Leveraging Prior Knowledge for Effective Design-Space Exploration in High-Level Synthesis

Lorenzo Ferretti[1], Jihye Kwon[2], Giovanni Ansaloni[1], Giuseppe Di Guglielmo[2], Luca P. Carloni[2], Laura Pozzi[1]
[1]*Università della Svizzera italiana, Lugano, Switzerland,* [2]*Columbia University, New York, United States*

***Abstract*—High-Level Synthesis (HLS) tools allow the generation of a large variety of hardware implementations from the same specification by setting different optimization directives. Each combination of HLS directives returns an implementation of the target application that is based on a particular microarchitecture. Designers are interested only in the subset of implementations that correspond to Pareto-optimal points in the 'performance vs. cost' design space. Finding this subset is hard because the relationship between the HLS directives and the Pareto-optimal implementations cannot be foreseen. Hence, designers must default to an exploration of the design space through many time-consuming HLS runs. We present a methodology that infers knowledge from past design explorations to identify high-quality directives for new target applications. To this end, we formulate a novel abstract representation of applications and their associated configuration spaces, introduce a similarity metric to compare quantitatively the configuration spaces of different applications, and a method to infer actionable information from a source space to a target space. Experimental results with the MachSuite benchmarks show that our approach retrieves close approximations of the Pareto frontier of best-performing implementations for the target application, in exchange for a small number of HLS runs.**

***Index Terms*—High-Level Synthesis, Knowledge Transfer, Design-Space Exploration, Hardware Acceleration.**

I. Introduction

High-level Synthesis (HLS) enables the automatic generation of hardware designs from high-level specifications given, for example, as C/C++ or SystemC code [1]. With HLS, designers can first specify complex functionalities in a simpler way by working at a higher level of abstraction than register-transfer level (RTL), and then synthesize many different implementations from the same specification by applying optimization directives before running the HLS tool. Examples of directives include the unrolling factor of a loop, the inlining of functions, and the mapping of arrays to memory structures. The combined application of directives has a major impact on the microarchitecture of the synthesized implementation. Hence, HLS directives enable a broad exploration of the design space in search of implementations that are Pareto-optimal with respect to conflicting objectives such as performance (latency, throughput) and cost (area, power) [2], [3]. For complex designs, however, the relationships between the combination of many directives and the quality of the synthesized implementations are difficult to foresee before the execution of time-consuming HLS runs. Moreover, the number of alternative implementations increases exponentially with the number of considered directives, thus making exhaustive explorations infeasible in practice even for simple cases.

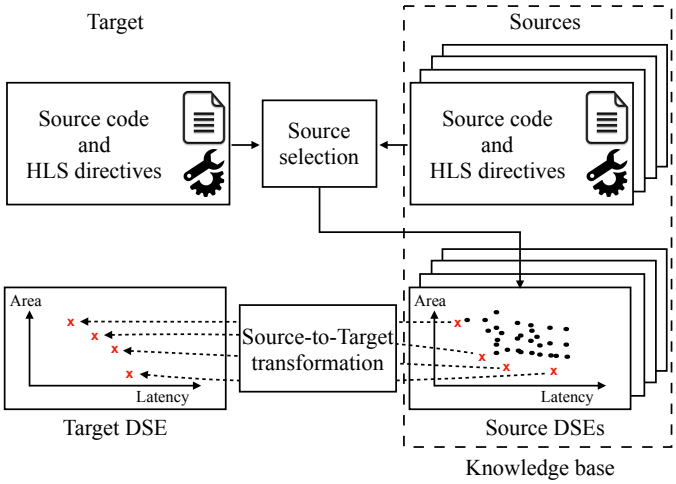

Fig. 1: High-level description of the methodology proposed in this work. A target design is compared to available sources in a knowledge base. The most similar source design is identified and leveraged to learn effective optimizations for the target, approximating the Pareto frontier of its design space while requiring few HLS runs.

Several research efforts (summarized in Section VI) have proposed strategies to discover, in the context of HLS-based designs, the most effective implementations from a cost and performance perspective, while minimizing the number of HLS runs. This problem is named *HLS-driven design-space exploration (DSE)*. Our contribution tackles this problem from a novel perspective: for the first time, we explore the feasibility of effectively *harnessing the knowledge of past synthesis outcomes to guide the optimization of new designs*.

Figure 1 illustrates our approach. When exploring a new *target* design with HLS, we first select a *source* design from a library of previously completed explorations. The source design is the one whose specification is most similar to the target one. Then, we consider the combinations of directives that led to the best results for the source design. These are Pareto-optimal implementations (marked with red crosses in Figure 1), i.e. those for which no other implementations resulted in both less area and lower latency. The main idea is that the similarity among the two designs justify the translation of the directives from the source design into directives for the target design, in order to lead the search for an approximation

Manuscript received April 18, 2020; revised June 12, 2020; accepted July 6, 2020. This article was presented in the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems 2020 and appears as part of the ESWEEK-TCAD special issue.

of the set of Pareto-optimal target implementations.

In more detail, our strategy employs a novel *similarity metric* to identify the most appropriate source design for a given target design. Then, it adopts a novel *mapping strategy* to link the directives between the related, but not identical, source and target designs. Ultimately our methodology derives a set of Pareto-optimal implementations for a new HLS design from a prior knowledge at a minimal cost in terms of synthesis runs.

Our contribution is therefore three-fold:

- We explore the feasibility of harnessing prior knowledge in the context of HLS-driven DSEs.
- We introduce an abstract representation of HLS design-space, a metric to assess the similarity between sources and targets, and a way to infer optimizations across different DSEs.
- We demonstrate the effectiveness of our approach across an extensive set of 39 different designs from MachSuite [4]. We retrieve close approximations of the Pareto set of the best implementations, achieving a Pareto frontier distance (Average Distance from Reference Set, i.e. ADRS) of 0.009 in the median case. On average, we only require the exploration of less than 8% of target design spaces, and only an average of 38 synthesis runs per design.

II. Motivation

When optimizing a design with HLS, an expert designer starts by identifying which directives are applicable. For example, given the code in Snippet 1 of Figure 2, the designer may be interested in exploring unrolling factors for loops, combined with different degrees of partitioning for the input/output arrays. Furthermore, the designer may recall to have already optimized in the past a design with a similar code structure, such as the one reported in Snippet 2 (also shown in Figure 2). Indeed, even if they are not identical (e.g., the loop boundaries and the memory access patterns differ), the two code snippets have some structural similarities: they both iterate over two nested loops and process data provided in input to the function through pointers. These similarities may be sufficient to suggest adapting those directives that lead to optimal implementations for Snippet 2 to the case of Snippet 1, instead of starting the DSE by trying anew many combinations of directives for Snippet 1.

The designer's empirical strategy to tackle the DSE task hence consists of three main steps: a) identify the main structural characteristics in the code of the target design, b) pinpoint a similar already-explored design, and finally c) transfer the knowledge from the source design to the new target design.

Our methodology performs these steps, but, differently from the above-described scenario, operates in a systematic and automated way. In Section IV, we show that our explorations, guided by prior knowledge, yield close approximations of the Pareto-optimal results from an exhaustive approach, while requiring very few synthesis runs. Our methodology answers the following three research questions.

R.Q.1: From an HLS perspective, how can similarities among designs be quantified?

In general, code written in a high-level programming language such as C/C++ or SystemC is ill-suited for the automatic identification of structural similarities. Therefore, we propose an abstract representation *that only retains the characteristics of interest for HLS optimizations*, e.g., the structure of loops and that of memory access patterns. We extract such a representation (that we termed *specification encoding*) with a custom compiler pass. Since the representation is in the form of a string of symbols, we can use *string-similarity algorithms* to quantify the similarity in terms of computational patterns that exist between a source design (from a library capturing prior knowledge) and the target design.

R.Q.2: How can the similarity between directive choices for different designs be assessed?

Besides the specification code, the other aspect affecting the HLS results is the choice of HLS directives. Indeed, a proper source of prior knowledge should have a choice of directives values similar to the one of the target. As an example, if a loop can be unrolled by only a small degree in a source, little information can be leveraged to optimize a loop in the target for very high unrolling factors. We introduce a domain-specific language to describe succinctly the set of directives associated with a design and a metric to measure the similarity between sets of directives associated with the source and target designs. Then, in the *source selection strategy* step, which is shown in the top part of Figure 1, we combine design and directive similarities to identify the most promising source for the given target design.

R.Q.3: How to infer from prior knowledge HLS directives that give optimal results?

We have designed a strategy that transforms the HLS directives for the source design into HLS directives of the target design, as shown in the lower part of Figure 1.

In the next section, we describe in detail the answers to these three research questions.

III. Methodology

A. Terminology

An *HLS design* (or *design*) is a functionality to be realized in hardware. A *specification* is a high-level description of the design in a programming language such as SystemC or C/C++, given in input to the HLS tool. An *implementation* of the design is the output of a run of the HLS tool. This output is typically expressed as an automatically generated RTL code written in Verilog or VHDL. Each implementation is characterized by the values of a performance metric and a cost metric.

A *synthesis configuration* (or, simply, *configuration*) defines the transformations that a design undergoes through HLS. A designer controls these transformations with constraint and optimization *directives*, such as loop unrolling or pipelining, array manipulation, and other control and datapath optimizations. A directive is associated with a *location* in the code specification. A location could be either a label in the code or a language construct; for example, a loop or an array declaration. A designer can further customize some directives by specifying the *values* for the directive parameters; for

Snippet 1: `last_step_scan` (target).

```
1  void last_step_scan(int bucket[SIZE], int sum[RADIX]){
2    int i, j, k;
3    loop_1:for(i = 0; i < RADIX;i++){
4      loop_2:for(j = 0; j < BLOCK; j++) {
5        k = (i * BLOCK) + j;
6        bucket[k] = bucket[k] + sum[i];
7      }
8    }
9  }
```

Snippet 2: `get_delta_matrix_weights2` (source).

```
1  void get_delta_matrix_weights2(double delta_weights2[
       N_NODES*N_NODES], double output_difference[
       N_NODES], double last_activations[N_NODES]) {
2    int i, j;
3    loop_1:for(i = 0; i < N_NODES; i++) {
4      loop_2:for(j = 0; j < N_NODES; j++) {
5        delta_weights2[i * N_NODES + j] = last_activations[i]
             * output_difference[j];
6      }
7    }
8  }
```

Fig. 2: Running example. Code snippets of two different functions from MachSuite [4]. Snippet 1 shows the C code of `last_step_scan` used in this paper as example of target design. Snippet 2 shows the C code of `get_delta_matrix_weights2` used as example of source design. The code has been rewritten to increase readability, without modifying the functionality.

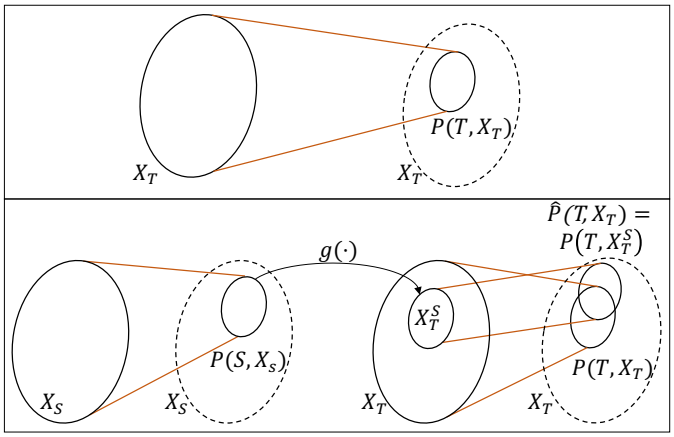

Fig. 3: (Top) Standard approach: the designer defines a set of configurations to be explored, X_T, given a target design T. Only after synthesizing all the X_T configurations, Pareto optimal ones $P(T, X_T)$ are identified.
(Bottom) Approach leveraging prior knowledge instead: the configurations to be synthesized X_T^S are *inferred* from the $P(S, X_S)$ of a similar design S. By synthesizing T with $X_T^S \ll X_T$ configurations, a close approximation $\widehat{P}(T, X_T)$ of the Pareto frontier is obtained.

example, the designer can customize the amount of parallelism in the implementation by unrolling a loop a certain number of times or by setting a certain initiation interval for the pipeline implementing the loop.

A *design space* is the set of all the possible design configurations and the associated costs and performance results from HLS. A *Pareto configuration* of a design is a configuration that leads to an implementation that is Pareto-optimal in the bi-objective optimization space defined by the performance and cost metrics. A (first-rank) Pareto frontier is the set of Pareto-optimal points. Finally, an $i-th$ *rank Pareto frontier* (for $i > 1$) is defined as the Pareto frontier obtained after removing the lower rank frontiers from the design space.

B. Problem description

For a design T, let $\mathcal{X}_T$ denote the set of all possible synthesis configurations. In general, $\mathcal{X}_T$ is a very large set, possibly of infinite size. In practice, designers explore a portion of the design space of T by trying a subset $X_T \subset \mathcal{X}_T$, whose elements they choose carefully based on their experience running HLS. Given T and X_T, the *design space exploration task* returns a subset of X_T that consists of all Pareto configurations, i.e.

$$P(T, X_T) = \{x | x \in X_T \text{ and } x \text{ is Pareto}\} \tag{1}$$

The subset $P(T, X_T)$ is obtained by first (1) performing $|X_T|$ HLS runs on T, one run for each $x \in X_T$, and then (2) by selecting only those configurations that turn out to be Pareto configurations.

Now, assume that before performing the DSE task for T (the *target* design), the designer has performed the DSE task for another design S (the *source* design), thereby obtaining $P(S, X_S)$ for a given subset X_S of the configuration set $\mathcal{X}_S$. Furthermore, assume that a function $g : X_S \rightarrow X_T$ exists that transforms a configuration for the source design into one for the target design, i.e.

$$g(x_s) = x_t \tag{2}$$

with $x_s \in X_S$ and $x_t \in X_T$. With the help of function g, the designer can leverage the *prior knowledge* on the source design in order to perform a DSE for the target design with a potentially much smaller number of HLS runs.

Let X_T^S be the set of all configurations for the target design T that are obtained by transforming the Pareto configurations (up to a certain Pareto frontier rank) of the source design, i.e.:

$$X_T^S = \{g(x_s) | x_s \in P(S, X_S)\} \tag{3}$$

By synthesizing the target design T with the configurations in X_T^S, we can obtain the set $P(T, X_T^S)$ as an approximation $\widehat{P}(T, X_T)$ of the set of Pareto configurations $P(T, X_T)$.

Figure 3 showcases the difference between a standard approach and one leveraging prior knowledge.

Note that this approximation requires $|X_T^S|$ HLS runs, while the derivation of the actual set of Pareto configurations would require $|X_T|$ HLS runs. Tuning the maximum Pareto frontier rank whose configurations are transformed from source to target, the synthesis effort and the approximation of $P(T, X_T^S)$ by $\widehat{P}(T, X_T)$ can be traded-off. We explore the effect of varying this parameter in Section IV. Since for a given design T the number of Pareto configurations $|P(T, X_T)|$ is, in general, much smaller than the number of configurations $|X_T|$, if sets $P(T, X_T)$ and $P(S, X_S)$ are of comparable sizes then leveraging prior knowledge allows a major reduction in the number of time-consuming HLS runs while deriving the approximated set $\widehat{P}(T, X_T)$.

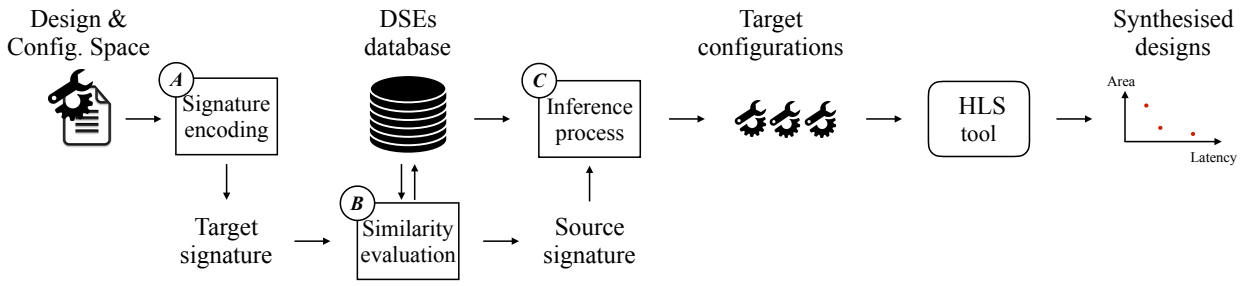

Fig. 4: Methodology flow. The target design and its configurations space are encoded into a signature, which is compared to the ones of existing DSEs. The source having the most similar signature is selected to drive the inference process and generate the target configurations.

Moreover, the degree to which $P(T, X_T)$ is approximated by $\widehat{P}(T, X_T)$ depends on the choice of a proper source design S to derive the prior knowledge for the given target T. To this end, we introduce a novel and concise representation to encode the specification and a configuration space of each design via an abstract characterization called *signature*. Then, we define a similarity metric between the signatures. If the signatures of two designs have a high similarity, then Pareto configurations for one design – when transformed to configurations for the other – may approximate well the actual Pareto configurations for the other design. Moreover, signatures are also employed to automate the transformation of Pareto configurations between source and target spaces, thereby realizing function $g(\cdot)$ of Equation 2.

Figure 4 illustrates the overall flow of our methodology. Given as input a target design's specification and configuration space, our strategy (A) derives the signature of the design, and (B) employs a similarity metric over such signature to search, in a database of already performed DSEs (the sources), for the most similar one. Once a source is selected, the Pareto configurations *for that source* are extracted, and (C) they are transformed by an inference process into valid configurations for the target.

Each of these three steps – signature encoding, similarity evaluation, and inference – are detailed in the remainder of this section.

C. Signature Encoding

This step aims to characterize a DSE with a compact representation that abstracts the specification (code) and the associated configurations (set of applied directives). The proposed *specification encoding* (SE) and *configuration space descriptor* (CSD) capture these two aspects. The combination of SE and CSD uniquely defines a *signature encoding*.

Specification Encoding. A specification encoding describes those aspects of an HLS specification that can be targeted by HLS directives, such as the presence of loops and read/write operations, while disregarding anything that is not interesting from a HLS-driven DSE perspective. The encoding process generates a string representation of the specification that highlights the source code structure.

Table I shows the encoding scheme adopted and the correspondence between the string symbols and the code constructs. We derive the SE from the C/C++ specification through an LLVM [5] pass. We extended the compiler to parse the abstract syntax tree and produce the SE string. The last column of Table I also shows the HLS directives that can be associated with each code construct. We use the curly braces to identify the scope associated with symbols, thus allowing hierarchical representations (e.g., a function containing multiple nested loops).

TABLE I: Specification encoding of design source code.

Symbols	Code constructs	HLS directives
F	Function definition	None
V	Function parameter passed by value	None
P	Function parameter passed by reference	Partitioning and resource
A	Arrays definition or declaration	Partitioning and resource
S	Struct definition or declaration	Partitioning and resource
L	Loops	Unrolling
R	Read operations	None
W	Write operations	None
C_{id}	Function call	Inlining
{ .. }	Scope	None

Running Example: Given the function `last_step_scan` from Snippet 1 of Figure 2 and the encoding in Table I, the proposed SE is `F{PP}L{L{RRW}}`. The encoding states that the function (`F`) receives two parameters by reference (`PP`), it has two nested loops and the innermost loop performs two reads and one write operations (`L{L{RRW}}`). Likewise, the SE for Snippet 2 from Figure 2 is `F{PPP}L{L{RRW}}`, showcasing a similar, but not identical, structure. ■

Configuration Space Descriptor. We defined a domain-specific language to concisely describe a user-defined configuration space. For source designs, CSDs describe which configurations are available in its design space, while for a target a CSD indicates the set of configurations that a designer wishes to explore. Each line of the descriptor encodes a *knob* (a type of directives, the location, and selected parameter values) that the designer considers for the DSE task. For a directive with multiple parameters, a set of values for each parameter is specified.

Running Example: Given the function `last_step_scan` in Snippet 1 of Figure 2, the associated CSD is shown in Snippet 3 of Figure 6. The descriptor defines seven different knobs that can be associated with the function `last_step_scan`. Line 1 of Snippet 3 shows a knob with a single value: it associates a dual-ported Block RAM (BRAM) to the array `bucket` that is the input of the function.

Line 3 instead defines a knob governing the array parti-

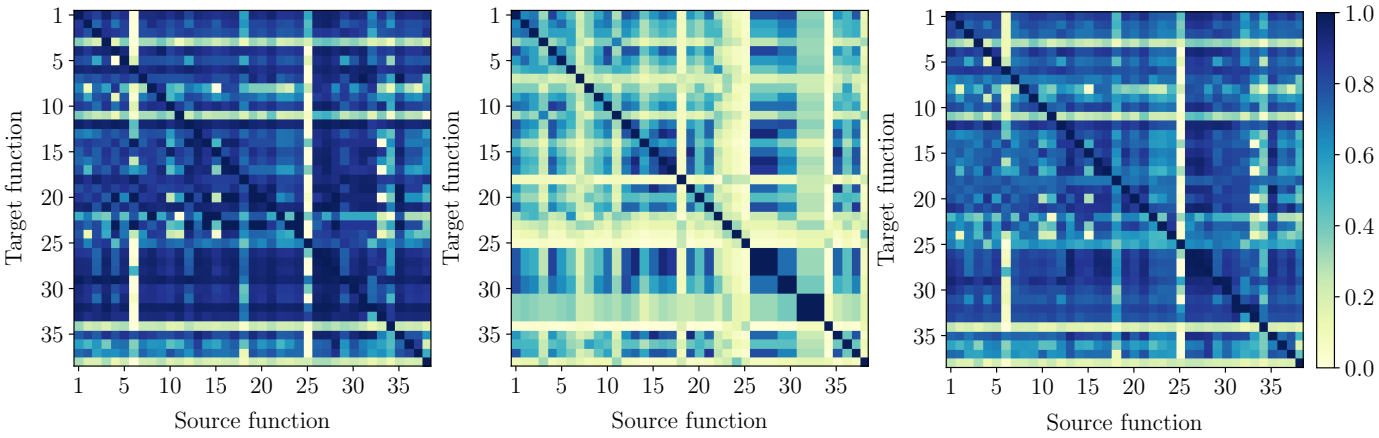

Fig. 5: (Left) Signature similarity matrix obtained with $\alpha = 0.2$. (Center) Specification Encoding similarity matrix. (Right) Configuration Space Descriptor similarity matrix. Darker color expresses a higher similarity. Each row of the matrix shows the similarity between a target design and the source ones. The indices on the axes corresponds to the function IDs in Table II.

Snippet 3: Configuration Space Descriptor of `last_step_scan`.

```
1 resource;last_step_scan;bucket;{RAM_2P_BRAM}
2 resource;last_step_scan;sum;{RAM_2P_BRAM}
3 array_partition;last_step_scan;bucket;1;{cyclic,block
      };{1->512,pow_2}
4 array_partition;last_step_scan;sum;1;{cyclic,block};{1->128,
      pow_2}
5 unroll;last_step_scan;last_1;{1->128,pow_2}
6 unroll;last_step_scan;last_2;{1,2,4,8,16}
7 clock;{10}
```

Snippet 4: Configuration Space Descriptor of `get_delta_matrix_weights2`.

```
1 array_partition;get_delta_matrix_weights2;delta_weights2;1;{
      cyclic,block};{1->256,pow_2}
2 array_partition;get_delta_matrix_weights2;output_difference
      ;1;{cyclic,block};{1->64,pow_2}
3 array_partition;get_delta_matrix_weights2;last_activations;1;{
      cyclic,block};{1->64,pow_2}
4 unroll;get_delta_matrix_weights2;loop_1;{1->64,pow_2}
5 unroll;get_delta_matrix_weights2;loop_2;{1->64,pow_2}
6 clock;{10}
```

Fig. 6: Running example. Code snippets of the Configuration Space Descriptors for the functions shown in Figure 2. Snippet 3 shows the CSD defined for `last_step_scan` used in this paper as an example of target CSD. Snippet 4 shows the CSD defined for `get_delta_matrix_weights2` used as an example of source CSD.

tioning directive defined by all the pairs having one of two partitioning strategies (`cyclic` and `block`) as first component, and the ten possible partitioning factors (all the powers of two from 1 up to 512) as the second one. ■

The CSD is parsed and the resulting set of configurations of the design space is generated as follows:

$$CS = K_1 \times K_2 \times \cdots \times K_N \tag{4}$$

where N is the number of considered knobs, and K_i is the set of values related to each i knob, i.e. the set of values that the directive associated to the knob i can assume. For a directive with multiple parameters, K_i is the Cartesian product among each set of values. The size of the configuration space is then given by its cardinality ($|CS|$).

D. Similarity evaluation

To choose a candidate source design for a target design, we compute a similarity metric between their signature encodings. We compute such a similarity given the similarities of the design SEs and CSDs:

$$Sim = \alpha Sim_{SE} + (1-\alpha) Sim_{CSD} \quad \alpha \in [0,1] \tag{5}$$

The parameter α in Equation 5 weights the contributions of the SE and CSD similarities. The best source candidate, selected according to the similarity metric, is used to transfer knowledge from source to target design during the inference step.

Figure 5-left shows the similarity matrix for the 39 functions in the MachSuite Benchmarks. Each row shows the similarity between a target design and all the candidate source designs; the diagonal elements show the similarity of the design to itself. The figure highlights a high similarity variance that discriminates well between similar and dissimilar sources. In Section IV, we show that our chosen similarity metric leads to an effective selection of the source for the given target.

SE similarity. Since we express the specification encoding as a string, we can use string-based algorithms to assess the similarity between SEs. In our approach, we adopt the *longest common subsequence (LCS)* metric [6]. This metric returns a score $Sim_{SE} \in [0,1]$, whose value is closer to 1 the more two strings are alike.

Figure 5-center expresses the Sim_{SE} matrix for the same 39 functions, where each row shows the similarity between a target design and all the candidate sources.

Running Example: Given the specification encoding of the target design `F{PP}L{L{RRW}}` and the source design `F{PPP}L{L{RRW}}` in Figure 2, the resulting SE similarity score is 0.93. ■

CSD Similarity. The similarity between two CSDs is assessed by comparing the knobs K_i for a target configuration space X_T (for design T) to the knobs K_j for a source configuration space X_S (for design S) using a mapping function

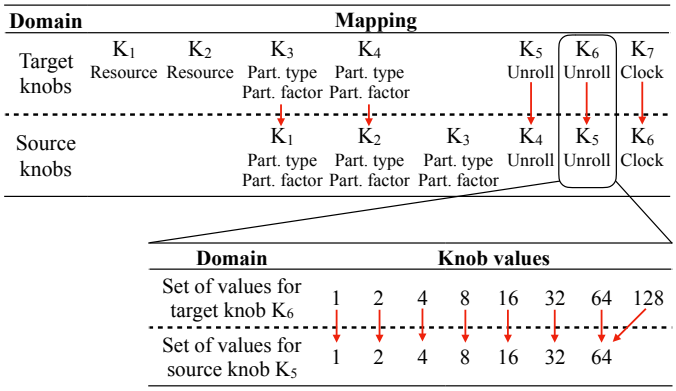

Fig. 7: Top: mapping between the knobs of source and target CSD shown as example (Snippet 1 and 2). Bottom: correspondence between the knob value sets of knobs K_6 and K_5 in target and source, leading to a distance Δ of 1.

$M_{T,S}$, which relates each knob of the target CSD to a specific knob of the source CSD:

$$M_{T,S}(K_i) = K_j \tag{6}$$

The function $M_{T,S}$ is determined through an alignment procedure. By iterating over the knobs of the target CSD from top to bottom, each knob is mapped to the first non-mapped knob of the same type belonging to the source CSD. Eventually, if no more knobs are available in the source, some target knobs may be left unmapped.

Running Example: Let us consider the target and source designs in Figure 2 and their CSDs in Snippet 3 and Snippet 4 of Figure 6, respectively. The CSD of function `last_step_scan` has seven different knobs – each knob is one line of the descriptor – while the CSD of function `get_delta_matrix_weights2` has only six knobs. Figure 7 shows the mapping $M_{T,S}$ between the two CSDs. Five out of seven knobs of function `last_step_scan` can be mapped by using `get_delta_matrix_weights2` as a source design; while knobs K_1 and K_2 of `last_step_scan` are unmapped. ■

Once a mapping $M_{T,S}$ is defined between a target configuration space X_T, defined with I different knobs, and a source configuration space X_S, defined with J different knobs, we compute their similarity as follows:

$$Sim_{CSD} = 1 - \left[\frac{1}{I}\sum_{i=1}^{I}\Delta(K_i, M_{T,S}(K_i))/D_{MAX}\right] \tag{7}$$

where D_{MAX} is a normalization factor – constant across all the source candidates for a given target – such that $Sim_{CSD} \in [0,1]$. Then, $\Delta(\cdot)$ is a function measuring the minimum distance between a source knob and a target knob:

$$\Delta(K_i, K_j) = \sqrt{\sum_{n=1}^{|K_i|}(\min_{m=1}^{|K_j|}|\delta(k_n, k_m)|)^2} \quad k_n \in K_i, k_m \in K_j \tag{8}$$

The above equation sums up the distance between each target knob value k_n and the one that is closest to it among all source knob values k_m. The function $\delta(k_n, k_m)$ computes the distance between two knob values of the same directive type that has Z parameters, e.g., $k_n = (k_{n,1}, \cdots, k_{n,Z})$:

$$\delta(k_n, k_m) = \sqrt{\sum_{z=1}^{Z}|k_{n,z}, k_{m,z}|^2} \tag{9}$$

where numerical parameter values are casted to their respective log_2 value, and categorical parameter values are represented with one-hot encoding.

Since for unmapped knobs there is no correspondence between source and target, the distance $\Delta(\cdot)$ in Equation 8 is computed between the values of the target knob and the default value of the directive.

Figure 5-right shows the resulting Sim_{CSD} matrix for the functions in the MachSuite Benchmark Suite considered in this work.

Running Example: Given the mapping between the functions in our running example (Figure 2), for each target knob we measure the distance with respect to the source one. Figure 7 (bottom) shows the computation of the distance for the target knob K_6 mapped to the source node K_5, each having a single value set of possible unrolling factors. K_6 specifies 8 factors (from 1 to 128, all of them powers-of-two), while K_5 comprises 7 values (from 1 to 64). Since the δ is calculated among numerical values, the directive values are casted to their respective $log2$; therefore, the knobs discrepancy leads to a Δ equal to $(log_2 128 - log_2 64) = 1$. When accounting for all target knobs, the CSD similarity between the source and target is 0.97. ■

E. Inference

After a source design is identified for a target design, the inference process transfers knowledge from the source to the target configuration space, hence implementing Equation 2. In the first step of such a process, we extract the configurations belonging to the Pareto frontier in the source configuration space from a library of prior knowledge. These are peeled from the source design space, allowing the identification of second-rank Pareto configurations. Then, we proceed iteratively to extract higher ranked Pareto frontiers, until a certain number of these have been retrieved from the source design space.

Each selected configuration is transformed into a valid one in the target CSD. To this end, first, knobs in the source and target spaces are mapped according to the mapping function $M_{T,S}$ described in the previous section. If a target knob K_i is not be mapped to a source knob, the value of x_T^i value is fixed to the directive default, since no prior knowledge related to that knob can be leveraged. The values of all other knobs are instead inferred from the source design space.

Then, given a source configuration $x_S = [x_S^1, \ldots, x_S^J] \in X_S$, with x_S^j being the value for knob j, the corresponding target configuration is set as $x_T = [x_T^1, \ldots, x_T^I] \in X_T$, where each component x_T^i are knob values associated to the knob i.

For each configuration component, we define the inference function ($g : X_S \rightarrow X_T$, introduced in Equation 2) as follows:

$$x_T^i = \arg\min_n\{\delta(k_n, x_S^j)\} \tag{10}$$

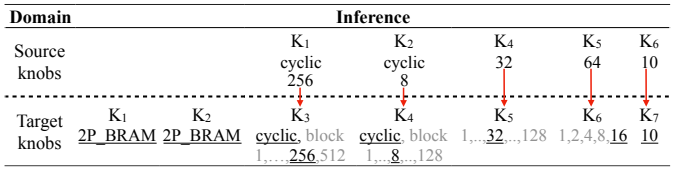

Fig. 8: Inference from source to target design spaces from the running example. The inferred values of the HLS directive knobs are underlined in the bottom part of the figure.

where $\delta(\cdot)$ is the distance function defined in Equation 9, x_S^j is the values assigned to the $j-th$ knob of the source, and $k_n \in K_i$ is the set of all possible sets of values that the knob K_i of the target design can assume, as specified by its configuration space descriptor. Therefore, each target directive value x_T^i is assigned to the closest value to x_S^j among those specified in the target knob set for knob i.

Running Example: Let us assume that, among the many Pareto configurations of the source design in Figure 2, one configuration has the directive values shown in Figure 8. Given the mapping function from Figure 7 and Equation 10, we transform the source Pareto configuration into a valid target configuration. We map the partitioning factors – 256 and 8 for K_1 and K_2 of the source – to the closest partitioning factor values of the target knobs – respectively 256 and 8 for K_1 and K_2 of the target. Similarly, we infer the same partitioning type – cyclic – from the source Pareto configuration for the target ones. Finally, we map the source unrolling factors and the clock, 32, 64 and 10 for K_4, K_5 and K_6 to 32, 16 and 10 for K_5, K_6 and K_7 in the target, respectively. ■.

IV. Experiments

A. Experimental setup

We implemented the similarity evaluation and inference algorithms in Python. We implemented the SE encoding in C++ as a custom compiler pass within the LLVM infrastructure, as described in Section III-C.

Our experiments targeted all of the functions in the MachSuite benchmarks collection [4], except those that expose very small design spaces, and those having a variable latency for different invocations during the benchmark execution due to input-dependent control flows. In total, 11 designs were discarded. The resulting suite comprises 39 functions, which have on average 40 lines of code and 308 in the biggest case.

For each design, we performed an extensive DSE across their configuration spaces up to tens of thousands of design points. We used Vivado HLS [7] to run synthesis with a target clock period of $10ns$ and targeting a ZynqMP Ultrascale+ (`xczu9eg`) FPGA chip. We collected the design configurations and synthesis results in a MySQL database.

In order to control the configuration space size[1], we only employed power-of-two values for directives having a numerical range (e.g., loop-unrolling and array-partitioning factors), and, in some cases, we forced related knobs to have the same value (e.g., the partitioning factor of an array and the unrolling of a loop accessing it once every iteration). Such a decision corresponds to the intuitive choice of constraints that a designer would impose when tasked with the exploration of the design space. More than 4 years of single-core machine time are required to generate the knowledge base. To speed up this process, we collected synthesis data from up to 60 instances of Vivado HLS running in parallel. This allowed us to reduce to produce the database in approximately 25 days.

We use these extensive DSEs in two ways. On the one hand, we use these results as ground truth to assess the performance of our approach. On the other, we use them as a source of prior knowledge. In the latter case, we adopted a leave-one-out cross-validation, considering each design as a target using all others as candidate knowledge sources.

Similarly to [8]–[11], we used as quality metric the *Average Distance from Reference Set (ADRS)*, which expresses the distance between a reference curve P (the Pareto frontier from ground truth data), and an approximated curve $\bar{P}$. The ADRS for two objective functions is defined as:

$$ADRS(\bar{P}, P) = \left[\frac{1}{|P|}\sum_{p\in P}\min_{\bar{p}\in\bar{P}}(d(\bar{p}, p))\right], \quad (11)$$

$$d(\bar{p}, p) = \max\{0, (A_{\bar{p}} - A_p)/A_p, (L_{\bar{p}} - L_p)/L_p\} \quad (12)$$

Low ADRS values are better, because they imply proximity between P and $\bar{P}$. In our scenario, the first objective function is the FPGA resource requirement (area, A) of an implementation, expressed as the average utilization of Flip-Flops, Look-Up Tables, DSP, and Block RAMs. The second objective function is run-time performance, i.e., latency (L) in nanoseconds.

B. Results

Outcome of Explorations. Table II summarizes the results of the explorations performed with our methodology. It reports the target function IDs (used as indexes in Figure 5), their benchmarks and the function names. Moreover, for each case, it provides the function IDs and the function names of the source having the highest similarity score, the obtained ADRS values in the target space, the number of synthesized configurations derived from that source, and the size of the related configuration space ($|CS|$).

For the vast majority of the targets, our approach requires very few syntheses to reach low ADRS scores. As an example, when targeting `aes_addRoundKey` (row index: 20) while leveraging the knowledge of the `add_bias_to_activations` source, only 13 out of 500 possible synthesis rounds are performed, still resulting in a perfect identification of the Pareto frontier of best performing implementations (ADRS = 0). Only 35 out of thousands of configurations are synthesized for the `gemm` target (row index: 5) while reaching a very close Pareto frontier approximation (ADRS = 0.012).

We obtained the results of Table II by inferring up to the 10th-ranked Pareto frontier (as defined in Section III-A) and by fixing the trade-off between SE and CSD similarity (introduced as α in Section III-D) to 0.2. We further explore both settings in the rest of this section.

[1]Even for the simple case in Snippet 2, considering all loop unrolling factors, two types of resources, two types of partitioning and all partitioning factors would result in more than 10^8 configurations.

TABLE II: List of functions explored from MachSuite [4] (grouped by benchmark). The table reports: target function ID, it benchmark name, target function name, source function IDs, source function names, ADRS value, number of synthesized configurations, and size of the configuration space ($|CS|$).

ID	Benchmark	Target function	Source ID	Source function	ADRS	# Synth.	|CS|
1	spmv ellpack	ellpack	28	get_delta_matrix_weights2	0.034	65	1600
2	bfs bulk	bulk	28	get_delta_matrix_weights2	0.010	39	2352
3	md knn	md_kernel	30	get_oracle_activations1	0.006	25	1600
4	viterbi	viterbi	28	get_delta_matrix_weights2	$8.7e^{-4}$	12	1152
5	gemm ncubed	gemm	28	get_delta_matrix_weights2	0.012	35	2744
6	gemm blocked	bbgemm	28	get_delta_matrix_weights2	1.689	35	1600
7	fft strided	fft	30	get_oracle_activations1	0.0007	15	1600
8	fft transpose	twiddles8	32	product_with_bias_input_layer	0.0002	24	64
9	sort merge	ms_mergesort	27	get_delta_matrix_weights1	0.322	31	1024
10		merge	30	get_oracle_activations1	0.262	19	4096
11	stencil stencil2d	stencil	28	get_delta_matrix_weights2	0.015	46	1344
12	stencil stencil3d	stencil3d	28	get_delta_matrix_weights2	1.88	16	1536
13	radix sort	update	30	get_oracle_activations1	0.009	28	2400
14		hist	28	get_delta_matrix_weights2	0.007	46	4704
15		init	18	local_scan	0.078	68	484
16		sum_scan	36	add_bias_to_activations	0.136	25	1280
17		last_step_scan	28	get_delta_matrix_weights2	0.004	90	800
18		local_scan	17	last_step_scan	0.005	71	704
19		ss_sort	32	product_with_bias_input_layer	0.0005	21	1792
20	aes	aes_addRoundKey	36	add_bias_to_activations	0	13	500
21		aes_subBytes	29	get_delta_matrix_weights3	0	8	50
22		aes_addRoundKey_cpy	28	get_delta_matrix_weights2	0	71	625
23		aes_shiftRows	16	sum_scan	0.013	8	20
24		aes_mixColumns	25	aes_expandEncKey	0	15	18
25		aes_expandEncKey	13	update	0.003	33	216
26		aes256_encrypt_ecb	4	viterbi	0.030	22	1944
27	backprop	get_delta_matrix_weights1	28	get_delta_matrix_weights2	0.002	139	21952
28		get_delta_matrix_weights2	27	get_delta_matrix_weights1	0.010	77	31213
29		get_delta_matrix_weights3	28	get_delta_matrix_weights2	0.030	222	21952
30		get_oracle_activations1	31	get_oracle_activations2	2.907	67	2401
31		get_oracle_activations2	29	get_delta_matrix_weights3	0.051	19	1372
32		product_with_bias_input_layer	34	product_with_bias_output_layer	3.560	3	1372
33		product_with_bias_second_layer	32	product_with_bias_input_layer	0	30	686
34		product_with_bias_output_layer	32	product_with_bias_input_layer	$2.5e^{-5}$	24	392
35		backprop	1	ellpack	$4.4e^{-5}$	4	2048
36		add_bias_to_activations	20	aes_addRoundKey	0.002	5	1372
37		soft_max	29	get_delta_matrix_weights3	0.053	9	64
38		take_difference	29	get_delta_matrix_weights3	0.0002	8	512
39		update_weights	4	viterbi	$1.1e^{-4}$	3	1024

Tuning of the similarity function. Figure 9 shows the ADRS scores achieved when selecting the most similar candidates according to the similarity metric in Equation 5 while varying the parameter α, i.e., the relative weight of specification encoding and configuration space similarity. Data is shown on a logarithmic scale and in an aggregated form across all targets. Boxes encompass the first and third quartile of the ADRS values obtained by the DSEs of all targets, while the lines inside them indicate the median case. The skewers above and below each box are the upper and lower 1.5 interquartile. As before, we inferred configurations up until the 10th-ranked Pareto frontier in the source design space.

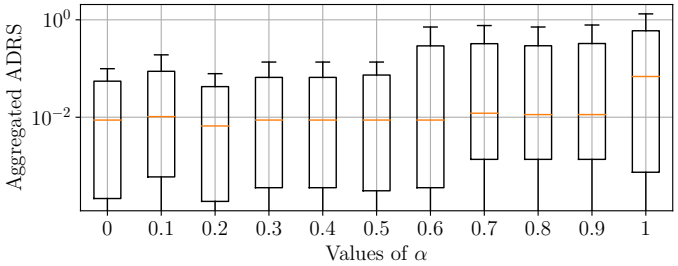

Fig. 9: ADRS evolution while changing the value of α.

Figure 9 that both SE and CSD have an impact on the quality of results and that CSD similarity generally has a more significant impact than the SE one. An α value of 0.2 both minimizes the interquartile range and the median ADRS.

Effectiveness of the similarity metric. Figure 10 highlights the importance of a proper source of prior knowledge in order to achieve effective explorations. It depicts four DSEs of the target design `last_step_scan`, from the running example, leveraging different sources of prior knowledge. Each plot depicts the ground truth of the target design space resulting from its exhaustive exploration – gray dots – as well as the Pareto frontier retrieved with the inference process – dark blue line. The top-left DSE shows the result of inferring configuration from `get_delta_matrix_weights2` (ID 27), the best-ranked source according to our similarity metric. In this case, the Pareto frontier is very well approximated and obtains a small ADRS of 0.004.

The top-right picture shows an example of DSE characterized by a low SE similarity, resulting in a partial approx-

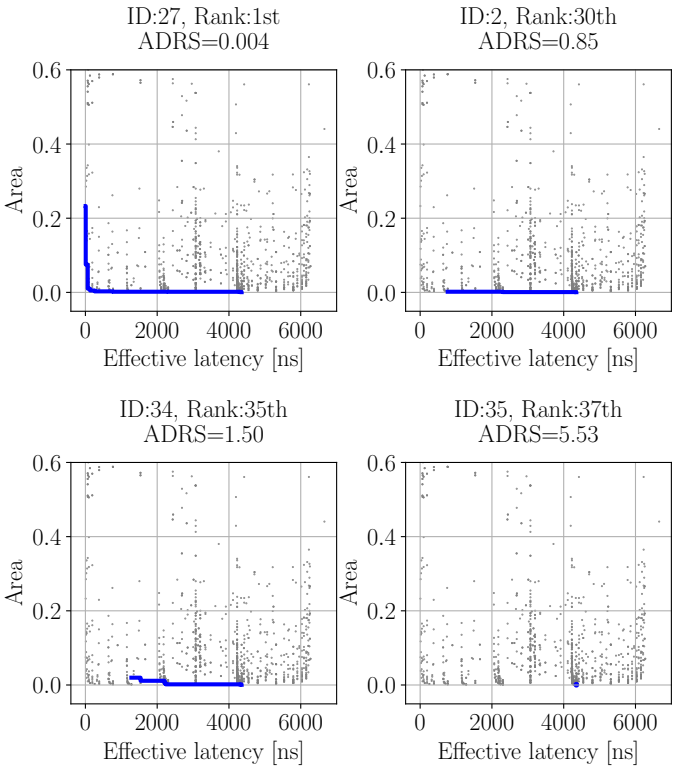

Fig. 10: Example of DSEs targeting `last_step_scan`, inferring from different sources. (Top-left) Good Pareto-approximation obtained with the best candidate source. (Top-right, bottom-left, bottom-right) Low quality Pareto approximations, from sources having low CSD and/or AS similarity. Gray dots represent the ground-truth for the target design `last_step_scan` while the dark blue line represents the Pareto frontier obtained performing the inference with different sources.

imation of the Pareto frontier, since only a few knobs can be mapped from source to target. In this case, the inference process uses as source design the function `bulk` (ID 2), which is ranked 30th in order of similarity score. Similarly, the bottom-left picture shows the result of the DSE when `product_with_bias_output_layer` (ID 34) is employed as a source design. In this case, the similarity score is penalized by a low CSD similarity. Therefore, only a portion of the target design space can be explored, due to inadequate coverage of the knob values of X_T by the ones in X_S. The Pareto frontier is hence well approximated only for the X_T region for which prior knowledge is available, resulting in an ADRS of 1.5. Finally, in the plot at the bottom-right of Figure 10, we show the result of the inference from `backprop` (ID 35). In this case, we observed both a low SE similarity (few knobs can be mapped from source to target) and a low CSD similarity (for mapped knobs, knob values are distant between source and target spaces) resulting in an extremely poor approximation of the Pareto frontier, since little prior knowledge can be harnessed. In this case, as reported in the figure, the retrieved Pareto frontier only comprises a single design point.

Figure 11 generalizes these findings by reporting the aggregated ADRS values when selecting the sources with the highest similarity score for each target, the second-best choices, etc. As in Figure 9, we plot the data on a logarithmic scale.

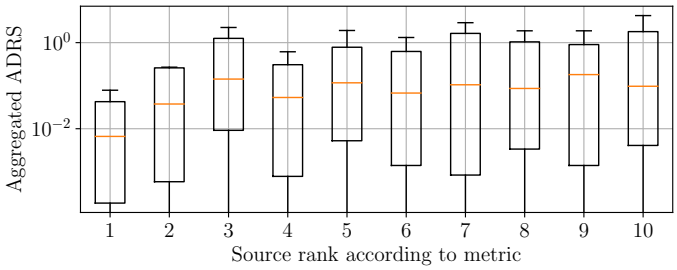

Fig. 11: ADRS according to source selection by metric ranking.

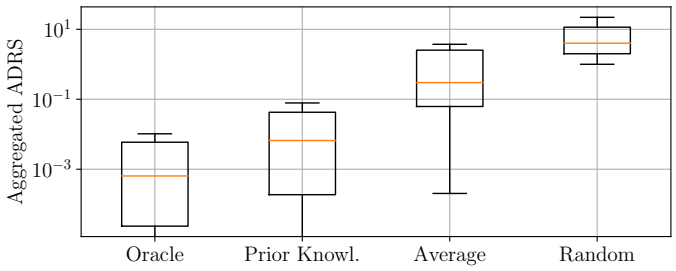

Fig. 12: ADRS according to source selection criterion.

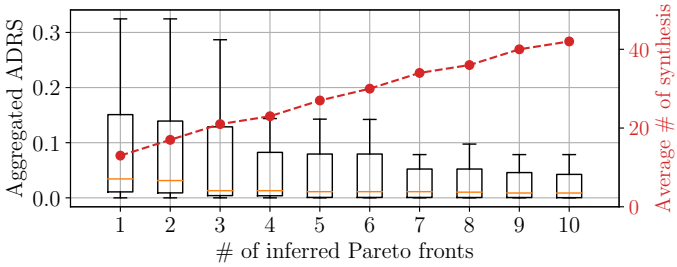

Fig. 13: Cumulative evolution of the ADRS while inferring from multiple Pareto frontiers. The average number of synthesis performed while the number of Pareto frontiers increases is shown with the dashed line.

Boxes encompass the first and third quartile of the ADRS scores across all targets, while the line inside the boxes indicates the median case. An order of magnitude separates the best and third-best choice, after which performance remains almost constant and tangibly worse than in the case of the best-ranked source.

Finally, Figure 12 compares the aggregated ADRS values obtained by our strategy (named Prior Knowl. in the figure), i.e., leveraging the source with the highest similarity, with three alternatives: a perfect oracle always choosing a-posteriori the best source (Oracle), the average of selecting all sources for each target (Average), and a random sampling of the target design space (Random) – disregarding knowledge transfer altogether. For the latter case, we performed several samplings equal to the ones required by our methodology, and we averaged the results over 100 different runs to minimize noise. A choice that is driven by the proposed similarity metric (Prior Knowl.) performs three orders of magnitude better than a choice that is not considering past explorations at all (Random) and 12X better when compared with a blind choice for the source design (Average).

Tuning the number of source Pareto frontiers. In a further round of experiments (Figure 13), we investigated the effect of varying the number of selected source Pareto frontiers. Each

TABLE III: Qualitative comparison with SoA methodologies. Average # of synthesis required to obtain an ADRS ≤ 0.04.

$\|\|CS\|\|$	**Prior Knowl.**	**Lattice** [10]	**Cluster** [9]	**RF-TED** [8]	**Zhong** [11]
< 200	7	36	37	155	NA
< 700	10	64	64	391	19
< 1800	22	230	290	1588	31
< 6000	19	460	460	1903	32
< 16000	NA	NA	NA	NA	35
< 32000	38	NA	NA	NA	NA

boxplot shows the aggregated ADRS outcomes when inferring an increasing number of Pareto frontiers from the highest-similarity source to the target. While increasing the number of frontiers always lowers ADRS scores, diminishing returns can be observed for a number of frontiers ≥ 7. The number of required synthesis, instead, linearly increases with the amount of inferred Pareto frontiers.

Comparison with State of the Art. Table III compares our methodology (named Prior Knowl. in the table) with four related works that also aim to automate the optimization of HLS designs. According to the taxonomy of Section VI, three of them are refinement-based approaches [8]–[10], while one is a model-based approach [11]. For fairness, we group the results in different brackets according to the configuration size of the employed benchmarks. When no benchmark is reported for a given size on past works, we marked the corresponding table cell with NA. In other cases, data shows the average number of synthesis runs required to reach an ADRS of 0.04, which Zhong et al. consider as an excellent Pareto frontier approximation [11] (and which is attained by our approach in 29 out of 39 cases (see Table II)).

The numbers in this table show that our approach greatly outperforms the refinement-based approaches (see the required number of synthesis in the first four columns), and this advantage grows with the size of the configuration space. Our strategy is *even* competitive with the model-based strategy of Zhong et al. [11] (see the last column), while being agnostic to the number and type of optimizations that can be considered. [2]

Leveraging prior knowledge across different clock constraints and platforms. All previous results assumed that the same clock constraint ($10ns$) and FPGA model (Xilinx Zynq xczu0eg) are employed for all synthesis runs. In practice, both these conditions may not be satisfied, as often past explorations may be performed for an FPGA than is different from the one of interest for a new design. Similarly, the clock constraints may, in general, not be the same for sources and target. Nonetheless, our methodology is robust toward variations of FPGA models and operating frequencies because it relies on the Pareto-dominance relationship in the cost/performance space of implementations in each DSE composing the knowledge base, as opposed to relying on the actual values of area and latency. This relationship, and consequently the set of Pareto configurations of a design, is not tangibly affected by the employed clock period and FPGA.

To investigate this characteristic, we have observed the ADRS obtained by identifying the implementations of the first-rank Pareto frontier of the `last_step_scan` benchmark

[2]Zhong et al. only considers the loop unrolling and the dataflow directives.

TABLE IV: ADRS obtained by leveraging the knowledge of the `get_delta_matrix_weights2` source while exploring the `last_step_scan` target, varying the clock constraints and FPGA models employed for the target benchmark.

Clock period (ns)	**Technology**	**ADRS**
10	ZynqMPU+	0.0044
5	ZynqMPU+	0.0044
25	ZynqMPU+	0.0044
50	ZynqMPU+	0.0044
10	Artix	0.0049
10	Virtex	0.0045
10	Kintex	0.0045

(13 out of 1600 implementations) synthesized with various clock periods, and inferring the related configurations for a different clock constraint. We have evaluated the result of the inference for target and sources with clock constraints of 5ns, 10ns, 25ns and 50ns. For all the experiments no changes in the inferred Pareto frontier have been observed, and hence good approximations of the Pareto frontier have been obtained. These results confirm that indeed the set of Pareto configurations are not tightly dependent on the operating frequency.

Similar remarks are obtained when varying the FPGA employed for the synthesis of source and target. We have observed the ADRS obtained by identifying the implementations of the first-rank Pareto frontier of the `last_step_scan` benchmark synthesized with various FPGAs (ZynqMPUltrascale+ `xczu9eg`, Virtex `xc7vh580`, Kintex `xc7k352`, and Artix `xc7a100`), and inferring the related configurations for a different platform. For all the combinations of target and sources platforms, the Pareto configurations are the same. Even in this case, the inferred Pareto frontier perfectly approximates the one obtained by an exhaustive exploration.

Concluding this round of experiments, Table IV shows an example of applying our methodology, again considering `last_step_scan` as a target, while employing the knowledge base in Table II. The design `get_delta_matrix_weights2` (ID 28) is identified as the most similar source and it is used to infer configurations up to its 10th-rank Pareto frontier – the same setting adopted for the results in Table II. In the first row of the table, provided for reference, both the clock constraint and the FPGA of the target design are equal to the ones used for the source. In rows 2-4, three different target clock constraints are used, while in rows 5-7 the target FPGA is different from the source one. In all cases, very similar, and small, ADRS are achieved, showcasing the robustness of our methodology.

V. Extensions & Future Works

Across a large variety of designs, leveraging prior knowledge can lead to the identification of high-quality implementations while requiring a low budget of synthesis.

However, the quality of the implementations discovered during a DSE relies on the presence of a similar DSE in the knowledge base. While we have shown that this scenario is not unrealistic – 75% (29 out-of 39) of the benchmarks have obtained an ADRS lower than 0.04 – it is still possible that none of the available sources has a high degree of similarity

for a given target. This drawback could be alleviated by partitioning target applications, and inferring prior knowledge for each of them separately, possibly from different sources. Moreover, a better coverage of target design spaces could be achieved by interpolating and extrapolating from the data in the source ones. In addition, more complex mapping algorithms could be conceived for matching source and target knobs (see Section III-D), e.g. taking into account loop nesting levels, the presence of loop carried dependency, control flow, etc.. Implementing such strategies would be a natural extension of this work.

Our framework is, for the most part, independent from the metrics used to measure cost and performance (for the experiments in Section IV we considered area and latency, respectively) as these are only employed to identify Pareto configurations in the source space. Further trade-offs can therefore be explored, for example including energy efficiency as an alternative or additional dimension.

Finally, experimental evidence confirms the intuition that high similarity between sources and targets results in better approximation of the Pareto frontier of a given design space. We hope such findings will spur follow-up efforts investigating the link between pre-synthesis evaluations (e.g. similarity) and post-synthesis validation (e.g. ADRS).

VI. Related work

Recent works proposing strategies to navigate HLS design-spaces can be organized in two main categories. On one hand, model-based approaches cite [11]–[14] rely on an estimation of performance and resource requirement of a given optimization. While mandating very few synthesis runs, such strategies struggle when coping with multiple, interdependent optimizations. Hence, they are often limited to capturing the effect of only few directives.

On the other hand, refinement-based frameworks rely on the outcome of some synthesis runs as a starting point, and aim to improve on this initial solution using different strategies such as random forest [8], genetic algorithms [15], simulated annealing [16], clustering [9] or local search techniques [10]. Refinement-based methods are agnostic to the set of considered directives, but usually exhibit a slower convergence rate, because they must incrementally build a knowledge of the design space being explored.

Our proposed methodology neither relies on an a-priori model nor tries to infer it from an ongoing exploration. By focusing on *previous* explorations, we can instead tap on a knowledge base which is both rich in terms of available data points and robust towards different directives and their combination. Indeed, as summarized in the surveys by Pan et al. [17] and Weiss et al. [18], useful information can be extracted from previous sets of experiments for which the outcomes are known, in order to efficiently explore new ones. Nonetheless, most of the strategies described in these surveys deal with domains that are distant from HLS, such as classification and object recognition applications.

In a recent editor's note, Doppa et al. [19] highlighted the importance of leveraging prior knowledge to effectively reduce the complexity of DSE problems. A small number of works take this stance in the context of hardware design. However, they do so from a different and somehow limited perspective: Dai et al. aim at improving the accuracy of HLS estimations using post-synthesis data [20], while the goal of Liu et al. is to estimate the performance on FPGA from an ASIC synthesis report [21]. Deshwa et al. leverage prior knowledge in the context of network-on-chip DSEs, to identify promising starting point for their exploration methodology [22]. More recently, Wang et al. [23] proposed a method to accelerate the process of HLS-driven DSE by pre-characterizing micro-kernels offline and creating predictive models of these. Finally, Martins et al. [24] also present a strategy to harness prior knowledge based on a similarity metric, but their framework is geared towards the selection of *compiler* optimizations, as opposed to targeting the hardware domain of HLS.

VII. Conclusions

In this work we have proposed a methodology for leveraging prior knowledge in HLS-driven design space exploration (DSE). We consider the Pareto-dominance relationship among directive configurations in a source design, and translate Pareto configurations from a source into corresponding ones in a target. By transferring knowledge from similar DSEs, we are able to retrieve high quality implementations with very sparse sampling of target DSEs, requiring few synthesis runs.

The proposed strategy assesses similarities between sources and targets – therefore identifying the most promising source to learn from – and performs inference of configurations based on a novel abstract representation. Such representation provides a succinct view of the design code and of the configuration space of a DSE.

Our methodology greatly outperforms state-of-the-art refinement-base strategies, requiring much fewer synthesis to derive the same DSE quality. Results are in line with model-based methods, but, as opposed to them, we are not restricted in the type of supported directives.

Acknowledgment

This work has been partially supported by the National Science Foundation (A#: 1527821), the MagicISEs (grant no. 200021-156397) and the ML-Edge (grant no. 200020-182009) projects evaluated by the Swiss NSF and by the MyPreHealth (grant no. 16073) project founded by the Hasler Stiftung.

References

[1] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *IEEE Design & Test of Computers*, vol. 26, no. 4, pp. 18–25, Aug. 2009.

[2] H.-Y. Liu, M. Petracca, and L. P. Carloni, "Compositional system-level design exploration with planning of high-level synthesis," in *Proceedings of the Conference on Design, Automation and Test in Europe*, Mar. 2012, pp. 641–646.

[3] L. Piccolboni, P. Mantovani, G. Di Guglielmo, and L. P. Carloni, "COSMOS: Coordination of high-level synthesis and memory optimization for hardware accelerators," *ACM Transactions on Embedded Computing Systems*, vol. 16, no. 5s, pp. 150:1–150:22, Sep. 2017.

[4] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks, "MachSuite: Benchmarks for accelerator design and customized architectures," in *Proceedings of the IEEE International Symposium on Workload Characterization*, Oct. 2014, pp. 110–119.

[5] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization*, Mar. 2004, p. 75.
[6] M. Paterson and V. Dančík, "Longest common subsequences," in *International Symposium on Mathematical Foundations of Computer Science*, Jun. 1994, pp. 127–142.
[7] "Vivado High-Level Synthesis." [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html
[8] H.-Y. Liu and L. P. Carloni, "On learning-based methods for design-space exploration with high-level synthesis," in *Proceedings of the 50th Design Automation Conference*, Jun. 2013, pp. 1–6.
[9] L. Ferretti, G. Ansaloni, and L. Pozzi, "Cluster-based heuristic for high level synthesis design space exploration," *IEEE Transactions on Emerging Topics in Computing*, no. 99, pp. 1–9, Jan. 2018.
[10] ——, "Lattice-traversing design space exploration for high level synthesis," in *Proceedings of the International Conference on Computer Design*, Oct. 2018, pp. 210–217.
[11] G. Zhong, V. Venkataramani, Y. Liang, T. Mitra, and S. Niar, "Design space exploration of multiple loops on FPGAs using high level synthesis," in *Proceedings of the International Conference on Computer Design*, Dec. 2014, pp. 456–463.
[12] N. K. Pham, A. K. Singh, A. Kumar, and M. M. A. Khin, "Exploiting loop-array dependencies to accelerate the design space exploration with high level synthesis," in *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2015, pp. 157–162.
[13] G. Zhong, A. Prakash, Y. Liang, T. Mitra, and S. Niar, "Lin-analyzer: a high-level performance analysis tool for FPGA-based accelerators," in *Proceedings of the 53rd Design Automation Conference*, Jun. 2016, pp. 136:1–136:6.
[14] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He, "COMBA: A comprehensive model-based analysis framework for high level synthesis of real applications," in *Proceedings of the International Conference on Computer Aided Design*, Oct. 2017, pp. 430–437.
[15] B. C. Schafer and K. Wakabayashi, "Machine learning predictive modelling high-level synthesis design space exploration," *IET computers & digital techniques*, vol. 6, no. 3, pp. 153–159, May 2012.
[16] A. Mahapatra and B. C. Schafer, "Machine-learning based simulated annealer method for high level synthesis design space exploration," in *Proceedings of the 2014 Electronic System Level Synthesis Conference*, 2014, pp. 1–6.
[17] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Transactions on knowledge and data engineering*, vol. 22, no. 10, pp. 1345–1359, Oct. 2009.
[18] K. Weiss, T. M. Khoshgoftaar, and D. Wang, "A survey of transfer learning," *SpringerOpen Journal of Big data*, vol. 3, no. 1, p. 9, May 2016.
[19] J. R. Doppa, J. Rosca, and P. Bogdan, "Autonomous design space exploration of computing systems for sustainability: Opportunities and challenges," *IEEE Design Test*, vol. 36, no. 5, pp. 35–43, 2019.
[20] S. Dai, Y. Zhou, H. Zhang, E. Ustun, E. F. Young, and Z. Zhang, "Fast and accurate estimation of quality of results in high-level synthesis with machine learning," in *Proceedings of the 26th IEEE Symposium on Field-Programmable Custom Computing Machines*, Apr. 2018, pp. 129–132.
[21] S. Liu, F. Lau, and B. C. Schafer, "Accelerating FPGA prototyping through predictive model-based HLS design space exploration," in *Proceedings of the 56th Design Automation Conference*, Jun. 2019, p. 97.
[22] A. Deshwal, N. K. Jayakodi, B. K. Joardar, J. R. Doppa, and P. P. Pande, "Moos: A multi-objective design space exploration and optimization framework for noc enabled manycore systems," *ACM Trans. Embed. Comput. Syst.*, vol. 18, no. 5s, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3358206
[23] Z. Wang, J. Chen, and B. C. Schafer, "Efficient and robust high-level synthesis design space exploration through offline micro-kernels pre-characterization," in *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2020, pp. 145–150.
[24] L. G. Martins, R. Nobre, J. M. Cardoso, A. C. Delbem, and E. Marques, "Clustering-based selection for the exploration of compiler optimization sequences," *ACM Transactions on Architecture and Code Optimization*, vol. 13, no. 1, p. 8, Apr. 2016.