Finding a Needle in the Haystack of Hardened Interconnect Patterns

Stefan Nikolić*, Grace Zgheib†, Paolo Ienne*
*École Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences, 1015 Lausanne, Switzerland
{stefan.nikolic, paolo.ienne}@epfl.ch
†Intel Corporation
San Jose, USA
grace.zgheib@intel.com

Abstract—**Circuits naturally exhibit recurring patterns of local interconnect. Hardening those patterns when designing *Field Programmable Gate Array* (FPGA) clusters can both eliminate slow programmable connections from the critical path and remove the need for transistors to implement them. While we may be able to manually design such clusters, based on intuition and observations, such an endeavour will always leave us in doubt whether we have used the potential of cheap and fast hardened connections to the fullest. On the other hand, since there are $\sim 10^7$ possible patterns of interconnect among only eight 5-input *Look-Up Tables* (LUTs), even without considering the possibilities for enforcing input sharing, performing an exhaustive exploration of the design space seems like a task as daunting as finding a needle in a haystack. Despite their enormous sizes, design spaces spanned by such cluster architectures are very well structured. In this paper, we leverage that structure to limit the search to only those points of the space that may perform better than any chosen reference. We demonstrate the usefulness of our techniques by narrowing the space spanned by five 5-LUT structures from hundreds of millions to a set of 261 structures that achieve $\geq 80\%$ utilization on a set of standard benchmarks, while requiring only 12 external inputs. We believe that the exploration techniques presented here are an important step towards truly exploiting the potentials of hardening programmable connections.**

I. Introduction

The idea of hardening connections between *Look-Up Tables* (LUTs) to remove some of the slow programmable connections from the critical path is not new. Notable examples include a cascade of three LUTs used in the *UTFPGA1* [6] and a structure composed of two LUTs feeding outputs to a third one in Xilinx *XC4000* [12]. These examples are very simple, single instances of an enormous number of possible structures and the question of whether a different, more complex choice could not have produced better results naturally arises. Academic research has gone further in studying more complex structures, by considering classes of trees [7]. Since patterns commonly occurring in circuits are not limited to trees [25], it seems natural to expect that more general structures would bring better results. Hardening of interconnect patterns does not need to stop at connections between LUTs. It can also be extended to patterns in which inputs are shared among them. Each input pair that gets fused eliminates the need for one whole multiplexer in the crossbar. Such an approach was already successfully used in fracturable logic elements [13], to reduce the area impact of large LUTs. This is a specific example, where the hardened pattern resulted from LUT implementation details, but there is no apparent reason to believe that extending the approach to the level of the largest implementable LUTs would not be successful in bringing down the crossbar cost.

One problem in using large, complex structures is that the algorithms currently available do not scale well with their size [3], [21]. Another, at least equally important problem is that of obtaining the right structures. Trying to handcraft them would always leave us in doubt whether we failed to visit the most interesting portions of the design space. Concentrating on patterns extracted from the circuits or observed to be used by CAD tools would not alleviate this problem either. The first approach depends highly on our ability to extract meaningful information from the circuits, while the second suffers from the issue that if the CAD algorithms did not make use of some pattern it does not mean that they would not have done so had they been deprived of other options.

To the best of our knowledge, prior studies considered at most a few hundred structures [7] meaning that all of them could be evaluated individually. Given that there are about two million ways just to share inputs between five 5-LUTs, exhaustive exploration of the design space seems not to be an option when structures more complex than trees are considered. Despite being enormous, this space is well structured. In this work, we introduce efficient techniques for incremental enumeration of LUT interconnect patterns, allowing us to leverage that structure. We demonstrate the usefulness of our approach by narrowing the space spanned by five 5-LUT structures from hundreds of millions to a set of 261 that, when replicated to form a 20-LUT cluster, require only 12 external inputs, while achieving $\geq 80\%$ utilization on a set of standard benchmarks.

The rest of the paper is organized as follows. Section II discusses prior work. Section III introduces techniques for exposing the structure of the design space and enumerating individual cluster architectures in a way that preserves this structure. A novel packing [5] algorithm designed for scalability, while taking into account the existence of hardened connections is presented in Section IV. Some practical concerns about testing usefulness of particular LUT structures are discussed in Section V. Experimental results are presented in Section VI, while Section VII draws final conclusions and comments on future work.

II. Prior work

A comprehensive study of LUT structures is given by Chung [7]. He considers only rooted directed trees, of small

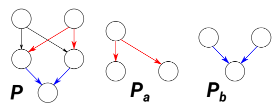


Fig. 1: Complex Interconnect Patterns as Combinations of Simpler Ones. An interconnect pattern (P) and two of the simpler patterns it contains (P_a and P_b). Each node represents a LUT.

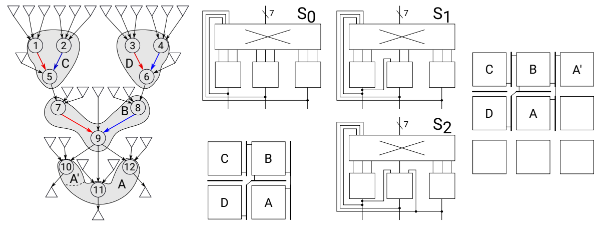


Fig. 2: Nonmonotonicity of Delay with Respect to Connection Hardening. Hardening one connection in the classical cluster with a fully populated crossbar (S_1) speeds up the red connections in the example circuit on the left. This brings no delay improvement, but results in splitting of the cluster A into A and A'. Hardening another connection, (S_2) brings no further packing quality deterioration and results in removal of two programmable local connections from the critical path.

size, without input sharing. Chung's results were further extended by Betz and Rose [4] to create a family of specialized architectures each fit for a different purpose. Ahmed [1] raised a question of whether it is possible to use a simple cascade of two 4-LUTs, to obtain speed benefits of a 7-LUT, without its area impact. This question was recently positively resolved through a series of papers by Ray et al. [21], Mishchenko [20], and Feng et al. [8]. They also presented general techniques for both technology mapping and delay-optimal packing for LUT structures. However, their work did not focus on efficiently exploring such structures and all of those actually analyzed remain simple.

A different attempt at gaining benefits of larger LUTs without their area penalty was presented by Hutton et al. [13], in a work that introduced the now standard concept of fracturable logic elements, where pairs of LUTs share inputs and are combined into a larger LUT by a multiplexer. Optimal amount of input sharing for such pairs was determined by Jiang et al. [14]. A study of Wang et al. [24] explored patterns in global routing, but was based only on empirical observations of what CAD algorithms generally used. Substantial research into patterns occurring in digital circuits was performed in the domain of computational biology [25]. Techniques for assessing graph similarity in protein interaction network alignment [15] gave inspiration to those used in our packer. Finally, the work of Goldberg [11] contains valuable results in the field of graph enumeration. While the algorithms presented in this paper were developed independently by the authors, it is possible that they overlap somewhat with those introduced by Goldberg.

III. Searching the design space

The design space is too large for us to visit each of its individual points. However, the interconnect pattern hardened in each of the structures is composed of a combination of simpler patterns (Figure 1). Identifying which of these are useful may not be easy, because they might become useful only when combined, as illustrated in Figure 2. Structure S_1 is obtained from the classical cluster (S_0) by hardening a single connection. In the circuit on the left, this is not

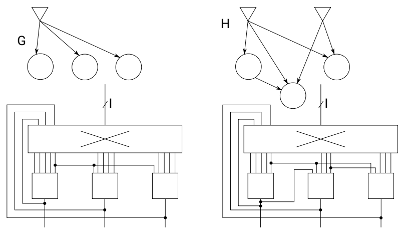


Fig. 3: Representing Structures as Graphs. Two example graphs, one being a subgraph of the other, and the structures they induce. Each LUT node (circle) is complemented by inputs to K (5 in this case) and has an optionally registered output observable from outside of the cluster. All crossbars are fully populated, unless specified otherwise.

sufficient to bring any reduction of the critical path delay, as, despite converting three local connections (red) from programmable to hardened, the critical path still contains two programmable local connections $((4,6),(8,9))$. Hardening another connection to form S_2 removes these two connections from the critical path. Performance is generally not monotonic with respect to connection hardening. For instance, performance of S_1 may be worse than that of the classical cluster, for the given example circuit, because the decreased packing flexibility leads to splitting of the cluster A, which may result in higher loading of LUT 9. If it were monotonically decreasing, determining that hardening a particular pattern leads to unacceptably poor performance would remove the need to consider all the structures whose hardened pattern contains it. Assuming that the crossbar is as fast as a hardened connection grants us this monotonicity, because, under that assumption, further connection hardening can only reduce packing flexibility but can no longer improve the performance. We formalize these thoughts in Section III-A. When coupled with careful incremental generation of structures presented in Section III-B, they become a powerful tool for making exploration of hundreds of millions of structures possible.

A. *Understanding the design space*

Let us define a *feasible* structure as a structure achieving post-routing delay and area below certain thresholds for a given circuit. We represent the structures as graphs (I, L, E), where I is the set of input nodes, L the set of LUT nodes, and $E \in (I \cup L) \times L$ the set of edges. There is no need to explicitly specify them entirely, though. We may say that, given a graph on N LUT nodes, its extension to a complete structure is obtained by complementing each LUT node by the appropriate number of input nodes, so that its in-degree is K. The complementary inputs are not shared among LUTs. Two example graphs and the structures induced by them, along with cluster crossbars are shown in Figure 3.

Graph G of Figure 3 is a subgraph of H,[1] meaning that the structure induced by G is less constrained than that induced by H. For a structure to be feasible, it suffices that there exists one realization of each design of interest in the FPGA architecture based on that particular structure, which achieves delay and area below the respective thresholds. Let us fix the global interconnect for all structures and set the reported area and delay of the crossbar and all hardened LUT-LUT connections

[1]We assume that all graphs have the same number of LUT nodes, even if some of them are isolated. Hence, only omission of edges and input nodes is allowed when forming a subgraph.

to zero.[2] This makes the cost of using the crossbar to connect an input of a particular LUT to any possible driver the same as if there existed a hardened connection achieving the same. From that and the way structures are induced by graphs, the following lemma immediately follows.

Lemma 1. *The structure induced by graph G is feasible only if the structures induced by all of its subgraphs are feasible.*

This becomes useful once we remember that a simple way of enumerating graphs is by iterative addition of edges [11]. If at any point in enumeration, a particular graph induces an infeasible structure, we can stop adding edges to it, since we know that structures induced by graphs obtained after any such addition will be infeasible as well.

B. Efficiently enumerating graphs

Now that we have set the general framework for our exploration, we need to find an efficient way to enumerate the structures, while keeping information about subgraph relations (*containment*).

We limit the scope of the search performed in this work to structures induced by acyclic graphs on five 5-LUT nodes. Without input sharing, there are only 302 such graphs[3], which we can enumerate naively. Let us call these graphs *kernels*. Adding input sharing, however, makes the problem considerably more complex. To minimize the amount of feasibility checks, we would like to output only one graph per isomorphism class. Figure 4 depicts two isomorphic graphs, obtained by adding input nodes to the same kernel. Recalling the concept of a *(vertex) orbit*—given a node v of a graph G, its orbit is the set $\{a(v) : a \in Aut(G)\}$, where $Aut(G)$ is the group of automorphisms of G [10]—we can notice that nodes B and C belong to the same orbit of the kernel. Representing the kernel by its orbits, instead of individual nodes, allows us to remove such duplicates.

For each input node we add, we need to list the edges connecting it to orbit representatives. If we keep the orbit representatives sorted, this translates to forming an ordered multiset (i.e., "set" with replicated elements) where each element represents the number of edges going to the particular orbit representative (possibly none). All inputs together are represented by a multiset of such multisets. Graph H of Figure 4 shows an example of adding three inputs to a kernel, along with the corresponding multisets. Our task is to enumerate all such multisets of multisets. Some of them will be discarded due to the number of inputs to particular orbits exceeding the total number of inputs of their elements.

Graph H represents another graph, besides G_1 and G_2. This is because there is, in general, more than one way of distributing the edges among the elements of an orbit to the representative of which they are incident. Thus, to obtain all graphs, we must go through the process of *expansion*. We classify the inputs according to their assigned multisets and then, for each orbit, assign an ordered multiset to each of its nodes, designating

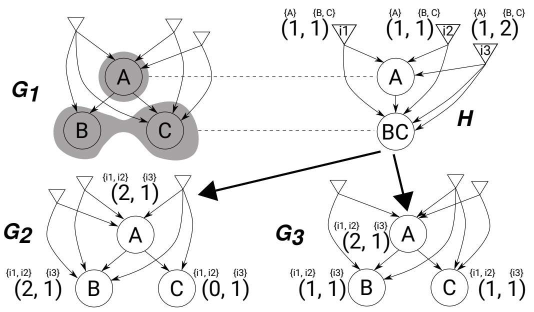


Fig. 4: Using Kernel Orbits to Avoid Listing Isomorphic Graphs. Graphs G_1 and G_2 are isomorphic. To avoid listing both of them, we can use the fact that nodes B and C belong to the same orbit of the kernel (an automorphism maps one to the other) and represent the kernel in terms of its orbits (H). We specify each input node by an ordered multiset (shown next to nodes) where each element indicates the number of edges to a particular orbit $((\{A\}, \{B, C\}))$. Edges represented by the multiset elements can be distributed among orbit elements in multiple ways resulting in nonisomorphic graphs (G_2 and G_3). Hence we need to classify the input nodes based on their multisets and assign to each node of an orbit a multiset specifying how many edges it takes from a particular input class $((\{i_1, i_2\}, \{i_3\}))$.

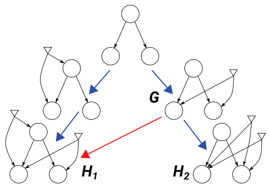


Fig. 5: Incompleteness of the Containment Information in the Enumeration Tree. Enumeration tree (blue edges) only carries information about containment of graphs that are obtained from one another by addition of input nodes. Since the red edge between graphs G and H_1 does not exist, if we prove graph G to be infeasible, we still need to perform the check on H_1, even though G is its subgraph. With the number of graphs rising exponentially with the number of inputs added, introducing the missing containment edges becomes a necessity.

the numbers of edges connecting it to the representatives of particular input classes, much like we did in the case of input adjacency enumeration. The assigned multisets serve to split the orbits into new equivalence classes. We finally need to update the input classes based on the resulting splitting. This method still fails to be exhaustive when there are orbits of cardinality greater than 3, or edges connecting orbits of cardinality greater than 1. We found only 19 (out of 302) kernels with such characteristics and chose to leave extending the algorithm to full exhaustiveness for future work.

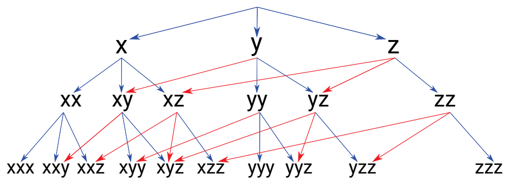
C. Constructing the containment graph

If we enumerate graphs by iteratively adding inputs to the kernel, one at a time, they will constitute a tree, where each graph connected by a path to another one would be its subgraph. Containment information encoded in this enumeration tree is not complete, however. Graph G of Figure 5 is a subgraph of both H_1 and H_2, but only H_2 is generated from it, as indicated by the edge in the enumeration tree. If a feasibility check renders G infeasible, H_1 will still be checked. To avoid this, we need to introduce additional containment information.

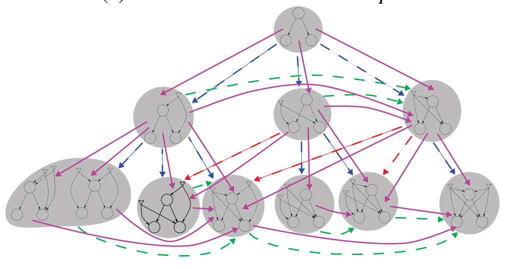
Remember that, to describe adjacency of an input, we assign to it an ordered multiset, where each element tells the number of edges connecting it to a particular orbit representative. Let us construct a set W containing all legal multisets and order it arbitrarily. We say that x is *smaller* than y if it appears in W before it. To describe an input sharing pattern with I

[2]Any constant can be chosen for area, provided that the corresponding threshold is adjusted accordingly. Any constant can also be chosen for delay, provided that it is the same for the hardened connections and the crossbar and that the delay threshold is adjusted accordingly.

[3]Counted by brute-force enumeration of underlying undirected graphs, cross-checked with *nauty* [19], followed by brute-force enumeration of acyclic orientations.



(a) *Abstract Containment Graph*



(b) *Concrete Containment Graph*

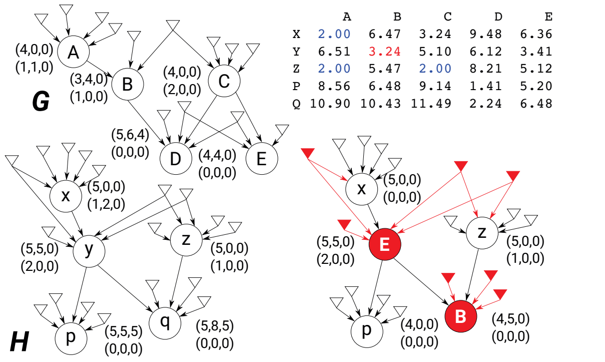
Fig. 6: Adding More Containment Information to the Enumeration Tree. Ordering the set of possible multisets specifying input adjacency and adding them in a nondecreasing fashion creates a regular recursive structure in the enumeration tree, as indicated by letters (x, y, z) (each designating a multiset) occurring as last at the first enumeration level of Figure 6a, $((x, y, z), (y, z), (z))$ at the second, etc. We can thus add the containment information not found in expansion edges (blue) using simple positional rules (red edges). Each node of such an *abstract containment graph* may contain more than one graph. We must add edges between these graphs, thus constructing a *concrete containment graph* (Figure 6b; the lowest level of Figure 6a not shown). Each abstract edge (dashed) indicates that graphs at its tail may be subgraphs of graphs at its head. If there is only one graph at either head or tail, an edge between it and all the graphs at the opposite end can be added (solid). Otherwise, explicitly checking containment may be necessary. Green abstract edges indicate the fact that the multiset last added to the tail node is contained in that added to the head node.

inputs, we draw I elements from W, one by one, placing them in a multiset.[4] To ensure that there are no multisets that are permutations of one another, for they describe the same sharing patterns, it is sufficient to only allow drawing of elements that are not smaller than the largest one already in the multiset. If we also draw elements in the order they appear in W, when extending the same multiset, elements will appear in the enumeration tree in a strictly regular, recursive fashion. This makes it easy to deduce a set of simple position-based rules, that describe containment of different multisets (Figure 6a).

We call the graph obtained by adding this additional containment information to the enumeration tree an *abstract containment graph*, because each of its nodes holds a number of graphs (obtained in the expansion phase described in Section III-B) and its edges do not in general mean containment between these graphs. Lack of an edge between two of its nodes u and v means, however, that no graph from u can be contained in any graph from v. We need to *concretize* the containment graph by determining the edges between the graphs in its nodes. Some of the abstract edges can be concretized trivially, but for some we must check containment between each of the graphs in the tail node and the head node. Although checking subgraph isomorphism is generally an intractable problem, specificities of the variation at hand make it nonproblematic. An example of a *concrete containment graph* is shown in Figure 6b.

Containment graphs are constructed incrementally. All graphs receiving an edge from an infeasible graph immediately

[4]It is possible that this multiset represents more than one input sharing pattern, as shown in Figure 4.



	A	B	C	D	E
X	2.00	6.47	3.24	9.48	6.36
Y	6.51	3.24	5.10	6.12	3.41
Z	2.00	5.47	2.00	8.21	5.12
P	8.56	6.48	9.14	1.41	5.20
Q	10.90	10.43	11.49	2.24	6.48

Fig. 7: Considering Topology in Local Placement. Graphs G and H represent the logical and the physical cluster, respectively. If we place candidate A at position y, placing candidate D becomes impossible. Counting the predecessors and successors at increasing distances and comparing the obtained vectors (shown next to nodes), we can create a similarity matrix, where a smaller value indicates a more favorable placement. Now the best candidate for position y is B. If we do not take care of input sharing, positions x and z are considered equal for candidate A. Hence, in final score computation, we include another similarity measure for that. Let us suppose that we misplaced candidates E and B at positions y and q respectively (graph on the bottom-right). The best position for node D would still be p, due to positions y and x still being counted, even though they are not accessible. To prevent this, we stop counting at occupied positions and count connections to their candidates instead. The new k-distance counts are shown next to nodes.

become infeasible themselves, while the remaining graphs of the last level are checked sequentially. If any abstract node is left without feasible graphs, we stop branching from it.

IV. Packing Algorithm

Having found a way to represent our search space in a well-structured manner, we can proceed with describing how to check the feasibility of its points. The first essential step to this goal is to do appropriate packing, that takes into account the existence of hardened connections. Although delay optimization is not a primary concern in this work, to protect the critical path from being cut too often, we employ the method of Vercruyce et al. [23], based on min-cut partitioning, when determining the logical clusters. While such an approach may not be the best for maximizing use of hardened connections, because the partitioner has no knowledge of their existence, it does help in bringing down the problem size, which was one of the major issues in prior research in the field [3]. We further simplify the problem by assuming that the logical clusters provided by the partitioner are ideal and try to cover as much of them as possible.

Conceptually, our algorithm is very similar to *AAPack* [17]. It keeps adding nodes to the physical cluster, checking routability at each step, until no more nodes can be added. The major difference is that local placement now plays a much more important role in maximizing cluster utilization. Let us denote the LUT of the logical cluster as *candidate* and the LUT of the physical cluster to which we are trying to assign it as *position*. Not caring about topological similarity of their neighborhoods can have large impact on future steps, as illustrated in Figure 7.

To describe the topology of neighborhoods of different nodes, we count their successors and predecessors at distance k, for k ranging from 1 to the length of the longest path in the structure, store the counts as vectors, and use Euclidean distance as a measure of similarity. This alone is not sufficient, as wrong decisions inevitably happen and their effect is only augmented by considering portions of the physical cluster detached from the current position by a node not in the neighborhood of the candidate (Figure 7). For this reason, we stop the counting at occupied positions (placed candidates) and introduce counting of parents and children among the occupied (placed) neighbors of the current position (candidate), as well as inputs shared with them, to patch the pieces back together. We can think of the first type of counting as a way to foresee the future and the second one as a connection to the past. Further metrics are used to address the differences between input sharing patterns.

We evaluate these functions for each candidate-position pair and form a linear combination of the values, such that a lower value indicates higher similarity, populate a score matrix, and at each step pick its lowest element. If the corresponding placement results in a routable packing, we recompute the score matrix, excluding the row and the column intersecting at the selected entry, and proceed to the next step. Otherwise, we mark the selected entry and choose the next smallest one. The process terminates when either the matrix is empty, or all entries are marked. While this approach could be infeasible for a packing algorithm that considers the whole circuit at once, for a partitioning-based one it is rather efficient.

After packing all clusters, we collect the unplaced LUTs, reconstruct the circuit subgraph induced by them and their parents and children, and repartition it. We keep alternating between partitioning and packing until all the LUTs get placed. To check routability, we construct the routing-resource graph in much the same way as AAPack and use an ILP solver to search for a routing solution.

V. Checking feasibility

Since delay measurements are susceptible to CAD tool related noise [22] and our packer currently does not prioritize delay optimization, setting a threshold on delay could obscure the results of architectural search. Hence, without loss of generality to our methodology, we do not to impose a delay-based threshold at present. This decision only removes one of the necessary conditions for an architecture to be feasible and will not result in elimination of any architecture that would not have been eliminated otherwise (i.e., it is conservative).

While it is generally true that high utilization may not translate to low area consumption, it is also an undeniable fact that too low a utilization inevitably translates to large area. Hence, we use post-packing utilization as a measure of feasibility, allowing us to skip the place and route process.

The intuition that the final utilization is predominantly determined by the success of packing in the first partitioning phase was confirmed in our preliminary experiments on kernel-induced architectures and we chose to measure utilization without performing repartitions. This makes all the considered clusters fully independent, whereas previously, those in future phases depended on those in prior ones. A question of whether all clusters need to be packed or a small sample of them may be sufficient to assess the final utilization now arises naturally. Somewhat arbitrarily, we chose the sample size of 5, which quickly ruled out random sampling as an option, due to high probability of making large errors.

Using the same sample for all architectures potentially reduces experimental noise, but also allows only for basing the bias on logical cluster characteristics that do not favour

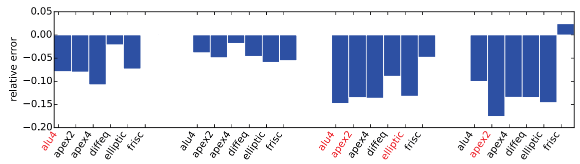


Fig. 8: Relative Errors of Predicting the Number of Used Clusters Based on Sampling Logical Clusters from the First Partitioning Phase, versus the Actual Number Used in Complete Packing, for Four Architectures of Varying Flexibility. Prediction is higher than actual value only in one case. Red labels indicate that the complete run would prune the architecture and sampling prediction would not.

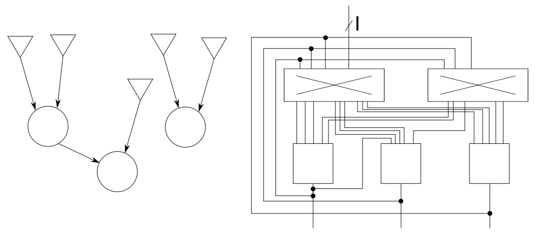


Fig. 9: Forcing Some of the Inputs to Be Provided by Feedbacks Only

any single architecture. We observed that clusters with a large number of LUTs with all K external inputs generally achieve the least utilization and chose the distribution of external in-degrees to be our metric. We arrange the degree counts in a vector with components indicating degrees $\in [0, K]$ and pick 5 clusters at the minimum Euclidean distance from the component-wise average.

Sampling in this manner inevitably removes conservativeness, but still provides reasonable predictions, as indicated by the plots of Figure 8. The speedup it brings more than justifies the error introduced. Moreover, it enables assessing the utilization on much larger benchmarks, when that is of concern.

Finally, in our experiments on kernel-induced structures, we observed that only the first 7 of a subset of the 20 largest *MCNC* [26] benchmarks used for feasibility checks[5] eliminated any structure. Moreover, the first one that was being applied—*alu4*—was responsible for pruning 73% of the eliminated structures. Hence, without loss of generality, we chose to reduce the benchmark set to that single element. As will be seen later, results we obtain on it generalize very well to other benchmarks, even if they are larger and newer.

VI. Experimental results

In our experiments, we consider cluster architectures composed of four copies of a particular structure, connected by a fully populated crossbar. Achieving $\geq 80\%$ utilization on the *alu4* MCNC benchmark, measured by sampling, as described in Section V, is used to define feasibility.

We enumerate input sharing on each kernel independently. Prior to starting, we force some of the LUT inputs to be provided by feedbacks only, through enumeration of patterns of inputs with a single fanout (Figure 9). That increases possibilities for further hardening of connections between structures, to create hierarchical structures of larger size.

This first search phase leaves us with $14,139$ surviving structures, with the number of forced feedback inputs $\in [0, 11]$.

[5]We chose a subset of MCNC benchmarks because of their small size. A comprehensive recent study [27] relieved us of worries that they are outdated.

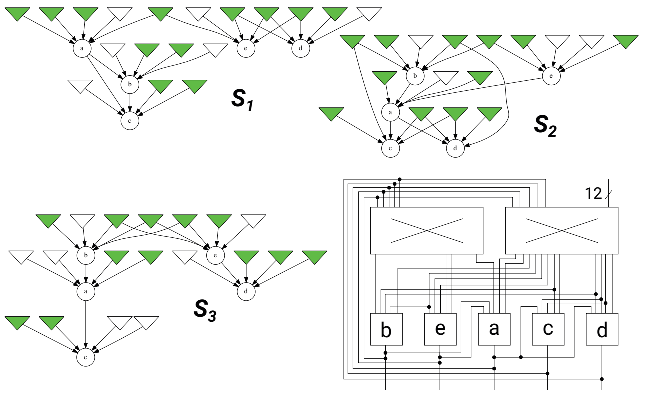


Fig. 10: Three of the 261 Structures with 12 External Inputs (Green). The white inputs are feedback-only. Schematic shows S_2. In the experiments, crossbars were common to all four copies of the particular structure.

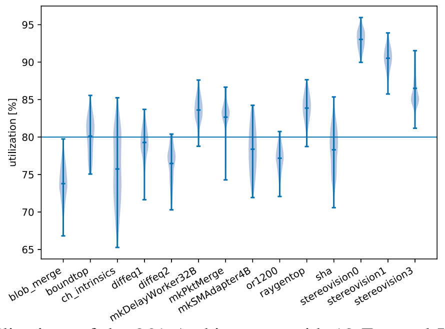


Fig. 11: Utilizations of the 261 Architectures with 12 External Inputs, for a Subset of *VTR* Benchmarks

We proceed with enumerating sharing patterns on remaining inputs, starting from maximal structures of the first phase, where by maximal we mean that forcing a single additional input to be feedback-only would push the structure below the threshold. There are $2,398$ such maximal structures, which we again process independently.

After completing the second search phase, we obtain $14,691$ new surviving structures, with the number of external inputs $\in [12, 18]$. Particularly interesting are the structures with 12 external inputs. A 20 LUT cluster they constitute needs only 48 inputs, which is even less than the 52 normally provided to classical architectures of the same size [2]. Note that this reduction was achieved purely statically, without using a crossbar.

We find 261 such architectures, three of which are shown in Figure 10. They contain some of the well studied interconnect patterns, such as feed-forward loops [25], cascades [20], pairs of LUTs with optimal number of shared inputs [14], etc. Combinations of these patterns that we can see here would perhaps not immediately come to our minds, which demonstrates the value in using the enumerative approach.

To see how these structures perform on more appropriate benchmarks, we took the *VTR* set [18], excluding the five biggest circuits due to runtime reasons, and packed it in each of the 261 selected architectures, using three different partitioner seeds. The plot of Figure 11 shows obtained utilizations averaged over the three seeds. As we can see, conclusions based on sampling the *alu4* benchmark generalize remarkably well to other, larger and newer benchmarks.

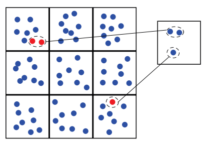


Fig. 12: Illustration of Merging Issues During Repartitioning. Red nodes are the LUTs that were not successfully packed during the first phase. Since the connections to their initial clusters are already cut, grouping them together in the next partitioning phase does not increase the cut. This, however, creates serious problems for placement and causes severe impact on delay.

We also ran the complete CAD flow on these architectures, taking the global routing architecture from Luu's *EArch* [16] and keeping all delays the same. *AAPack* was used for packing of hard IP blocks while *VPR* [18] was used for placement and routing. For comparison, a simple architecture of twenty 5-LUTs and 52 cluster inputs, with a fully populated crossbar, also derived from *EArch*, with delays unchanged, was packed with AAPack, using default settings. The geomean post-routing delay over all benchmarks was between 18 and 29% higher than the reference, while the worst case even surpassed 50% for some architectures. By analyzing critical paths, we came to a conclusion that this was mostly due to the partitioner grouping together near-critical nodes from distant clusters, after they have not been successfully packed in the previous partitioning phase, making decent placement impossible (Figure 12). Together with appropriate delay and area modeling, this is something to consider in the future.

All the structure enumeration and containment graph construction, along with all necessary pre- and post-processing and equivalence checking was implemented in Python. The packer itself was implemented in C, using *glpk* [9] with a timeout of one second as the ILP solver. On a Xeon E5-2680 v3 based server with 256 gigabytes of RAM, the first search phase completed in six hours, while the second one took four and a half.

VII. Conclusions

We presented efficient techniques for the exploration of large design spaces spanned by structures of LUTs with hardened interconnect patterns. These techniques leverage the structure of the design space and make it possible to select only those points that may perform better than any chosen reference. We demonstrated the usefulness of the techniques by narrowing the design space spanned by five 5-LUT structures from hundreds of millions to a set of 261 that need only 12 external inputs, while achieving $\geq 80\%$ utilization on a set of standard benchmarks. We learned an important lesson by doing this exploration: it is not the size of the space we need to search that should intimidate us, but its lack of structure. We believe that the presented search techniques are general and that they could also be used for exploration of architectural components other than LUT structures. Apart from the search techniques, we also introduced a packing algorithm that takes into account the hardened connections. To be able to bring more meaningful conclusions about the value of hardening complex interconnect patterns, we must equip the packer with sound delay optimization heuristics it currently lacks and appropriately characterize the structures in terms of area and delay.

References

[1] E. Ahmed. The effect of logic block granularity on deep-submicron FPGA performance and density. Master thesis, University of Toronto, Toronto, 2001.

[2] Altera Corporation. *Stratix II Device Handbook, vols. 1 and 2*. http://www.altera.com/literature/.

[3] J. H. Anderson, Q. Wang, and C. Ravishankar. Raising FPGA logic density through synthesis-inspired architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 20(3):537–550, March 2012.

[4] V. Betz and J. Rose. Using architectural "families" to increase FPGA speed and density. In *Proceedings of the Third International ACM Symposium on Field-Programmable Gate Arrays,FPGA 1995, Monterey, California, USA, February 12-14, 1995*, pages 10–16, 1995.

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

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

[7] K. C. K. Chung. *Architecture and Synthesis of Field-Programmable Gate Arrays with Hard-wired Connections*. Ph.D. thesis, University of Toronto, Toronto, 1994.

[8] W. Feng, J. W. Greene, and A. Mishchenko. Improving FPGA performance with a S44 LUT structure. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2018, Monterey, CA, USA, February 25-27, 2018*, pages 61–66, 2018.

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

[10] C. Godsil and G. F. Royle. *Algebraic graph theory*. Graduate Texts in Mathematics. Springer, 1 edition, 2001.

[11] L. A. Goldberg. *Efficient Algorithms for Listing Combinatorial Structures*. Ph.D. thesis, University of Edinburgh, Edinburgh, 1991.

[12] H.-C. Hsieh, W. S. Carter, J. Ja, E. Cheung, S. Schreifels, C. Erickson, P. Freidin, L. Tinkey, and R. Kanazawa. Third-generation architecture boosts speed and density of field-programmable gate arrays. In *Proceedings of the IEEE Conference on Custom Integrated Circuits*, pages 31–38, Boston, Mass., May 1990.

[13] M. Hutton, J. Schleicher, D. Lewis, B. Pedersen, R. Yuan, S. Kaptanoglu, G. Baeckler, B. Ratchev, K. Padalia, M. Bourgeault, A. Lee, H. Kim, and R. Saini. Improving FPGA performance and area using an adaptive logic module. In *Proceedings of the 14th International Conference on Field Programmable Logic and Application*, pages 135–144, 2004.

[14] Z. Jiang, C. Y. Lin, L. Yang, F. Wang, and H. Yang. Exploring architecture parameters for dual-output LUT based FPGAs. In *24th International Conference on Field Programmable Logic and Applications, FPL 2014, Munich, Germany, 2-4 September, 2014*, pages 1–6, 2014.

[15] C. Kingsford and R. Patro. Global network alignment using multiscale spectral signatures. *Bioinformatics*, 28(23):3105–3114, 10 2012.

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

[17] J. Luu, J. H. Anderson, and J. Rose. Architecture description and packing for logic blocks with hierarchy, modes and complex interconnect. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 227–36, Monterey, Calif., Feb. 2011.

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

[19] B. D. McKay and A. Piperno. Practical graph isomorphism, II. *Journal of Symbolic Computation*, 60(0):94–112, 2014.

[20] A. Mishchenko. LUT structure for delay: Cluster or cascade? In *Proceedings of the 21st International Workshop on Logic and Synthesis*, Berkeley, Calif., 2012.

[21] S. Ray, A. Mishchenko, N. Eén, R. K. Brayton, S. Jang, and C. Chen. Mapping into LUT structures. In *2012 Design, Automation & Test in Europe Conference & Exhibition, DATE 2012, Dresden, Germany, March 12-16, 2012*, pages 1579–1584, 2012.

[22] R. Rubin and A. DeHon. 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, FPGA 2011, Monterey, California, USA, February 27, March 1, 2011*, pages 173–176, 2011.

[23] D. Vercruyce, E. Vansteenkiste, and D. Stroobandt. How preserving circuit design hierarchy during FPGA packing leads to better performance. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 37(3):629–642, 2018.

[24] G. Wang, S. Sivaswamy, C. Ababei, K. Bazargan, R. Kastner, and E. Bozorgzadeh. Statistical analysis and design of HARP FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-25(10):2088–2102, 2006.

[25] J. Wang and G. Provan. On motifs and functional modules in complex networks. In *2009 IEEE Toronto International Conference Science and Technology for Humanity (TIC-STH)*, pages 78–82, Sep. 2009.

[26] S. Yang. Logic synthesis and optimization benchmarks user guide, version 3.0. Technical report, Microelectronics Center of North Carolina, Research Triangle Park, N.C., Jan. 1991.

[27] G. Zgheib and P. Ienne. Evaluating FPGA clusters under wide ranges of design parameters. In *Proceedings of the 27th International Conference on Field-Programmable Logic and Applications*, Ghent, Belgium, Sept. 2017.