresponds to a median of delays obtained for five different placement seeds. Entries have been computed independently of the corresponding entries in other columns. For instance, the routed critical path delay of *or1200* amounting to 12.20 ns does not necessarily correspond to the postrouting critical path delay of the placement for which the median postplacement critical path delay of 11.75 ns was obtained. Rather, the entries state that the median postplacement delay for this circuit was 11.75 ns, whereas the median routed delay was 12.20 ns. Similarly, the 230 ps of overhead due to direct-connection-selection multiplexers is the median penalty that was paid and does not necessarily correspond to the amount of overhead, which contributed to the median routed delay being 12.20 ns.

for $W = 0$ on a subset of circuits for which the average relative improvement was 2.22%, using a 40-input architecture. The improvement rose to 3.55%, which is much closer to the previously reported results [27]. This illustrates the importance of considering other architectural parameters when deciding which direct connections are the most beneficial.

The placements for $W = 0$ are legal by construction, but those for $W = 1$ are not. The postlegalization results are also reported in Table 1. The delay does sometimes deteriorate due to legalization, but in most cases by a modest amount.

9.2.2 Postrouting Delays. The postrouting delays are reported in the column designated as *w/ dir.* The postlegalization relative improvement is generally retained throughout the routing process. Many of the cases where a nonnegligible deterioration occurs can be explained by the delays of the additional multiplexers that are not modeled during placement. Those circuit connections that are implemented as direct are forced to suffer this additional delay, while the others are rarely impacted by it. This is due to the sparsity of the direct connections, which causes relatively few LUT inputs to be delayed. The difference that dedicated placement brings to the postrouting delay is shown in Figure 17.

In the placement ILP formulation, we allow connection delays to decrease only when implemented as direct. However, it is possible that some of the delay improvements in Table 1 are due to shortening of programmable connections or packing improvement. To verify if this is the case, we also routed the circuits placed with the dedicated algorithm, but without actually using the direct connections. The results are reported in the *w/o dir.* column of Table 1. Clearly, it is not the overall improvement of placement that led to the positive results. In fact, the dedicated placer significantly distorts the general placement, in a way that makes sense only in the presence of direct connections.

9.2.3 Sensitivity to the Starting Placement. In Section 3.3, we argued that it is sufficient to apply a dedicated detailed placer to a general starting placement produced without knowledge of existence of the fast direct connections, because the additional delay decreases due to appropriate usage of these direct connections is small compared to the amount of optimization that the general

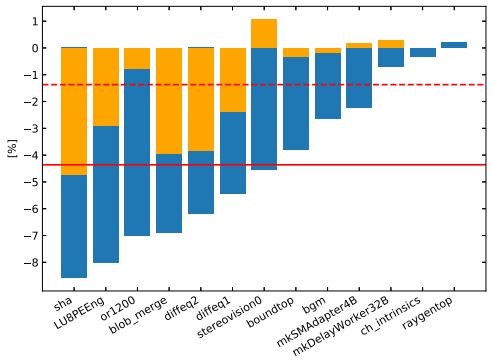


Fig. 17. Relative change in the postrouting critical path delay. The $W = 0$ and the $W = 1$ cases are shown in orange and blue, respectively. The dashed red line represents the relative change of the geometric mean critical path delay over all circuits, for $W = 0$, while the solid line represents the same for $W = 1$.

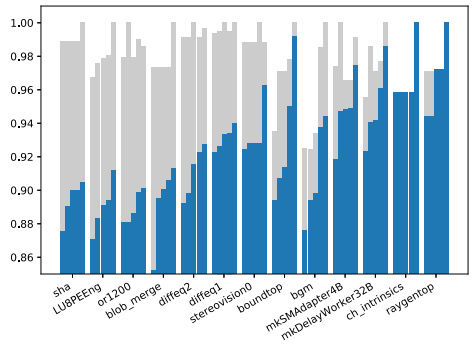


Fig. 18. Sensitivity to starting placement. Starting postplacement delays for all five starting placements of each circuit are shown in grey. Postlegalization delays for $W = 1$ are shown in blue. All values are normalized by the maximum starting postplacement delay occurring for the particular circuit.

placer can achieve with respect to a random, unoptimized placement. Given that in practice, during detailed placement, a small subset of movable nodes can be moved only to a limited distance, how much of this further delay reduction can actually be achieved could depend on the starting placement.

To assess this, we plot in Figure 18 the starting postplacement delays (grey) and the final postplacement delays after legalization (blue) for all five starting placements of each circuit. All delays corresponding to the same circuit are normalized by the largest starting postplacement delay occurring for that circuit. We can see that there are significant differences in the achieved relative delay improvement between different placements of the same circuit, even if the starting postplacement delay is the same. A notable example is the *blob_merge* circuit.

As discussed in Section 3, we believe that successfully constructing a dedicated placer that would combine global and detailed placement in a scalable manner, while actually maximizing the benefit from using the direct connections, is not particularly likely. Nevertheless, Figure 18 suggests that providing some information about the direct connections to the general placer may allow it to create more opportunity for the detailed placer to improve the critical path delay.

9.3 Improvement Subgraphs

The size and the structure of the circuit subgraphs induced by the connections selected for improvement (the solid edges in Figure 19) influence both the time needed to solve the placement ILPs and the achievable critical path delay reduction. Some basic properties of the last successfully placed subgraph in the run resulting in the median postplacement delay are given in Table 2. The circuits that achieved a final delay improvement of <3% are omitted, as their subgraphs were either very small, or no successful placement was found for any of them.

Perhaps the most apparent feature of the subgraphs is their fragmentedness, visible in the *components* columns, which show the sizes of the *weakly connected components* (maximal subgraphs where every node can be reached from all others when edge orientation is neglected). The *diameter* (longest of the shortest paths between all pairs of nodes) often remains substantial, however. The node degrees are low, which is appropriate for the architecture of Figure 2.

Table 2. Properties of the Subgraphs Induced by the Connections Selected for Improvement

	size				components			degrees				
circuit	$\|V\|$	$\|E\|$	W	H	#	$\langle\|V\|\rangle$	max$\|V\|$	$\langle$total$\rangle$	max total	max in	max out	diameter
boundtop	29	18	18	9	11	2.64	5	1.24	3	2	1	4
	25	14	18	9	11	2.27	3	1.12	2	2	1	3
stereovision0	51	39	39	13	12	4.25	16	1.53	5	4	2	6
	36	20	39	12	16	2.25	3	1.11	2	2	2	3
diffeq1	38	32	7	6	6	6.33	14	1.68	6	1	5	9
	31	19	7	6	12	2.58	4	1.23	3	1	2	4
diffeq2	38	33	6	4	5	7.60	17	1.74	4	3	2	11
	30	17	6	4	13	2.31	3	1.13	2	2	1	3
blob_merge	49	37	4	8	15	3.27	9	1.51	5	2	4	6
	47	28	4	8	19	2.47	3	1.19	2	2	2	3
or1200	97	71	14	16	26	3.73	11	1.46	3	3	2	11
	87	50	14	15	37	2.35	4	1.15	3	3	1	3
LU8PEEng	334	227	24	21	111	3.01	10	1.36	5	4	5	8
	294	164	24	21	130	2.26	4	1.12	3	3	2	3
sha	74	52	13	8	23	3.22	10	1.41	4	2	3	7
	59	33	13	8	26	2.27	4	1.12	2	2	2	4

The shaded rows show the corresponding properties of the subgraphs induced by the connections that were successfully improved by being implemented as direct. Columns W and H correspond respectively to the width and the height, in number of clusters, of the region bounding the movable nodes. Angular brackets denote an average.

The subgraphs induced by the connections that are actually implemented as direct (the blue edges in Figure 19) are noticeably smaller than the ones originally selected for improvement, but they still cover a large portion of their edges.

Without the information on how the individual connections selected for improvement are positioned within the entire circuit graph, it is not apparent how covering each of them influences the reduction of the critical path delay. We show one particular improvement subgraph in Figure 19. The dashed arrows mark the edges between the movable nodes that were not selected for improvement. It should not be surprising that they often occur as intermediate edges of paths that were selected for improvement. The intention of the selection LPs of Section 5—although there is no guarantee that it will actually be realized—is to select a minimal subset of edges of a path as this directly influences the size of the placement ILPs. We can see that, in this case, the numerous small connected components are not merely pieces of unrelated paths, but in fact constitute a carefully selected subgraph of a nontrivial graph. This showcases the generality of the movable node selection method that was mentioned in Section 4.

Another interesting observation that can be made from Figure 19 is that the connections that were successfully covered use a wide variety of direct connections available in the architecture, with different span lengths and directions, both vertical and horizontal. This seems to confirm what was found in prior work [26]: that using only very simple patterns of direct connections, such as the vertical cascades, may not expose their full potential.

9.4 Runtimes

Runtime breakdowns for the placement run that resulted in the median postplacement delay of the given circuit are reported in Table 3. Circuits *ch_intrinsics* and *raygentop* are omitted because for them no improvement was possible in the median case, and this was detected immediately after computing the lower bound on achievable critical path delay (Section 7). In all cases, the number of ILPs solved until convergence is small ($\leq$7). For majority of the solved ILPs, a solution at least as good as the previous best one was obtained, meaning that infeasible cases were often eliminated at the LP level, because the sought critical path delay was impossible to meet. The LP solution

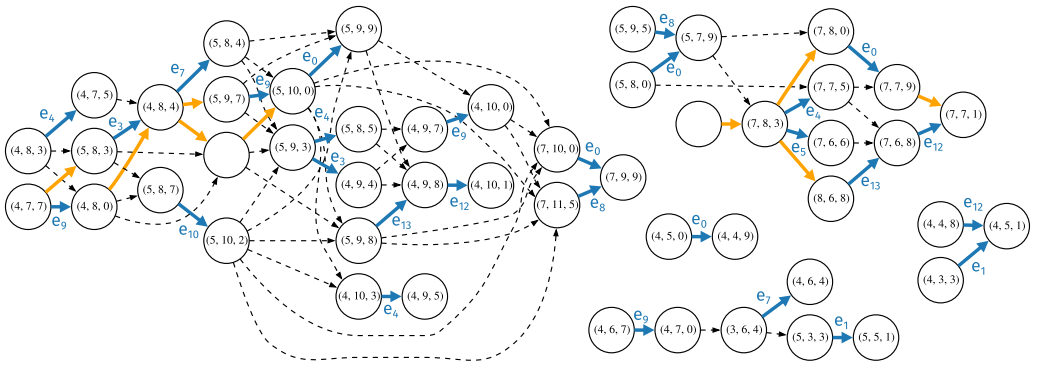


Fig. 19. Subgraph selected for improvement in the *blob_merge* circuit. All nodes are movable. The blue solid arrows correspond to the edges that were successfully implemented as direct after dedicated placement. The orange solid arrows represent the edges selected for improvement that were not successfully implemented as direct. Finally, the dashed arrows depict the edges that exist between the movable nodes but were not selected for improvement. Those nodes that have an incident direct connection have their final positions as labels. The label of each blue edge corresponds to the identifier used in Figure 2 to mark the direct connection that implemented it.

Table 3. Solution Runtime. All runtimes are in seconds

				ILP preparation		ILP status			flexibility (last feas.)			⟨ILP sol. t⟩	
circuit	#LUTs	wall clock	CPU	⟨LP sol. t⟩	⟨setup t⟩	feas.	infeas.	timeout	⟨pos./u⟩	⟨pair/e⟩	⟨|tree|⟩	feas.	infeas.
diffeq2	322	92.05	370.63	0.05	3.12	4	0	0	42	81	73	12.88	—
diffeq1	485	156.86	810.27	0.07	3.69	5	0	0	43	78	215	22.93	—
mkSMAdapter4B	1 982	52.82	34.92	0.33	2.38	2	1	0	32	37	0	0.06	0.04
sha	2 280	2 355.68	38 358.80	0.21	6.44	5	0	2	38	88	4 561	210.42	—
or1200	3 054	279.96	1 423.68	0.11	6.71	3	0	0	44	85	682	59.12	—
boundtop	3 070	87.45	73.94	0.22	3.64	4	1	0	31	50	0	0.44	0.02
mkDelayWorker32B	5 602	127.45	110.88	0.26	5.44	3	0	0	42	39	0	0.09	—
blob_merge	6 019	2 967.47	47 072.96	0.46	4.76	5	0	2	42	93	9 455	311.82	—
stereovision0	14 779	274.80	271.00	0.41	10.13	3	1	0	36	79	0	2.13	0.02
LU8PEEng	26 455	4 637.07	55 084.80	3.43	72.80	3	0	4	37	81	4 666	264.04	—
bgm	36 480	1 624.23	3 445.75	3.38	71.54	4	0	0	36	82	5 412	43.03	—

Columns under *ILP preparation* hold the average time taken to set up each ILP as well as the average time taken to solve the preceding LP. Columns under *ILP status* hold the number of algorithm iterations which resulted in a feasible ILP, an infeasible ILP, and a timed-out ILP, respectively. Columns under *flexibility* hold the average number of positions per movable node (covering position pairs per edge selected for improvement). Finally, columns under ⟨*ILP sol.t*⟩ hold the average time taken by the ILP solver to find a feasible solution (prove infeasibility), as well as the average size of the branching tree. The remaining runtime entering the wall clock time includes loading of datastructures, setting up the initial LP, attempts to solve infeasible LPs, and final legalization.

time was generally very small, with a trend of increasing with the increasing circuit size, which is expected since the LP models the entire circuit. This solution time can be further reduced by considering only the critical subgraphs of the timing graphs.

The solution times for the ILPs are displayed in the last two columns of the table. There seems to be no correlation between the size of the circuit and the solution time, which is expected, as the size of the movable node set has no a priori correlation with the circuit size either. Some of the ILPs are solved by merely solving the continuous relaxation of the problem in the root of the search tree (⟨|tree|⟩ = 0). Others, however, require substantial branching. In these cases, the capability of CPLEX to branch in parallel can be useful. For this size of the search trees, memory is, however, not a concern. The largest trees required on the order of a few hundreds of megabytes.

Each ILP also needs time to be constructed. This time is reported under the ⟨*setup t*⟩ column. In some cases, it is nontrivial, but this is mostly due to a fairly inefficient Python implementation.

Table 4. Critical Path Delays Obtained on the Architecture of Figure 20

	postplacement					postrouting				
circuit	ref.	$W = 1$	$-\Delta$ [%]	$W = 2$	$-\Delta$ [%]	ref.	$W = 1$	$-\Delta$ [%]	$W = 2$	$-\Delta$ [%]
raygentop	4.70	4.70	0.00	4.70	0.00	4.87	4.88	−0.21	4.88	−0.21
ch_intrinsics	3.15	3.15	0.00	3.15	0.00	3.28	3.30	−0.61	3.29	−0.30
mkDelayWorker32B	6.83	6.63	2.93	6.58	3.66	7.09	7.06	0.42	7.04	0.71
mkSMAdapter4B	5.16	5.02	2.71	5.02	2.71	5.38	5.31	1.30	5.25	2.42
bgm	23.56	22.85	3.01	22.66	3.82	23.66	23.20	1.94	22.84	3.47
boundtop	6.10	5.89	3.44	5.69	6.72	6.05	5.91	2.31	5.73	5.29
stereovision0	3.74	3.74	0.00	3.74	0.00	3.74	3.76	−0.53	3.76	−0.53
diffeq1	20.45	19.62	4.06	19.42	5.04	21.16	20.33	3.92	20.20	4.54
diffeq2	15.69	14.97	4.59	14.61	6.88	16.14	15.46	4.21	15.18	5.95
blob_merge	9.90	9.26	6.46	9.17	7.37	9.89	9.26	6.37	9.23	6.67
or1200	13.08	12.13	7.26	11.95	8.64	13.12	12.50	4.73	12.21	6.94
LU8PEEng	105.05	97.51	7.18	94.55	10.00	104.86	98.03	6.51	95.03	9.37
sha	11.89	10.97	7.74	10.87	8.58	11.88	11.01	7.32	10.92	8.08
geomean	10.29	9.90	3.79	9.78	4.96	10.46	10.16	2.87	10.04	4.02

The average number of positions per movable node (⟨pos./u⟩) is substantial in most cases. It is lower than the theoretical maximum of 90, however. The average number of covering pairs for each edge (⟨pair/e⟩) is also much lower than the 8,100 that would occur in the naive formulation.

Finally, the table also shows wall-clock times measuring the duration of the entire detailed placement run, from loading datastructures with the output of VPR to storing the postlegalization results. These times also include a one-time LP setup, which is later updated by merely changing the target critical path delay bound and the objective of Section 8.2.2. As with ILP setup, the combined overhead of the aforementioned steps is tolerable, but not always negligible (maximum being 1,140s for *bgm*); we believe this to be mostly due to a fairly inefficient Python implementation.

The reader may notice that the runtime spent on dedicated detailed placement is substantial, given the size of the circuits in Table 3. In some cases, it even overshadows the rest of the CAD flow; for example, the standard VTR 7.0 flow takes only 22.34 s on the *blob_merge* circuit, with a fixed routing channel width. The fact that this single additional stage in the CAD flow requires 133× more runtime than the rest may seem daunting at first, but it is important to note that its most runtime-intensive phase—solving the placement ILPs—depends not on circuit size, but on the size of the movable node sets. On the other hand, runtime expanded on the rest of the CAD flow is directly related to the circuit size, meaning that for larger circuits, the algorithm proposed here would likely take but a fraction of the total runtime.

9.5 Independent Subpattern

In our previous architectural study, we observed that the first four direct connections that were added to the pattern were responsible for 68% of the geomean routed critical path delay reduction achieved by the complete pattern of Figure 2 [26]. The subpattern containing these four connections is shown in Figure 20. A particularly interesting feature of this architecture is that each LUT has at most one incident direct connection. This means that the set of edges selected for improvement that are covered by any valid solution of a placement ILP will constitute a *matching* in the circuit graph. Since matchings can be found efficiently [11], we can intuitively expect that the placement problems formulated for this class of architectures are easier to solve in practice, although the timing dependence between the edges and the necessity to avoid overlaps between the movable nodes could mean that they are not necessarily easier in theory.

9.5.1 Optimization-Runtime TradeOff. To assess whether the subpattern of Figure 20 still causes bulk of the improvement achieved by the entire pattern of Figure 2 when nodes are allowed to move across clusters, we repeat the experiments on it as well. The results shown in Table 4 indeed

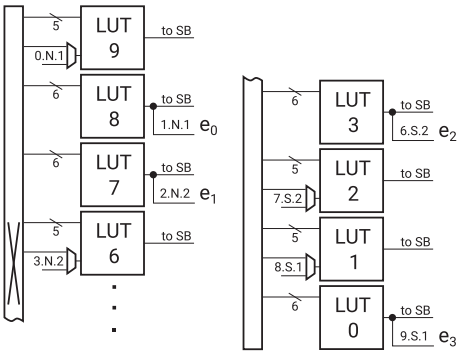


Fig. 20. The architecture composed of the first four direct connections added to the best pattern found in the previous architectural study [27]. Each LUT has at most one incident direct connection. LUTs without incident direct connections are omitted from the figure.

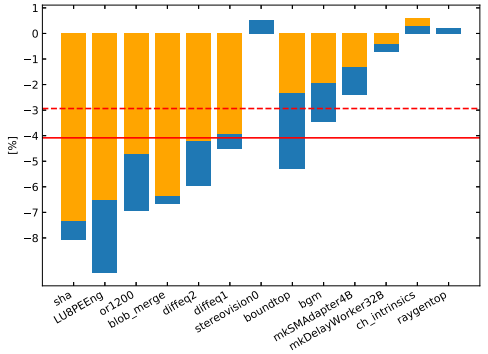


Fig. 21. Relative change in the postrouting critical path delay due to direct connections of the architecture of Figure 20. The $W = 1$ and the $W = 2$ cases are shown in orange and blue, respectively. The dashed red line represents the relative change of the geometric mean critical path delay over all circuits, for $W = 1$, while the solid line represents the same for $W = 2$.

confirm this, with the architecture achieving a 2.87% geomean routed delay reduction, or 67% of the 4.30% achieved by the complete architecture. The total wall-clock runtime spent for median-postplacement-delay runs of all circuits was 2,760s, while the total CPLEX runtime amounted to only 336s. For the complete architecture, the total wall-clock runtime was 12,720s, while the solver alone used 8,790s. Hence, 67% of the delay improvement was achieved in 4.6× less total time and using 26× less solver runtime.

9.5.2 Increasing Movement Freedom. The much smaller solver runtime spent on the small independent subpattern of Figure 20 allowed us to assess the potential benefits of increasing the movement freedom of the movable nodes on the obtained critical path delay reduction. Results for $W = 2$—allowing each node to move in a 5×5 cluster region with 250 candidate positions—are also shown in Table 4, as well as in Figure 21. They show that a number of circuits experience a large additional improvement and that the average improvement is significantly increased as well. This suggests that future effort invested in making the solution of the placement problems more scalable with respect to freedom of movement may pay off. As an illustration, with $W = 2$ the solver runtime rose to 7,025s (21× increase).

It is also interesting to observe that by increasing W to 2, the simple architecture of Figure 20 was able to achieve 93% of the geomean routed critical path delay reduction achieved by the architecture of Figure 2 for $W = 1$, while still using less runtime. This illustrates that increased CAD effort may compensate for architectural inefficiencies. It could also make the direct connections more compelling, given that the architecture of Figure 20 is significantly simpler to implement.

10 CONCLUSIONS AND FUTURE WORK

In this work, we introduced an efficient ILP-based placement algorithm for FPGA architectures with direct connections between LUTs, which vastly improves their effectiveness compared to architecture-oblivious algorithms. We also removed some important limitations of the previously used experimental methodology and showed that the direct connections continue to bring benefits in this more realistic setting.

The fact that a simple change in the underlying architecture—increase in the number of cluster inputs—substantially altered the conclusions about the utility of a particular type of direct connections suggests that a comprehensive study of the mutual influence of direct connections and other architectural parameters is due.

Our experiments showed that increasing the freedom of movement beyond what was done in this work would lead to increased benefits. Another future step on the algorithmic front should therefore be to address the scalability issues that prevent this at the moment, by integrating incremental solution approaches [16], or even other solution techniques, such as SAT or SMT [23, 24], that could be better suited to the nature of the problem. Additional performance gains could perhaps also be achieved by repeated application of the algorithm with previously replaced nodes kept fixed and by further improving the problem formulation along the lines of the discussion of Section 8.3.

REFERENCES

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

[2] Stephen Cauley, Venkataramanan Balakrishnan, Y. Charlie Hu, and Cheng-Kok Koh. 2011. A parallel branch-and-cut approach for detailed placement. *ACM Transactions on Design Automation of Electronic Systems* 16, 2 (2011), 18:1–18:19.

[3] Gang Chen and Jason Cong. 2004. Simultaneous timing driven clustering and placement for FPGAs. In *Proceedings of the Field Programmable Logic and Application*, Jürgen Becker, Marco Platzner, and Serge Vernalde (Eds.), Berlin, Heidelberg, 158–167.

[4] Scott Y. L. Chin and Steven J. E. Wilton. 2011. Towards scalable FPGA CAD through architecture. In *Proceedings of the ACM/SIGDA 19th International Symposium on Field Programmable Gate Arrays, FPGA 2011, Monterey, California, February 27, March 1, 2011*, John Wawrzynek and Katherine Compton (Eds.), 143–152.

[5] 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 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Seaside*. ACM, 140–149.

[6] Kevin Chung. 1994. *Architecture and Synthesis of Field-Programmable Gate Arrays with Hard-wired Connections*. Ph. D. Dissertation. University of Toronto.

[7] Nima Karimpour Darav, Andrew A. Kennings, Kristofer Vorwerk, and Arun Kundu. 2019. Multi-commodity flow-based spreading in a commercial analytic placer. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 122–131.

[8] Shounak Dhar, Saurabh N. Adya, Love Singhal, Mahesh A. Iyer, and David Z. Pan. 2016. Detailed placement for modern FPGAs using 2D dynamic programming. In *Proceedings of the 35th International Conference on Computer-Aided Design*. ACM, 1–8.

[9] Shounak Dhar, Mahesh A. Iyer, Saurabh N. Adya, Love Singhal, Nikolay Rubanov, and David Z. Pan. 2017. An effective timing-driven detailed placement algorithm for FPGAs. In *Proceedings of the 2017 ACM on International Symposium on Physical Design*. ACM, 151–157.

[10] Wenyi Feng. 2012. K-way partitioning based packing for FPGA logic blocks without input bandwidth constraint. In *Proceedings of the 2012 International Conference on Field-Programmable Technology, FPT 2012, Seoul, Korea (South), December 10–12, 2012*. IEEE, 8–15.

[11] Alan M. Gibbons. 1985. *Algorithmic Graph Theory*. Cambridge University Press.

[12] Susanne E. Hambrusch and Hung-Yi Tu. 1997. Edge weight reduction problems in directed acyclic graphs. *Journal of Algorithms* 24, 1 (1997), 66–93.

[13] Michael Hutton, Vinson Chan, Peter Kazarian, Victor Maruri, Tony Ngai, Jim Park, Rakesh Patel, Bruce Pedersen, Jay Schleicher, and Sergey Shumarayev. 2002. Interconnect enhancements for a high-speed PLD architecture. In *Proceedings of the 2002 ACM/SIGDA 10th International Symposium on Field-Programmable Gate Arrays*. Monterey, CA, 3–10.

[14] David Lewis, Vaughn Betz, David Jefferson, Andy Lee, Chris Lane, Paul Leventis, Sandy Marquardt, Cameron McClintock, Bruce Pedersen, Giles Powell, Srinivas Reddy, Chris Wysocki, Richard Cliff, and Jonathan Rose. 2003. The stratix routing and logic architecture. In *Proceedings of the 2003 ACM/SIGDA 11th International Symposium on Field Programmable Gate Arrays*. Monterey, CA.

[15] Chen Li, Min Xie, Cheng-Kok Koh, Jason Cong, and Patrick H. Madden. 2007. Routability-driven placement and white space allocation. *IEEE Transactions on CAD of Integrated Circuits and Systems* 26, 5 (2007), 858–871.

[16] Shuai Li and Cheng-Kok Koh. 2012. Mixed integer programming models for detailed placement. In *Proceedings of the International Symposium on Physical Design*. ACM, 87–94.
[17] Wuxi Li and David Z. Pan. 2019. A new paradigm for FPGA placement without explicit packing. *IEEE Transaction on CAD of Integrated Circuits and Systems* 38, 11 (2019), 2113–2126.
[18] Jason Luu, Jeffrey Goeders, Michael Wainberg, Andrew Somerville, Thien Yu, Konstantin Nasartschuk, Miad Nasr, Sen Wang, Tim Liu, Nooruddin Ahmed, Kenneth B. Kent, Jason Anderson, Jonathan Rose, and Vaughn Betz. 2014. VTR 7.0: Next generation architecture and CAD system for FPGAs. *ACM Transactions on Reconfigurable Technology and Systems* 7, 2 (July 2014).
[19] Jason Luu, Conor McCullough, Sen Wang, Safeen Huda, Bo Yan, Charles Chiasson, Kenneth B. Kent, Jason Anderson, Jonathan Rose, and Vaughn Betz. 2014. On hard adders and carry chains in FPGAs. In *Proceedings of the 2014 IEEE 22nd International Symposium on Field-Programmable Custom Computing Machines*. 52–59.
[20] Igor L. Markov, Jin Hu, and Myung-Chul Kim. 2012. Progress and challenges in VLSI placement research. In *Proceedings of the 2012 IEEE/ACM International Conference on Computer-Aided Design*. ACM, 275–282.
[21] Alexander Marquardt, Vaughn Betz, and Jonathan Rose. 2000. Timing-driven placement for FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. ACM, 203–213.
[22] Timothy Martin, Dani Maarouf, Ziad Abuowaimer, Abeer Alhyari, Gary Grewal, and Shawki Areibi. 2019. A flat timing-driven placement flow for modern FPGAs. In *Proceedings of the 2019 56th ACM/IEEE Design Automation Conference*. 1–6.
[23] Andrew Mihal. 2013. A difference logic formulation and SMT solver for timing-driven placement. In *Proceedings of the SMT Workshop 2013 11th International Workshop on Satisfiability Modulo Theories. Informal Proceedings*. 16–25.
[24] Andrew Mihal and Steve Teig. 2013. A constraint satisfaction approach for programmable logic detailed placement. In *Proceedings of the 16th International Conference on Theory and Applications of Satisfiability Testing. Proceedings (Lecture Notes in Computer Science, Vol. 7962)*. Springer, 208–223.
[25] Stefan Nikolić, Grace Zgheib, and Paolo Ienne. 2020. Timing-driven placement for FPGA architectures with dedicated routing paths. In *Proceedings of the 2020 30th International Conference on Field-Programmable Logic and Applications*. 153–161.
[26] Stefan Nikolić, Grace Zgheib, and Paolo Ienne. 2019. Finding a needle in the haystack of hardened interconnect patterns. In *Proceedings of the 2019 29th International Conference on Field Programmable Logic and Applications*. 31–37.
[27] 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*. Seaside, CA, 150–160.
[28] Raphael Rubin and André DeHon. 2011. Timing-driven pathfinder pathology and remediation: Quantifying and reducing delay noise in VPR-pathfinder. In *Proceedings of the ACM/SIGDA 19th International Symposium on Field Programmable Gate Arrays*. ACM, 173–176.
[29] Elias Vansteenkiste, Alireza Kaviani, and Henri Fraisse. 2015. Analyzing the divide between FPGA academic and commercial results. In *Proceedings of the 2015 International Conference on Field Programmable Technology*. 96–103.
[30] Kristofer Vorwerk, Andrew Kennings, Jonathan Greene, and Doris T. Chen. 2007. Improving annealing via directed moves. In *Proceedings of the 2007 International Conference on Field Programmable Logic and Applications*. 363–370.
[31] H. Paul Williams. 2013. *Model Building in Mathematical Programming* (5th edition ed.). Wiley.

Received 30 June 2021; revised 29 September 2021; accepted 19 November 2021