

# FRESCO: Efficient Subgraph Enumeration for Scalable Clustering in Heterogeneous CGRAs

Louis Coulon  
EPFL

Lausanne, Switzerland  
louis.coulon@epfl.ch

Adham Ragab  
University of Toronto

Toronto, Canada  
adham.ragab@mail.utoronto.ca

Jason Anderson  
University of Toronto

Toronto, Canada  
janders@ece.utoronto.ca

Mirjana Stojilović  
EPFL

Lausanne, Switzerland  
mirjana.stojilovic@epfl.ch

Paolo Ienne  
EPFL

Lausanne, Switzerland  
paolo.ienne@epfl.ch

**Abstract**—In recent years, there has been a trend towards reconfigurable fabrics at the intersection between *field-programmable gate arrays* (FPGAs) and *coarse-grained reconfigurable arrays* (CGRAs): using FPGA-like interconnect but word-based and built around coarse-grained primitives. These architectures often employ complex clusters with far more heterogeneous resources than FPGAs or typical CGRAs—sometimes over a hundred primitives, most of which are bypassable. As a result, clustering, the problem of covering the application netlist with architecture clusters, is a key challenge for design tools targeting these fabrics. Clustering is analogous to the instruction selection problem in CISC architectures, albeit with orders of magnitude more complex “instructions”. In this work, we propose a two-phase, architecture-agnostic clustering algorithm that scales to highly complex architecture clusters. The first phase enumerates potential cluster matches in the application netlist using a strategy based on an abstract decision tree. The second phase selects a cover from the enumerated matches. We show that our algorithm effectively prunes the search space for complex clusters, scales well to circuits composed of many clusters, and achieves better clustering quality than CLUMAP, a state-of-the-art CGRA clustering algorithm, for the simple cases that CLUMAP can handle.

## I. INTRODUCTION

The most flexible reconfigurable fabrics are *field-programmable gate arrays* (FPGAs) [1], reconfigurable at the bit level. They are often used with *high-level synthesis* (HLS) compilers that allow complex software programs to be synthesized into hardware circuits and implemented on such fabrics, provided that enough hardware resources are available [2]–[5]. However, unless an application benefits from bit-level reconfigurability [6], FPGAs typically suffer from a severe performance and cost overhead compared to *application-specific integrated circuits* (ASICs) [7].

*Coarse-grained reconfigurable arrays* (CGRAs) have been proposed as denser word-level reconfigurable arrays that can achieve closer to ASIC performance [8]. Traditionally, CGRAs were highly restrictive architectures only capable of running feedforward programs consisting of perfectly nested loops [9]. The limited amount of routing resources created dependencies between placement and routing, and initial CGRAs required a unified mapping problem formulation [10], [11]. However, there has been a trend towards increasingly flexible and heterogeneous architectures in recent years. CGRAs can now contain *switch blocks* (SBs) [8], [12], elastic primitives [13]–[15], some control flow operations [16], or even rely on FPGA-like interconnects to support the mapping of circuits generated by particular HLS compilers [15]. These more flexible CGRAs expose a considerably larger design space to mapping tools that require new algorithms to utilize the architecture fully [16]–[18].

One of the central challenges for compilers targeting heterogeneous CGRAs is clustering—the task of mapping or covering an application netlist with architecture clusters so that each node in the netlist fits within a compatible physical cluster. Figure 1a illustrates a simple example cluster (often referred to as a processing element in CGRA

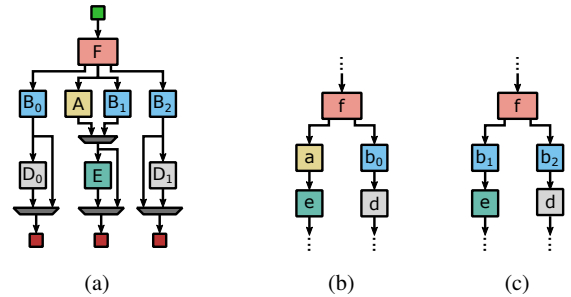


Fig. 1: (a) An example cluster featuring bypassable ( $D_0$ ,  $D_1$ , and  $E$ ), mutually exclusive ( $A$  and  $B_1$ ), and primitives that can function as identities (e.g.,  $F$ ,  $B_0$ ,  $B_1$ , and  $B_2$ ). Gray multiplexers implement various cluster configurations. The green and red squares denote cluster input and outputs, respectively. (b–c) Two example circuit subgraphs that can be mapped onto the cluster shown in (a).

literature). This example highlights three common architectural features: (i) Some primitives can be bypassed to increase mapping flexibility. (ii) Certain primitives can act as identity functions under appropriate configurations (e.g., an adder with a zero operand). (iii) Some primitives are mutually exclusive, meaning they share internal resources such as configuration multiplexers and cannot be used simultaneously. Figures 1b–1c show two application subgraphs that can be mapped onto this cluster. In these figures, nodes labeled with lowercase letters (e.g.,  $f$ ) correspond to primitives labeled with the matching uppercase letters (e.g.,  $F$ ); indices distinguish different instances. A capable clustering algorithm must understand the constraints imposed by the above architectural features. For example,  $A$  and  $B_1$  are mutually exclusive;  $b_1$  must be mapped to  $B_1$ , given that it is followed by  $e$ ;  $b_0$  and  $b_2$  can only be mapped to  $B_0$  or  $B_2$ , because they are followed by  $d$ . Failure to respect such constraints leads to suboptimal clustering, requiring more clusters than necessary and ultimately degrading performance. While the example in Figure 1 is intentionally simple, it reflects real-world structures [15], where clusters can contain over a hundred primitives. Such architectures would be sparsely utilized without an appropriate clustering algorithm, negatively impacting critical paths and area.

### A. The Main Challenge

Even for the simple motivating example in Figure 1, plenty of application subgraphs are mappable on the given cluster. To get a sense of the quick combinatorial growth, let us count the possibilities. First, we denote by  $t_0$  the subset of the architecture consisting only of  $B_0$  and  $D_0$ , by  $t_1$  the one consisting of only  $A$ ,  $B_1$ , and  $E$ , and by  $t_2$  the one consisting of only  $B_2$  and  $D_1$ . Then, we note

that  $t_0$  accepts three possible mappings:  $\{\{B_0\}, \{D_0\}, \{B_0, D_0\}\}$ , assuming that  $B_0$  can be used as an identity (route-through) element. The same applies to  $t_2$ . Similarly,  $t_1$  accepts five possible mappings:  $\{\{A\}, \{B_1\}, \{E\}, \{A, E\}, \{B_1, E\}\}$ . Then, we enumerate the total number of subgraphs varying the number of used outputs of  $F$ . With one used output, we can utilize primitives in either  $t_0$ ,  $t_1$ , or  $t_2$ , leading to  $3 + 3 + 5 = 11$  application subgraphs. With two used outputs, we can use  $t_0$  and  $t_2$ , giving  $3 \times 3 = 9$  subgraphs, or either  $t_0$  or  $t_1$  together with  $t_2$ , for the additional  $2 \times 3 \times 5 = 30$  possible subgraphs. With all three outputs used, there are  $3 \times 5 \times 3 = 45$  subgraphs. Finally, we can map  $F$  primitive independently, giving us eight additional subgraphs. Overall, we get to  $11 + 30 + 45 + 8 = 94$  possible combinations, including many unique ones which may all be valuable for clustering a particular application graph.

Ultimately, the number of application circuit subgraphs mappable on a given architecture cluster depends on the cluster topology, the nature of the primitives, and the number and location of the configuration multiplexers. A good clustering algorithm must explore all unique possibilities: not considering all mappable application subgraphs would result in less dense utilization of the clusters and consequently impact the quality of the resulting clustering. Unfortunately, naively enumerating all possible subgraphs to select a cover would not be computationally tractable—even simple clusters with only eight primitives (Figure 1) already have many mappable application subgraphs. Clusters in modern CGRAs may contain significantly more primitives, even above a hundred [15]. Enabling efficient application mapping on architectural clusters of such and higher complexity is the key motivation for this work.

### B. A New Clustering Algorithm

In this paper, we present FRESKO<sup>1</sup>, a novel clustering algorithm that efficiently enumerates valid mappings of application nodes onto architectural primitives. The first phase of our algorithm identifies feasible matches between architectural clusters and the application netlist, centered around selected root nodes in the application graph. This is achieved through a decision tree that explores all valid placements, effectively pruning the search space by leveraging the clusters’ structural features. In the second phase, a simple greedy heuristic selects a cover from the enumerated matches—a set of nonoverlapping cluster mappings that collectively implement the entire netlist. These two phases are described in Section IV. To determine whether a particular cluster can implement a given set of circuit nodes, we must verify that it has sufficient routing resources to connect the required components. To this end, we model clusters using routing resource graphs and formulate the internal routing task as an *integer linear programming* (ILP) problem, as described in Section V. We assess the scalability and effectiveness of our approach through a comprehensive evaluation presented in Section VI. Our results show that the proposed clustering algorithm effectively prunes the search space for complex clusters, scales well to circuits composed of many clusters, and achieves better clustering quality than CLUMAP [17], a state-of-the-art CGRA clustering algorithm, for the simple cases that CLUMAP can handle.

## II. CLUSTERING PROBLEM STATEMENT

An *application circuit* is a graph  $G = (V_G, E_G)$  where the vertices  $V_G$  represent computational primitives and the edges  $E_G$  represent connections between them. An architecture is defined by a set of different *cluster types*  $Cl = \{Cl_0, Cl_1, \dots, Cl_{N-1}\}$ , where  $N$  is

the number of available cluster types. Here,  $Cl_i = (V_{Cl_i}, E_{Cl_i})$  and  $0 \leq i < N$ . Clusters represent the types of physical resources available in the architecture: they are sets of primitives interconnected in predefined but often configurable ways. *Clustering* is the problem of finding a particular cover  $C$  of the application  $G$  using the available architectural clusters. We say a *cover*  $C$  of  $G$  is a collection of subgraphs  $C = \{G_0, G_1, \dots, G_{k-1}\}$  such that (1) the vertex sets form a partition of  $V_G$  (i.e.,  $V_G = \bigsqcup_{j=0}^{k-1} V_j$ ) and (2) each subgraph  $G_j = (V_j, E_j)$  is induced (i.e.,  $E_j = \{(u, v) \in E \mid u, v \in V_j\}$ ).

A cover is *valid* if each  $G_i$  can be successfully mapped to an architecture cluster of type  $Cl_i$ ; we say that  $Cl_j$  *implements*  $G_i$  and denote this relation as  $Cl_j \sim G_i$ ; we will detail in Section V what this exactly entails. Globally, the following condition must hold:

$$\forall i \in \{0, \dots, k-1\}, \exists Cl_j : Cl_j \sim G_i. \quad (1)$$

We aim to find a cover composed of the minimum number of induced subgraphs, that is, the solution using the minimum number of clusters in the architecture.

Therefore, our clustering goal consists of two subproblems: (1) finding a good partitioning  $C$  and (2) given a subgraph  $G_i$  of  $G$ , verifying that  $Cl_j \sim G_i$  holds. After reviewing related work, we will address these two problems in the subsequent sections.

## III. RELATED WORK

### A. Packing in FPGAs

Clustering is analogous to the packing problem in FPGAs. Packing groups primitives with hard-wired architectural connections—such as carry-chains, local links between *lookup tables* (LUTs) and *flip-flops* (FFs), or *digital signal processing* (DSP) blocks. To handle programmable routing within *configurable logic blocks* (CLBs), packing algorithms also group timing-related primitives (orthogonal to this work). The state-of-the-art FPGA packing algorithm appears in the open-source Versatile Place and Route (VPR) framework [20]. It uses *packing patterns*—annotations in the architecture file that enumerate supported application subgraphs. In this setup, each configuration multiplexer increases the required packing patterns multiplicatively. For instance, the architecture in Figure 1a would require 94 packing patterns. Consequently, FPGA packing algorithms do not scale well to more complex clusters.

### B. Mapping in CGRAs

*Mapping* is the catch-all term for the CGRA CAD flow, where the application circuit graph  $G$  (see Section II) is mapped onto the CGRA, producing a configuration bitstream. The CGRA is also typically represented as a graph, and the cluster graphs, e.g.,  $Cl_i$ , can be considered subgraphs of the overall CGRA device model graph. Many previously published CGRAs (e.g., [11], [12]) contain simple clusters, comprising an *arithmetic logic unit* (ALU), registers, and a constant unit. In such architectures, it was often possible to map  $G$  directly to the CGRA in a *flat* manner, without a clustering step. Mapping subtasks include: (i) scheduling the operations of  $G$  in time (only in nonelastic CGRA architectures); (ii) placing the nodes of  $G$  into corresponding CGRA device primitives; (iii) routing the edges of  $G$  using the CGRA interconnect. A wide range of approaches have been proposed for mapping, including formulating the entire problem as an ILP instance [10], genetic algorithm-based approaches [21], decision diagram-based approaches [18], and others [22], [23].

Recent CGRAs have more complex clusters, necessitating a clustering step before placement and routing to create a legal mapping. A recent CGRA mapper, CLUMAP [17], incorporates clustering and is thus most comparable to this work. CLUMAP’s clustering step

<sup>1</sup>FRESKO is included in the open-source FRIDA compiler [19].

groups primitives of  $G$  that are then kept together during placement. It first selects a seed vertex  $v$  from  $G$  to initiate a new cluster, then grows the cluster based on vertices connected to  $v$  or those already in the cluster, checking routing legality at each addition. The approach is sensitive to the order in which vertices are considered for cluster addition, thereby impacting quality, as demonstrated in the experimental results section of this paper (Section VI-C). CLUMAP uses simulated annealing for placement and a negotiated-congestion approach for routing, inspired by FPGAs [24].

### C. Instruction Selection and Technology Mapping

Instruction selection is another clustering problem that maps abstract compiler operations to the operations supported by the target architecture. Technology mapping is the analogous problem in hardware synthesis, where one maps technology-independent circuit representations to technology-specific ones. Because of their structural similarity, both problems use similar algorithms. VF2 [25] is among the most advanced, designed for pattern matching in general graphs. However, it only supports exact matches and cannot handle architectures with configuration multiplexers. Other approaches decompose graphs into forests of trees [26], which works well only on simple graphs. Finally, domain-specific algorithms, such as those optimized for a particular technology—for example, FlowMap [27], which minimizes the depth of circuits composed of lookup tables—do not apply to this work.

## IV. ENUMERATION AND SELECTION

As already evoked, we formulate our search for a minimal cover in two parts, as is a common approach in this type of problem [27]: (i) enumerating valid mappings of subgraphs of  $G$  to the available cluster types  $Cl$  and (ii) selecting the mappings that result in the cover with the smallest number of subgraphs.

We first describe how to enumerate candidate mappings corresponding to a single cluster  $Cl_j$  centered at an arbitrary application node  $n \in V_G$ . To do so, we want to explore the space consisting of all sets  $S \subseteq (V_G \times V_{Cl_j})$ , each representing a possible mapping from a subgraph of  $G$  to the cluster  $Cl_j$ , and containing  $n$ . In this section, we set aside the problem of determining whether a particular mapping  $S$  is *valid*—that is, whether  $Cl_j \sim G_i$  holds. We abstract this check via the relation  $R(S)$ , which we assume can be evaluated:  $R(S)$  is true if the mapping  $S$  allows  $Cl_j$  to implement  $G_i$ . The details of how  $R(S)$  is implemented will be discussed in Section V. Our focus here is on the core problem of efficiently enumerating valid mappings using decision trees. We conclude the section by outlining the limitations of our algorithm and describing how to construct a cover for the application circuit from the identified cluster mappings.

### A. Building the Decision Tree

To navigate the state space of all possible  $S$ , we define *unitary state transitions*, which incrementally extend a partial mapping by adding a single pair—that is, a transition from  $S$  to  $S'$  where  $S' = S \cup \{(u, v)\}$  with  $u \in V_G$  and  $v \in V_{Cl_j}$ . Starting from an initial state  $S_0 = \{(u, r)\}$ , our goal is to explore all reachable states through sequences of such unitary transitions. We perform this exploration using a decision tree that, given any current state  $S$ , enumerates all valid unitary transitions to new states  $S'$ . Figure 2 illustrates an example of such a decision tree, which maps the circuit from Figure 1b onto the architecture cluster shown in Figure 1a. In this section, we describe how to effectively prune the naive tree (Figure 2a) to a significantly smaller and more manageable one (Figure 2b), using structural and routing constraints. The traversal

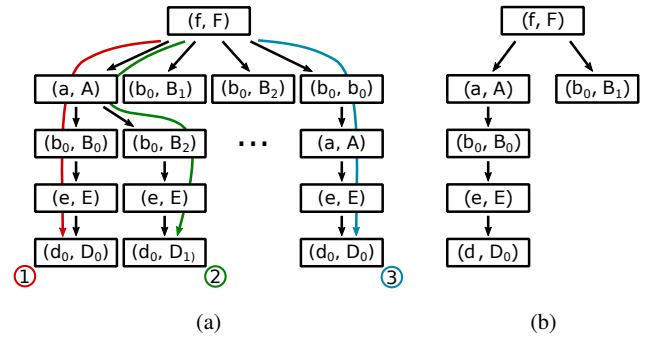


Fig. 2: (a) The complete decision tree enumerating mappings of the application circuit in Figure 1b on the architecture cluster of Figure 1a. We highlight three equivalent mappings: (1) and (3) are enumerated in a different order but are identical, while (1) and (2) map  $b_0$  and  $d$  to symmetric locations. Mutual exclusivity (Section IV-B) guarantees the enumeration of at most one of (1) and (3) and symmetry pruning the enumeration of at most one of (1) and (2) (Section IV-D). (b) The pruned decision tree.

procedure of the decision tree is detailed in Algorithm 1, which is a recursive algorithm. It begins with the base case, where the current state is  $S_0$ . Each recursive call corresponds to a unitary transition that extends the current state  $S$ . At each step, the algorithm computes the set of valid next states, constructs the corresponding incompatibility graph, and selects a clique to explore next, as explained in detail below. When no further element can be added to the current mapping, the algorithm returns the current  $S$  as a candidate mapping. All candidate mappings are aggregated across calls corresponding to parallel branches. Note that the algorithm only returns mappings corresponding to leaves of the decision tree: we refer to these mappings as *maximal*, since the corresponding subgraphs of the application can no longer be extended.

For simplicity, we denote by  $M_G(S)$  the set of nodes of  $G$  appearing in  $S$ , and by  $M_{Cl_j}(S)$  the set of cluster nodes from  $Cl_j$  used in  $S$ . The property  $R(S)$ , which we will detail in Section V, holds if  $M_{Cl_j}(S) \sim M_G(S)$ . We also define  $N_G(S)$  as the set of neighbors in  $G$  (not already in  $S$ ) of the nodes in  $M_G(S)$ ;  $N_{Cl_j}(S)$  is defined analogously:

$$N_G(S) = \{v \in V_G \setminus M_G(S) \mid \exists u \in M_G(S), (u, v) \in E_G \vee (v, u) \in E_G\}. \quad (2)$$

We can now define  $P_v(S, G, Cl_j)$ , the set of valid candidate map-

---

#### Algorithm 1: Enumerate All Mappings from a Root

---

**input** :  $G$ ,  $Cl_j$ , and the root mapping  $S_0 = \{(u, r)\}$   
**output**: The set of all mappings rooted at  $S_0$   
**return** enumerateMappings( $G, Cl_j, \{u, r\}$ )  
**function** enumerateMappings( $G, Cl_j, S$ ):  
  **if** isEmpty( $P_v(S, G, Cl_j)$ ) **then**  
  | **return**  $\{S\}$   
  **else**  
  |  $I_g \leftarrow$  buildIncompatibilityGraph( $P_v(S, G, Cl_j)$ )  
  |  $M \leftarrow \emptyset$   
  | **for**  $q \in$  getAnyMaximalClique( $I_g$ ) **do**  
  | |  $M \leftarrow M \cup$  enumerateMappings( $G, Cl_j, S \cup q$ )  
  | **return**  $M$

---

pings resulting from unitary transitions in the decision tree:

$$P_v(S, G, Cl_j) = \{m = (u, v) \mid u \in N_G(S), v \in N_{Cl_j}(S), R(S \cup \{m\})\}. \quad (3)$$

However, we note that simply exploring all candidate mapping pairs in  $P_v(S, G, Cl_j)$  would result in the enumeration of many duplicated mappings. For example, in Figure 2, consider  $S_0 = \{(f, F)\}$ : exploring all candidate mapping pairs in  $P_v(S_0) = \{(a, A), (b_0, B_1), (b_0, B_2), (b_0, B_0)\}$  would result in the enumeration of two completely equivalent mappings, denoted by (1) and (3) in the figure. In general, if  $S_j$  is reached through the sequence of state transitions  $T(S_j) = S_0 \rightarrow S_1 \dots \rightarrow S_j$  and  $S_k$  is reached through  $T(S_k) = S_0 \rightarrow S'_1 \dots \rightarrow S_k$ , we want to ensure that our decision tree never reaches any two states  $S_j = S_k$  through two distinct paths  $T(S_j) \neq T(S_k)$ .

### B. Preventing Duplicated Enumeration Through Mutual Exclusivity

Naturally, some mappings are mutually exclusive, as a circuit primitive can only be mapped once within a cluster (e.g., in Figure 1a,  $b_0$  may be mapped to either  $B_0$  or  $B_2$ , but not both). Additionally, a crucial aspect of our problem formulation—essential for accurately modeling meaningful configurable clusters—is that some cluster primitives cannot be used simultaneously. For example, in Figure 1a, the configuration multiplexer enforces that only one of  $A$  and  $B_1$  may be active at a time. This mutual exclusivity implies that certain candidate mapping pairs in  $P_v(S, G, Cl_j)$  are incompatible: selecting one for the next state prevents the other from ever becoming a valid candidate later on. Ignoring these constraints would require exploring all candidates indiscriminately, often leading to redundant paths and duplicated enumeration. Recognizing and exploiting mutual exclusivity is central to our strategy for pruning the decision tree and is a distinctive feature of our approach.

Formally, we say that states  $S_0$  and  $S_1$  are *mutually exclusive* if each represents a valid mapping on its own, but their union does not:

$$P_{\text{excl}}(S_0, S_1) = R(S_0) \wedge R(S_1) \wedge \neg R(S_0 \cup S_1). \quad (4)$$

Since  $P_{\text{excl}}$  is a pairwise property, we can construct an *incompatibility graph* over the elements of  $P_v(S)$ . We denote this graph by  $I_g(S) = (V_{I_g}(S), E_{I_g}(S))$  and define it as follows:

$$V_{I_g}(S) = P_v(S), \quad (5)$$

$$E_{I_g}(S) = \{(u, v) \mid u, v \in P_v(S), P_{\text{excl}}(S \cup \{u\}, S \cup \{v\})\}. \quad (6)$$

For instance, for  $S = (f, F)$ , the corresponding incompatibility graph  $I_g(S)$  is shown in Figure 3. To avoid duplicated enumeration, we do not explore all candidate mappings as children of a decision node. Instead, we restrict exploration to a subset of pairwise mutually exclusive mappings—that is, a clique in the incompatibility graph. Intuitively, any candidate left out of the selected clique must be

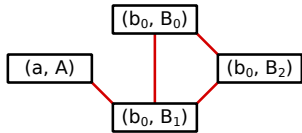


Fig. 3: Incompatibility graph computing in the decision tree of Figure 2 to find the children of  $(f, F)$ .  $(b_0, B_0)$ ,  $(b_0, B_1)$ ,  $(b_0, B_2)$  are mutually exclusive because they use the same circuit node and  $(b_0, B_1)$  and  $(a, A)$  are mutually exclusive because they need to use the same multiplexer to exit the cluster.

compatible with at least one of the chosen mappings and will thus be revisited in a later branch of the decision tree.

Note that we are not interested in finding a maximum clique in  $I_g$ , but rather any clique centered at an arbitrary vertex. The only requirement is that this clique be *maximal*—that is, it should not be strictly contained within a larger clique. Such maximal cliques are easy to construct: starting from a single vertex, we iteratively add neighboring vertices that maintain the clique property.

**Theorem 1.** For any two states  $S_j$  and  $S_k$  containing the same mappings ( $S_j = S_k$ ) and constructed with the above decision tree,  $T(S_j) = T(S_k)$ .

*Proof.* Assume two states  $S_j$  and  $S_k$  with  $S_j = S_k$  but  $T(S_j) \neq T(S_k)$ . The two paths  $T(S_j)$  and  $T(S_k)$  must have a common prefix since they originate from the root of the decision tree. They must also diverge at least once; otherwise, they would be identical. Let  $S_\ell$  be the last common node on the paths, and let  $S_{\ell_j}$  and  $S_{\ell_k}$  be the first diverging nodes on  $T(S_j)$  and  $T(S_k)$ , respectively. By construction,  $P_{\text{excl}}(S_{\ell_j}, S_{\ell_k})$  holds and thus  $S_j$  and  $S_k$  must differ in at least  $S_{\ell_j}$  and  $S_{\ell_k}$ —a contradiction. Therefore,  $T(S_j) = T(S_k)$ .  $\square$

### C. Limitations

While our approach guarantees that no state is enumerated more than once, the pairwise definition of the mutual exclusivity property prevents us from exploring architectures allowing  $k$  out of  $n$  cluster primitives to be used simultaneously, such as partial crossbar interconnects found in some FPGAs, for example. Generalizing our approach to handle such multi-way exclusivity is left as future work.

Additionally, checking the property  $R(S)$  for each state  $S$  in the enumeration occasionally necessitates enriching clusters with additional configuration multiplexers to ensure internal signals can reach the cluster I/Os. Fortunately, this enhancement is also beneficial for improving the cluster generality. Failing to implement such multiplexers would render some valid solutions unreachable in the enumeration, but never produce invalid ones. On the other hand, neglecting to check the validity  $R(S)$  of every partial state  $S$  would risk enumerating invalid states, leading to the violation of the guarantees outlined in the previous section. For instance, in Figure 4a, the pairs  $(a, A)$ ,  $(b, B)$  and  $(a, A)$ ,  $(c_0, C_0)$  are mutually exclusive due to the configuration multiplexer. Accurately identifying this exclusivity requires considering the routability of the signals to the I/Os.

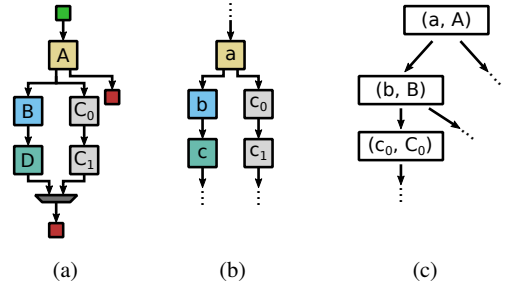


Fig. 4: (a) An example cluster with two mutually exclusive sets of bypassable primitives:  $B$  cannot be mapped at the same time as  $C_0$  and  $C_1$ . (b) An example application graph that is partially mappable onto the cluster in (a). (c) An incorrect decision tree that would result if routing  $B$  and  $C_0$  had not been tested, leading to the failure to identify  $(b, B)$  and  $(c_0, C_0)$  as mutually exclusive.

#### D. Architecture Symmetries

Further pruning opportunities arise from inherent symmetries in cluster architectures. For example, in the cluster described in Figure 1a, the subset of the architecture consisting of  $B_0$  and  $D_0$  is identical to the one consisting of  $B_2$  and  $D_2$ : they are identical subgraphs connected to the I/Os. As a result, mapping application subgraphs to either subset results in equivalent mappings: in Figure 2a, (1) and (2) are equivalent. We annotate symmetry information on the architecture description and use it when building the decision tree to avoid exploring symmetric states, leading to further pruning.

#### E. A Greedy Covering Heuristic

We now describe the greedy heuristic used to select a cover of the application circuit using architecture clusters. To identify all potential mappings needed for covering, we apply the enumeration technique introduced in Section IV-A to each pair consisting of an application circuit node and a cluster primitive. The resulting set of enumerated mappings is denoted by  $M$ . To limit excessive overlap in the enumeration, the cluster designer can optionally specify a set of root nodes from which the clustering algorithm initiates decision tree explorations. Since these explorations are independent, we parallelize the implementation by running them concurrently.

We intentionally keep the selection policy simple, as our primary focus is on the quality of subgraph enumeration. To this end, we adopt a basic greedy heuristic for selecting covers, outlined in Algorithm 2: Mappings are ranked by size and selected in decreasing order until either the application circuit is fully covered or no valid mappings remain. In the latter case, the algorithm terminates with an error—though this situation cannot arise if the architecture includes universal clusters capable of implementing every primitive individually. When a mapping  $m$  is selected for inclusion in the cover, we update each remaining mapping  $m' \in M \setminus \{m\}$  by removing any overlapping nodes with  $m$  (i.e., we redefine each  $m' \leftarrow m' \setminus m$ ) and run each  $R(m')$  again. Mappings invalidated by this process remain in  $M$ , as further node removals may later restore their validity, but, naturally, we only include valid mappings from  $M$  into the final cover.

#### V. ROUTING SUBGRAPHS INSIDE CLUSTERS

We now turn to the problem of assessing whether an application subgraph  $G'$  can be implemented by an architecture cluster  $Cl_j$ . The relation  $Cl_j \sim G'$  denotes (i) that a particular mapping of  $G'$  represents a valid placement on cluster  $Cl_j$ , and (ii) that this placement is routable. This section describes the data structures and algorithms that we use to route application subgraphs in clusters.

---

##### Algorithm 2: Find Greedily a Cover

---

```

input :  $G$  and a set  $M$  of all candidate mappings
output: A full cover  $C$  of  $G$ , if it exists
return selectCover( $G, M, \emptyset$ )
function selectCover( $G, M', C_0$ ):
    if  $\forall q \in G, q \in C_0$  then
        | return  $C_0$ 
    else
        if isEmpty( $M'$ ) then
            | reportError(no cover)
        else
            |  $m \leftarrow$  findLargestMapping( $M'$ )
            |  $M'' \leftarrow$  removeAllOverlaps( $M', m$ )
            | return selectCover( $G, M'', C_0 \cup \{m\}$ )

```

---

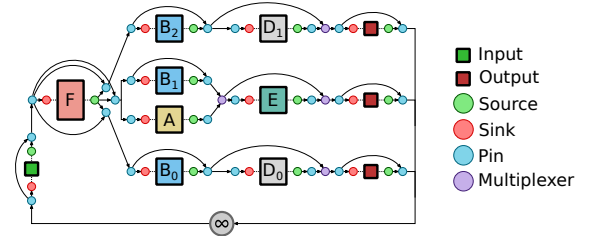


Fig. 5: The RRG corresponding to the cluster in Figure 1a. Bypass multiplexers are represented explicitly, similar to traditional FPGA RRGs. Identity primitives are represented with virtual edges from their input to output pins. Inputs (green squares) and outputs (red squares) are represented explicitly as identity primitives, allowing edges to exit and reenter the cluster. The routing resource with infinite capacity represents the global interconnect.

#### A. Routing Resource Graph Representation of Clusters

Similar to FPGAs [28], we construct a *routing resource graph* (RRG) to model clusters. In our RRG, nodes represent configuration multiplexers, input and output pins of architectural primitives, and special nodes termed *sources* and *sinks*, which connect to sets of equivalent output and input primitive pins, respectively. Edges in the RRG capture connectivity between routing resources, established via local interconnects (i.e., intra-cluster wires) or global interconnects (e.g., routing channels in an FPGA-style routing network). Figure 5 illustrates the RRG corresponding to the example cluster in Figure 1a.

We permit architecture clusters to include *identity* primitives, which can be configured to forward data directly from input to output ports. Supporting identity primitives is particularly important in CGRAs, where internal routing resources are limited and costly. Common examples include adders with a constant zero operand, forks in elastic circuits [2] when only a single output is used, or branches in elastic circuits when the input condition is constant. To model identity primitives, we insert a virtual edge from their input port to each output port. Furthermore, to facilitate connections between primitives across clusters using the general interconnect, we model all cluster ports as identity primitives. We connect the output pins (respectively, input pins) of all cluster outputs (respectively, inputs) to an RRG node with infinite capacity, representing highly flexible general interconnect resources.

#### B. An ILP Formulation of the Placement and Routing Problem

We formulate the placement and routing problem as an *integer linear program* (ILP), adapting prior work [29] to suit our objectives better. Notably, we employ an edge-centric approach instead of a node-centric one to facilitate combinational loop detection and support more flexible I/O placements.

In the following ILP formulation, we assume the existence of a function  $f_{\text{ext}}(G')$  that replaces neighboring nodes in  $G$  with I/O nodes, and use  $G'$  and  $f_{\text{ext}}(G')$  interchangeably.

##### 1) Definitions:

- **Value Set:**  $f_{\text{val}}(n)$  returns the set of values produced by node  $n \in G$ . The values produced by a node in the application circuit are simply the names of the nets it produces. The complete set for  $G'$  is  $f_{\text{val}}(G')$ .
- **Targets:**  $f_{\text{target}}(j)$  returns the set of targets for value  $j$ , representing multiple endpoints in a fanout.
- **Target Location:**  $f_{\text{tl}}(k)$  associates each target  $k$  with its corresponding node.

- **Target Function:**  $f_{\text{type}}(u)$  assigns a type (or simply, a name) to each resource  $u$ , such as functional units or I/Os. For example, in Figure 5,  $f_{\text{type}}(B_0) = B$ .
- **RRG:** For cluster  $Cl_j$ ,  $V_{\text{RRG}_i}$  and  $E_{\text{RRG}_i}$  denote its nodes and edges, respectively.
- **Capacity:**  $f_{\text{cap}}(r)$  associates routing resource node  $r \in V_{\text{RRG}_j}$  with its capacity.
- **Source and Sink Functions:**  $f_{\text{src}}(v)$  and  $f_{\text{sink}}(v)$  associate cluster primitive  $v \in V_{Cl_j}$  with its source and sink nodes in the RRG.

### 2) ILP Variables:

- $F_{p,q}$ : Binary variable indicating the cluster primitive  $p \in Cl_j$  is mapped to application node  $q \in G$ .
- $R_{r,j}$ : Binary variable indicating if routing resource  $r$  is used to route value  $j$ .
- $E_{r,r',j,k}$ : Binary variable indicating if RRG edge  $(r, r')$  is used to route value  $j$  to target  $k$ .

3) **ILP Constraints: Operation Placement:** ensure each application node  $q \in G'$  is mapped appropriately. If  $q$  is not an I/O, it should be mapped to exactly one cluster primitive. If  $q$  is an I/O, we allow it to be mapped to multiple cluster primitives, so that the same producer can send a value to multiple consumers within the cluster,

$$\sum_{p \in V_{Cl_j}} F_{p,q} \geq 1, \quad \forall q \in V_{G'} : f_{\text{type}}(q) = \text{I/O}, \quad (7)$$

$$\sum_{p \in V_{Cl_j}} F_{p,q} = 1, \quad \forall q \in V_{G'} : f_{\text{type}}(q) \neq \text{I/O}. \quad (8)$$

**Primitive Exclusivity:** prevent multiple mappings to the same primitive,

$$\sum_{q \in V_{G'}} F_{p,q} \leq 1, \quad \forall p \in V_{Cl_j}. \quad (9)$$

**Functional Unit Legality:** ensure type compatibility between application nodes and cluster primitives,

$$F_{p,q} = 0, \quad \forall q \in V_{G'}, \forall p \in V_{Cl_j} : f_{\text{type}}(p) \neq f_{\text{type}}(q). \quad (10)$$

**Source and Sink Legality:** enforce correct typing for routing edges. For example, a multiplexer has a condition and two data inputs. We assume primitive ports (and application values and targets) are well typed, meaning that we can unambiguously find the sources(s) and sink(s) a given net can use for routing,

$$E_{r,r',j,k} = 0, \quad \text{if } f_{\text{type}}(j) \neq f_{\text{type}}(r) \text{ or } \text{type}(k) \neq f_{\text{type}}(r'). \quad (11)$$

**Route Overuse:** limit usage of routing resources to their capacities,

$$\sum_{j \in f_{\text{val}}(G')} R_{r,j} \leq f_{\text{cap}}(r), \quad \forall r \in V_{\text{RRG}_j}. \quad (12)$$

**Routing Resource Usage:** link edge usage to routing resource utilization,

$$R_{r,j} \geq E_{r,r',j,k}, \quad \forall (r, r') \in E_{\text{RRG}_j}, \forall j \in f_{\text{val}}(G'), \forall k \in f_{\text{target}}(j), \quad (13)$$

$$R_{r',j} \geq E_{r,r',j,k}, \quad \forall (r, r') \in E_{\text{RRG}_j}, \forall j \in f_{\text{val}}(G'), \forall k \in f_{\text{target}}(j). \quad (14)$$

**Routing Continuity:** ensure continuous routing paths, i.e., if an edge  $(r, r')$  is used for routing value  $j$  to sink  $k$ , then at least one outgoing edge of  $r'$  is also used to route value  $j$  to sink  $k$ ,

$$\sum_{r'' \in \text{Succ}(r')} E_{r',r'',j,k} \geq E_{r,r',j,k}, \quad (15)$$

$$\forall r \in V_{\text{RRG}_j}, \forall j \in f_{\text{val}}(G'), \forall k \in f_{\text{target}}(j).$$

**Route Start:** initiate routing, i.e., if application node  $q$  is mapped to primitive  $p$ , then all values produced by  $q$  are mapped to the appropriate source of  $p$ ,

$$\sum_{r' \in \text{Succ}(r)} E_{r,r',j,k} \geq F_{p,q}, \quad (16)$$

$$q \in V_{G'}, \forall j \in f_{\text{val}}(q), \forall k \in f_{\text{target}}(j)$$

$$\forall p \in V_{Cl_j}, \forall r \in f_{\text{src}}(p) : f_{\text{type}}(r) = f_{\text{type}}(j).$$

**Route End:** terminate routing, i.e., if a route from value  $j$  to target  $k$  ends in sink  $r'$  of primitive  $p$ , and the target is associated with node  $q$ , then  $p$  is mapped to  $q$ ,

$$E_{r,r',j,k} \geq F_{p,q}, \quad (17)$$

$$\forall (r, r') \in E_{\text{RRG}_j} : r' \in f_{\text{sink}}(q)$$

$$\forall p \in V_{Cl_j}, \forall q \in V_{G'}, \forall j \in f_{\text{val}}(q), \forall k \in f_{\text{target}}(j).$$

**Prevent Loop:** avoid self-terminating combinational loops,

$$\sum_{r' \in \text{Pred}(r)} E_{r',r,j,k} \leq 1, \quad (18)$$

$$\forall r \in V_{\text{RRG}_j}, \forall j \in f_{\text{val}}(G'), \forall k \in f_{\text{target}}(j).$$

4) **Objective Function:** Minimize the total number of routing resources used,

$$\min \sum_{r,j} R_{r,j}. \quad (19)$$

### C. Handling Multiple I/O Mappings

We have defined constraints that direct routing from net sources to sinks. This scheme assumes each application node is mapped only once, aligning with prior ILP formulations [29]. However, an application node outside a cluster may need to deliver a value to multiple consumers within the same cluster, necessitating support for duplicate I/O mappings. Unfortunately, the ILP constraints described above would never map duplicated inputs. Equation 7 would be satisfied with a single input mapping, and routing constraints would not enforce the initiation of additional routes, leaving subsequent sinks unconnected without violating any constraints.

Therefore, we employ two sets of ILP constraints based on the nature of the routed edge. For edges between non-I/O cluster primitives or between a cluster primitive and an output, we apply the standard constraints. Conversely, for edges between an input and a cluster primitive, we reverse the routing direction—tracing from the sink back to the source. In this reversed approach, Equation 17 ensures that all reached sinks enforce the mapping of their corresponding I/Os. This bidirectional constraint system effectively manages cases involving multiple consumers of a single value within a cluster.

### D. Incremental ILP Computation

Invoking  $R(S)$  for any state  $S$  (see Section IV) entails generating the complete ILP described previously. However, the unitary state transitions incurred by the decision tree modify only a small subset of the constraints and variables—specifically, those affected by adding or removing a mapping. Profiling our implementation revealed that a significant portion of execution time was devoted to setting up and freeing ILP constraints. To address this, we implemented an incremental ILP approach that updates only the constraints and variables pertinent to the state transition. Given that constraints can overlap, we employ reference counting for all ILP-related constructs to determine when they can be safely deallocated, akin to mechanisms used in transactional memory systems.

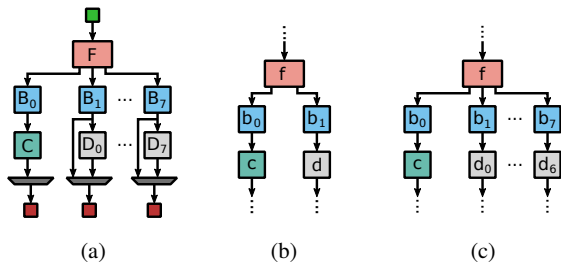


Fig. 6: (a) An example of a cluster of width eight (the  $F$  primitive can have up to eight outputs). (b)–(c) Examples of circuits of width two and eight, both mappable on the cluster in (a).

## VI. EVALUATION

We implemented FRESKO in Scala and ran all the experiments on a machine with Ubuntu 24.04 equipped with an AMD Ryzen 9 7950X processor and 125 GB of RAM. FRESKO and other artifacts for reproducing the results are available in an online repository [19].

To evaluate FRESKO’s performance and scalability, we use an openly available description of a highly complex architecture cluster, representative of modern heterogeneous fabrics at the intersection of FPGAs and CGRAs [15]. This cluster, denoted the *Control cluster*, has  $\sim 60$  inputs and a similar number of outputs. It contains 210 primitives of 29 different types, among which are 264 configuration multiplexers and  $\sim 200$  identity primitives. It also includes  $\sim 100$  buffers, all bypassable. We first evaluate the scalability of our approach on synthetic benchmarks, and then move on to using real circuits and comparing with CLUMAP [17].

### A. Scaling Within Clusters

The first experiment maps single-cluster application graphs onto a simplified version of the Control cluster [15] to evaluate the efficiency of the pruning strategies. Figure 6a shows part of the cluster, highlighting a typical feature: several identical branches extending from primitive  $F$ . Enumerating the possible mappings for the graph in Figure 6c is significantly more complex than for Figure 6b, due to the larger number of valid permutations.

Figure 7 plots the runtime when enumerating all possible mappings for graphs where the  $F$  primitive fans out to two, three, or more branches. The results are presented as an ablation study, selectively disabling specific optimizations to isolate their impact. For example,

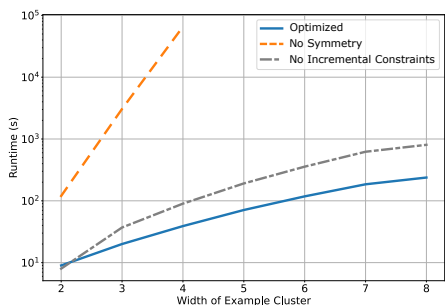


Fig. 7: Ablation Study for Mapping on a Single Cluster. The application circuits vary in width (i.e., number of outputs of primitive  $F$ ); see Figures 6b–6c. Each curve, except *Optimized*, corresponds to the variant of the algorithm with the specified optimization disabled.

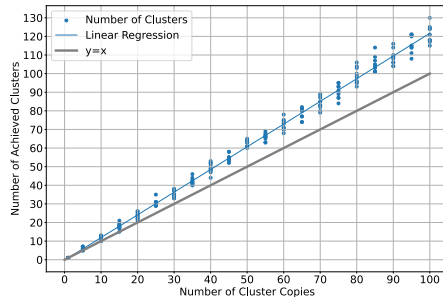


Fig. 8: Clustering Solution Quality vs. Circuit Complexity. FRESKO consistently stays within  $\sim 20\%$  of the quasi-optimal solutions.

disabling symmetry optimizations causes the runtime to become prohibitive with the number of branches. Exploiting symmetries significantly reduces runtime, resulting in subexponential growth on this practical case. Adding the incremental ILP routing optimization further improves performance: the relative speedup it provides remains nearly constant across problem sizes, increasing slightly with complexity—suggesting greater effectiveness for larger problems. As a result, the largest instance takes just over an order of magnitude more time than the smallest, rather than several.

### B. Scaling to Large Circuits

In the following experiments, we evaluate the performance of our clustering algorithm on increasingly larger circuit graphs. Since circuit graphs of arbitrary size are not readily available—and computing optimal clustering solutions is unfeasible—we carefully generate synthetic circuit graphs. To construct a circuit graph, we start from a given number of cluster instances. We then randomly suppress a fraction of the nodes by removing bypassable nodes near the cluster’s center or entire branches (such as those emanating from  $F$  in Figure 6a). Finally, we stitch together the remaining parts by randomly connecting a fraction of the remaining cluster outputs to the inputs; unconnected terminals are treated as primary inputs and outputs. In our experiments, we remove 60% of the primitives from the cluster replicas and connect 50% of the outputs to the inputs.

Although we cannot determine the exact optimal clustering for these circuits, it is reasonable to assume that the same number of clusters used initially to generate the circuit is a proxy for a close to optimal solution. We compare this baseline to the number of

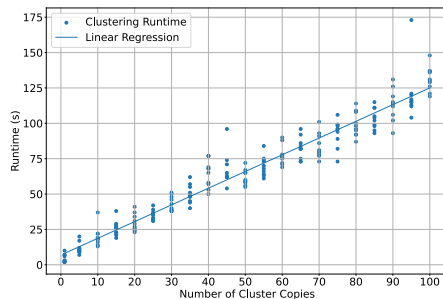


Fig. 9: Clustering Runtime vs. Circuit Complexity. Runtime grows only linearly with circuit size and remains well within practical limits, even for large circuits.

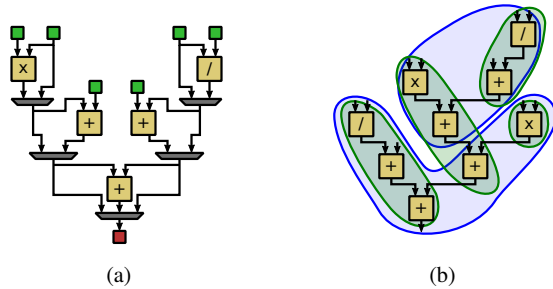


Fig. 10: **(a)** A simple architecture with a tree of adders, a multiplier, and a divider. All primitives are bypassable, so any connected subset can be used. **(b)** A synthetic application using two instances of the architecture in (a): one omits an adder between the multiplier and root; the other includes all primitives. Blue shows the optimal cover by the cluster in (a); green shows a valid but less efficient one.

clusters used in the solution computed by FRESCO. Figure 8 shows the results. For each circuit size, we generate ten random instances. Across the board, FRESCO consistently finds solutions that use only 20% more clusters than the baseline, on average. Importantly, solution quality remains stable even as the graph size increases, and for smaller circuits, the algorithm often recovers near-optimal clustering.

Figure 9 shows the corresponding runtimes. Crucially for practical use, runtime grows linearly with circuit size and remains manageable even for the largest circuits—those composed of 100 cluster copies—where it slightly exceeds a couple of minutes.

### C. Comparison Against State of the Art

Finally, we compare our clustering algorithm with CLUMAP, a recently published open-source CGRA mapper [17]. For this, we use a much simpler cluster architecture compatible with CLUMAP’s limitations (see Section III-B), shown in Figure 10a.

In the first experiment, shown in Figure 11, we use synthetic circuit graphs generated similarly to those in Section VI-B (see also Figure 10b). We generate ten different circuits for each circuit size (defined by the number of cluster copies). The results for FRESCO are not qualitatively different from those in Section VI-B, except that they are closer to the quasi-optimal solution, because the clusters are significantly simpler. CLUMAP follows the same general trend but with a consistent quality drop relative to the baseline. Two key

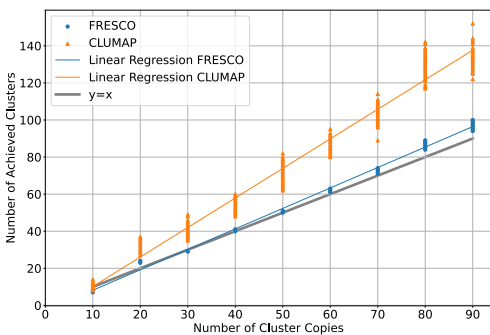


Fig. 11: FRESCO vs. CLUMAP on Synthetic Benchmarks. While both exhibit similar trends, FRESCO consistently achieves better and more stable solution quality.

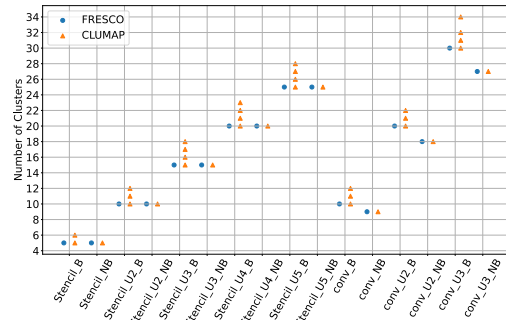


Fig. 12: FRESCO vs. CLUMAP on Riken Benchmarks. FRESCO consistently matches CLUMAP’s best solution, which CLUMAP only finds on some runs due to tie-breaking randomness.

differences stand out: (i) CLUMAP’s solution spread is noticeably wider and (ii) the quality of the solutions is roughly 50% worse than the baseline, substantially below FRESCO’s.

We also compare FRESCO and CLUMAP on CGRA evaluation benchmarks [17], [18], [30]. Figure 12 reports the cover quality from 75 random seeds per benchmark. We only plot distinct-quality solutions. In FRESCO, randomness arises from tie-breaking between equally sized mappings in the greedy heuristic. CLUMAP, by contrast, forms clusters deterministically, starting from the application circuit inputs. Therefore, we shuffle the application netlist to vary the cluster formation order and the resulting solutions. Both FRESCO and CLUMAP can achieve the optimal cover on these benchmarks (the optimal is known due to the manageable complexity of the clusters and applications). However, FRESCO does so deterministically, while CLUMAP only reaches it on some random seeds.

## VII. CONCLUSION

While FPGAs offer great flexibility, their fine-grained structure is often inefficient for general-purpose computing. Practical reconfigurable computing may require a coarser-grained architecture—one that better balances flexibility and efficiency. Although the ideal form of such architectures remains unclear, a viable candidate would likely follow the FPGA island-style blueprint: clustering primitives into tiles with local routing, as in FPGA clusters of look-up tables, flip-flops, and crossbars. However, routing among word-oriented primitives is far costlier, pushing toward larger clusters with many heterogeneous primitives and limited configurability. This fundamentally shifts the mapping problem from packing look-up tables into FPGA clusters to mapping heterogeneous application primitives onto complex clusters with tight internal routing.

In this paper, we have shown that such exploration is feasible for the largest cluster sizes in the literature. Our techniques prune the search space effectively by leveraging cluster topology to discard invalid or redundant solutions early, yielding orders-of-magnitude speedups over brute-force search. In practical cases, our approach exhibit subexponential complexity with respect to cluster size and, crucially, it scales linearly with the application graph size. While still open to further improvement, we believe that our techniques can play a key role in exploring and enabling more efficient CGRAs with complex cluster structures.

## VIII. ACKNOWLEDGEMENTS

This project was partially supported by the Swiss National Science Foundation (grant No. 219299).

## REFERENCES

- [1] A. Boutros and V. Betz, "FPGA architecture: Principles and progression," *IEEE Circuits and Systems Magazine*, vol. 21, no. 2, pp. 4–29, 2021.
- [2] L. Josipović, R. Ghosal, and P. lenne, "Dynamically scheduled high-level synthesis," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18. Association for Computing Machinery, 2018, pp. 127–136.
- [3] J. Cong and Z. Zhang, "An efficient and versatile scheduling algorithm based on SDC formulation," in *Proceedings of the 43rd ACM/IEEE Design Automation Conference*, ser. DAC '06. Association for Computing Machinery, 2006, pp. 433–438.
- [4] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoon, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: High-level synthesis for FPGA-based processor/accelerator systems," in *Proceedings of the 2011 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11. Association for Computing Machinery, 2011, pp. 33–36.
- [5] H. Ye, C. Hao, J. Cheng, H. Jeong, J. Huang, S. Neuendorffer, and D. Chen, "Scalehls: A new scalable high-level synthesis framework on multi-level intermediate representation," in *In Proceedings of the 2022 IEEE International Symposium on High-Performance Computer Architecture*, ser. HPCA'22. IEEE, 2022, pp. 741–755.
- [6] Y. Umuroglu, Y. Akhauri, N. J. Fraser, and M. Blott, "Logicnets: Co-designed neural networks and circuits for extreme-throughput applications," in *Proceedings of 2020 30th International Conference on Field-Programmable Logic and Applications*, ser. FPL'20. IEEE, 2020, pp. 291–297.
- [7] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 203–215, 2007.
- [8] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel patterns," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA'17. Association for Computing Machinery, 2017, pp. 389–402.
- [9] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix," in *Proceedings of the 2003 International Conference on Field-Programmable Logic and Applications*, ser. FPL'03. Springer, 2003, pp. 61–70.
- [10] S. A. Chin, N. Sakamoto, A. Rui, J. Zhao, J. H. Kim, Y. Hara-Azumi, and J. Anderson, "CGRA-ME: A unified framework for CGRA modelling and exploration," in *Proceedings of 2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors*, ser. ASAP'17. IEEE, 2017, pp. 184–189.
- [11] B. Mei, M. Bereković, and J.-Y. Mignolet, "ADRES & DRESC: Architecture and compiler for coarse-grain reconfigurable processors," in *Fine- and Coarse-Grain Reconfigurable Computing*. Springer, 2007, pp. 255–297.
- [12] M. Karunaratne, A. K. Mohite, T. Mitra, and L.-S. Peh, "HyCUBE: A CGRA with reconfigurable single-cycle multi-hop interconnect," in *Proceedings of the 54th ACM/IEEE Annual Design Automation Conference*, ser. DAC'17. Association for Computing Machinery, 2017, pp. 1–6.
- [13] O. Ragheb, S. Wicklund, M. Walker, R. Beidas, A. Ragab, T. Yu, and J. Anderson, "CGRA-ME 2.0: A research framework for next-generation CGRA architectures and CAD," in *Proceedings of the 2024 IEEE International Parallel and Distributed Processing Symposium Workshops*, ser. IPDPSW'24. IEEE, 2024, pp. 642–649.
- [14] O. Ragheb, T. Yu, D. Ma, and J. Anderson, "Modeling and exploration of elastic CGRAs," in *Proceedings of the 2022 International Conference on Field-Programmable Logic and Applications*, ser. FPL'22. IEEE, 2022, pp. 404–410.
- [15] L. Coulon, L. Ramirez, J. Anderson, M. Stojilović, and P. lenne, "FRIDA: Reconfigurable arrays for dynamically scheduled high-level synthesis," in *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '25. Association for Computing Machinery, 2025, pp. 147–158.
- [16] G. Gobieski, S. Ghosh, M. Heule, T. Mowry, T. Nowatzki, N. Beckmann, and B. Lucia, "RipTide: A programmable, energy-minimal dataflow compiler and architecture," in *Proceedings of 2022 55th IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO'22. IEEE, 2022, pp. 546–564.
- [17] O. Ragheb and J. H. Anderson, "CLUMAP: Clustered mapper for CGRAs with predication," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, ser. DAC '24. Association for Computing Machinery, 2024, pp. 1–6.
- [18] R. Beidas and J. H. Anderson, "CGRA mapping using zero-suppressed binary decision diagrams," in *Proceedings of 2022 27th Asia and South Pacific Design Automation Conference*, ser. ASP-DAC'22. IEEE, 2022, pp. 616–622.
- [19] EPFL-LAP. (2024) FRIDA Compiler. Accessed: 2025-23-07. [Online]. Available: <https://github.com/EPFL-LAP/FRIDA>
- [20] J. Luu, "Architecture-aware packing and CAD infrastructure for field-programmable gate arrays," Ph.D. Thesis, University of Toronto, 2014.
- [21] T. Kojima, N. A. V. Doan, and H. Amano, "GenMap: A genetic algorithmic approach for optimizing spatial mapping of coarse-grained reconfigurable architectures," *2020 IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 11, pp. 2383–2396, 2020.
- [22] K. J. M. Martin, "Twenty years of automated methods for mapping applications on CGRA," in *Proceedings of the 2022 IEEE International Parallel and Distributed Processing Symposium Workshops*, ser. IPDPSW'22, 2022, pp. 679–686.
- [23] G. Zhou, M. Stojilović, and J. H. Anderson, "GRAMM: Fast CGRA application mapping based on a heuristic for finding graph minors," in *Proceedings of the 2023 International Conference on Field-Programmable Logic and Applications*, ser. FPL'23, 2023, pp. 305–310.
- [24] L. McMurchie and C. Ebeling, "PathFinder: A negotiation-based performance-driven router for FPGAs," in *Proceedings of the 1995 ACM International Symposium on Field-Programmable Gate Arrays*, ser. FPGA'95, 1995, pp. 111–117.
- [25] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub)graph isomorphism algorithm for matching large graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 10, pp. 1367–1372, 2004.
- [26] K. Keutzer, "DAGON: Technology binding and local optimization by dag matching," in *Proceedings of the 24th ACM/IEEE Design Automation Conference*, ser. DAC'87. Association for Computing Machinery, IEEE, 1987, pp. 341–347.
- [27] J. Cong and Y. Ding, "FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 1, pp. 1–12, 1994.
- [28] V. Betz and J. Rose, "Automatic generation of FPGA routing architectures from high-level descriptions," in *Proceedings of the 2000 ACM/SIGDA Eighth International Symposium on Field Programmable Gate Arrays*, ser. FPGA '00. Association for Computing Machinery, 2000, pp. 175–184.
- [29] S. A. Chin and J. H. Anderson, "An architecture-agnostic integer linear programming approach to CGRA mapping," in *Proceedings of the 55th ACM/IEEE Design Automation Conference*, ser. DAC '18. Association for Computing Machinery, 2018, pp. 1–6.
- [30] B. Adhi, C. Cortes, Y. Tan, T. Kojima, A. Podobas, and K. Sano, "The cost of flexibility: Embedded versus discrete routers in CGRAs for HPC," in *Proceedings of the IEEE International Conference on Cluster Computing*, ser. CLUSTER'22. IEEE, 2022, pp. 347–356.