IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 24 October 2024, accepted 29 November 2024, date of publication 4 December 2024, date of current version 12 December 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3511345

RESEARCH ARTICLE

Machine Learning for FPGA Electronic Design Automation

ARMANDO BISCONTINI[1,2], E. POPOVICI[1], (Senior Member, IEEE), AND A. TEMKO[1,2], (Senior Member, IEEE)

[1]Department Electrical and Electronic Engineering, University College Cork, Cork 21, T12 K8AF Ireland
[2]QT Technologies Ireland Ltd., Qualcomm, Cork 21, T23 YY09 Ireland

Corresponding author: Armando Biscontini (armando.biscontini@umail.ucc.ie)

ABSTRACT In the last decades, field-programmable gate arrays (FPGAs) have become increasingly important to the electronics industry, offering higher performance and lower power consumption as transistor technology continues to scale down. Machine learning (ML) algorithms have become pivotal in the electronic design automation (EDA) of FPGAs, enabling the learning of relationships between input and the desired output based on representative data properties rather than physical laws. As FPGA capacity expands, the EDA tools must also scale to handle larger, denser digital systems, and ML offers to fill the gap with resulting computational efficiency and improved solution quality. This study reviews ML methods utilized in FPGA EDA, from the perspective of formulated problems, input space representation, learned mapping, and methods used to achieve that. The work also presents the main ML methodologies and future challenges, serving as a roadmap for FPGA practitioners to navigate in the area of ML for FPGA EDA.

INDEX TERMS Artificial intelligence, electronic design automation, field programmable gate arrays, machine learning.

I. INTRODUCTION

Over the past decades, field-programmable gate arrays (FPGAs) have acquired increased importance in the electronics industry [1]. As transistor sizes continue to scale following the Moore's Law trend [2], [3] and new technologies pave the way for post-Moore scaling [4], FPGAs have shown a potential for higher performance and lower power consumption [5]. When compared to microprocessors and application-specific integrated circuits (ASICs) [6], they offer lower non-recurrent engineering costs, reduced development time, easier debugging, and reduced risk. Nowadays, modern FPGAs have started to replace ASICs in many fields where they can meet the same power, performance, and area (PPA) requirements. Since their inception in 1985, FPGAs have been pervasive in several different applications including signal processing, multimedia processing, cryptography, chip multiprocessor emulation, financial engineering, bioinformatics, automotive, and artificial intelligence (AI) [7], [8].

This trend indicates that the FPGA electronic design automation (EDA) tools need to also scale to perform more operations to complete the implementation of larger (and denser) digital systems. As FPGA capacity increases, the entire computer-aided design (CAD) implementation process from high-level synthesis (HLS) to routing could take a considerable amount of time, particularly in the case of high device utilization [9], [10]. The problem becomes more challenging given that processors used to run the CAD tools exhibit sub-linear scaling speed with transistor size, even when parallel processing is implemented. CAD runtime becomes a critical issue [11].

Machine learning (ML) algorithms have started to play an important role in FPGA CAD/EDA design. The ML algorithms allow learning the mapping between a given input and a desired output based on the properties of the data rather than based on laws of physics. Due to their intrinsic computational efficiency, the established mapping provides

The associate editor coordinating the review of this manuscript and approving it for publication was Massimo Cafaro.

more opportunities for exploratory analysis, trading off some degree of accuracy for quicker turnaround time. This leads to an improvement in the solution quality compared to the traditional CAD solution methods.

In this study, we examine data-driven methods within the context of FPGA design, specifically focusing on the learned mappings between input and output. Our goal is to present a detailed classification of ML techniques applied to FPGA EDA, with an emphasis on various design stages and application tasks. The works in [12] and [13] provide a systematic review of ML techniques used across the FPGA implementation flow, offering an in-depth discussion of methodologies, approaches, and key findings. Building on these reviews, we offer a broad overview of the literature, adding a systematic analysis of input space representations and their learned mappings. The survey in [14] presents the integration of ML techniques in the HLS-based FPGA backend design flow, focusing on research challenges, limitations, and opportunities. Extending this scope, we further explore ML applications at the register transfer level (RTL). Additionally, [15] provides a concise overview of deep learning methods for end-to-end FPGA bitstream generation, highlighting the importance of open-source data for training large models. Inspired by this, we broaden our analysis to include traditional ML approaches and build on the survey in [16] by offering a detailed analysis of existing EDA datasets.

The paper is organized as follows. Section II provides the background on the EDA process and describes its main steps. Section III conceptually outlines relevant ML methodologies such as leveraging past data and learning from the current design on the fly. The studies that utilize ML for the main EDA steps, namely, HLS, RTL development, logic synthesis and optimization, placement, and routing are reviewed in Sections IV-VIII. A summary of the benchmark datasets employed in these studies is provided in Section IX. Section X examines common ML methodologies. Sections XI and XII address the discussion and outstanding challenges related to the application of ML in FPGA design. This review serves as a comprehensive roadmap for practitioners in the FPGA field.

II. CONVENTIONAL EDA FLOW FOR FPGA

FPGA hardware implementation relies heavily not only on its architecture but also on its EDA backend flow [17]. Large tech companies such as Synopsys, Cadence, Intel, AMD, and Mentor provide part or entire automation suites for FPGA implementation. There is also an increasing body of work in the open domain [18], [19], [20], [21], [22], [23]. An overview of the conventional FPGA implementation flow is presented in Fig. 1. Here, we introduce two flow variants: (a) RTL-based and (b) HLS-based. The distinction between them lies in their design generation methodologies. In the RTL flow, the process is manual, requiring engineers to handcraft the high-level description language (HDL) code, which makes development time variable and heavily reliant on the designers' expertise. Conversely, the HLS flow automates this step by taking an abstract behavioral specification of the system as input and generating the corresponding HDL code. Once the HDL is defined, both flows converge on the same implementation process, encompassing logic synthesis and optimization, placement, and routing, before the final translation into bitstreams for FPGA deployment. HLS, which emerged in the early 2000s, aims to elevate the level of design abstraction, reduce programming effort, and accelerate time to market. Analyzing runtime data from the Vivado monolithic configuration with 32 processors, as reported in reference [24], we estimate that in an HLS-enabled flow, approximately 46% of the total workflow time is dedicated to HLS, while logic synthesis and optimization account for 36%, and placement and routing each contribute around 9% to the overall runtime.

Below we briefly describe the five main steps of the FPGA implementation process.

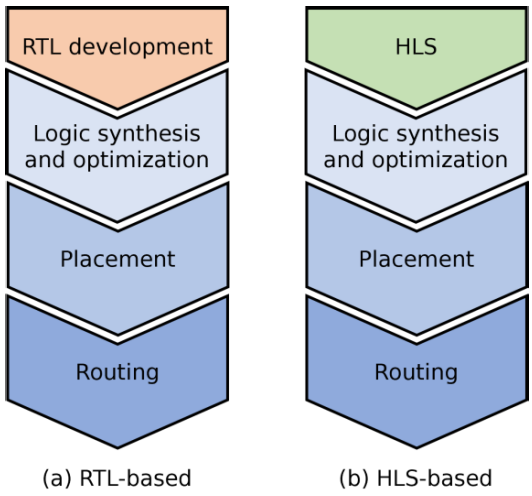


FIGURE 1. FPGA implementation flow.

A. HLS

During this step, a hardware behavioral description in C/C++/MATLAB is translated to an HDL such as VHDL or Verilog [25]. HLS is being adopted by more companies as part of their industrial EDA flow, even though it has not fully replaced the use of RTL flows yet. It can target large and complex chip architectures enabling a smooth transition from architecture to final RTL design, leading to a higher level of abstraction and design re-usability. It enables a larger vision of the overall system architecture, allowing microarchitecture generation to happen automatically by simply changing the synthesis directives, called "pragmas". This opens a path for direct system implementation without the use of HDLs. The pragmas configuration drives the generation of different micro-architecture flavors for the same design so that the generated hardware can be re-tuned to fit into specific PPA targets without modifying its behavioral description. On top of allowing rapid design and verification cycles, this enables faster Design Space Exploration (DSE). Historically, this

area has been dominated by traditional algorithms to solve multi-objective optimization problems. We can distinguish two categories: (1) integer linear programming [26] and (2) heuristics [27] algorithms. (1) usually results in optimal solutions. As such, they suffer from scalability issues with run times that are exponential in the worst-case scenario. On the other hand, (2) achieve sub-optimal solutions with better runtime complexity at the cost of higher sensitivity to hyper-parameters configuration. We can further classify heuristics into two main categories: design space pruning techniques and meta-heuristics. Design space pruning techniques aim to restrict the design space by pruning some solutions. This could lead to missing entire exploration regions. Meta-heuristics are based on phenomena frequently found in nature and aim to perform extensive searches in the design space, often tolerating worse local optima to find better sub-optimal solutions. Examples of these algorithms include simulated annealing [28], genetic algorithms [29], and ant colony [30] optimizations.

HLS comes with its limitations. HLS tools generally produce designs with lower performance and less efficient resource utilization compared to manual RTL implementations. Despite recent advancements, substantial quality of results (QoR) gaps persist, particularly in performance and resource optimization areas [31]. The effectiveness of HLS is often constrained by the specific tools and frameworks used, which can limit the designer's control over the final hardware implementation. RTL is frequently preferred for its superior fine-tuning capabilities, especially in cycle-accurate designs. While HLS can automate certain optimizations like pipelining and loop unrolling, it often struggles with more complex tasks. Designers may need to manually refine the code to achieve optimal performance, particularly when dealing with dynamic data structures and memory hierarchies. Although HLS facilitates behavioral verification, the resulting RTL code can also be difficult to debug and verify at a detailed level. The opaque relationship between high-level code and generated RTL complicates issue identification and the resolution of logic redundancies. In terms of performance relative to cost, the study in reference [32] evaluated productivity by comparing code quality against development hours. The average productivity of HLS was reported to be up to 6× greater than that of RTL design, thereby enabling designers to accomplish their tasks more efficiently. The research indicated that the design time required for HLS was approximately one-third of the time needed for RTL design, while the size of the HLS input code was nearly halved.

B. RTL DEVELOPMENT

RTL development is a methodology for designing digital circuits, with "register-transfer" referring to the abstraction level at which the design is conceptualized. At this level, the design is described in terms of data flow between registers and the logic operations performed on that data, without details on the transistor-level hardware implementation. RTL

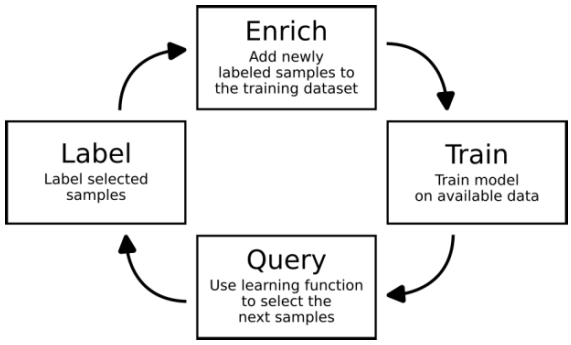


FIGURE 2. Active learning loop.

development is typically divided into two stages. The first stage, RTL coding, involves formulating and optimizing the initial RTL description. This process includes writing RTL code that accurately represents the system's data flow and operations while optimizing the design to meet specific PPA requirements. Techniques such as pipelining, which divides the design into stages that can operate in parallel, and parallelism, executing multiple operations simultaneously, are commonly employed to enhance design speed. Power optimization methods, such as clock gating, which disables the clock signal to unused parts of the design, and power gating, which cuts off power to inactive sections, are used to reduce power consumption. Area optimization can be achieved through resource sharing, where multiple components use the same resources, and constant propagation, which simplifies operations by broadcasting constants through the design. The second stage is RTL verification, which ensures the correctness of the RTL design. This stage involves code simulation, where a series of input vectors is applied to the RTL design, and the resulting output vectors are observed to assess whether the design behaves as expected. Formal verification is also employed to mathematically prove the correctness of the RTL design. In this approach, the intended behavior is expressed as a set of formal properties, which are verified using formal verification tools to ensure they hold true for all possible input vectors. Additionally, linting is a valuable technique used for ensuring design correctness, involving static analysis of the RTL code to identify common coding errors, ensure adherence to coding standards, and detect potential synthesis and timing issues.

C. LOGIC SYNTHESIS AND OPTIMIZATION

This step takes as input the high-level description of the hardware and translates it into a gate-level netlist [33]. The multi-objective search makes this step very demanding in terms of resources. It is generally divided into two sub-steps. First, a technology-independent synthesis is run, where the objective is to represent the netlist as a directed acyclic graph using technology-agnostic gates (e.g. AND, OR, NOT gates) and optimize it. The most common representation is the And-Inverter graph (AIG) where the nodes perform 2-input NAND operations [34]. Technology-independent synthesis

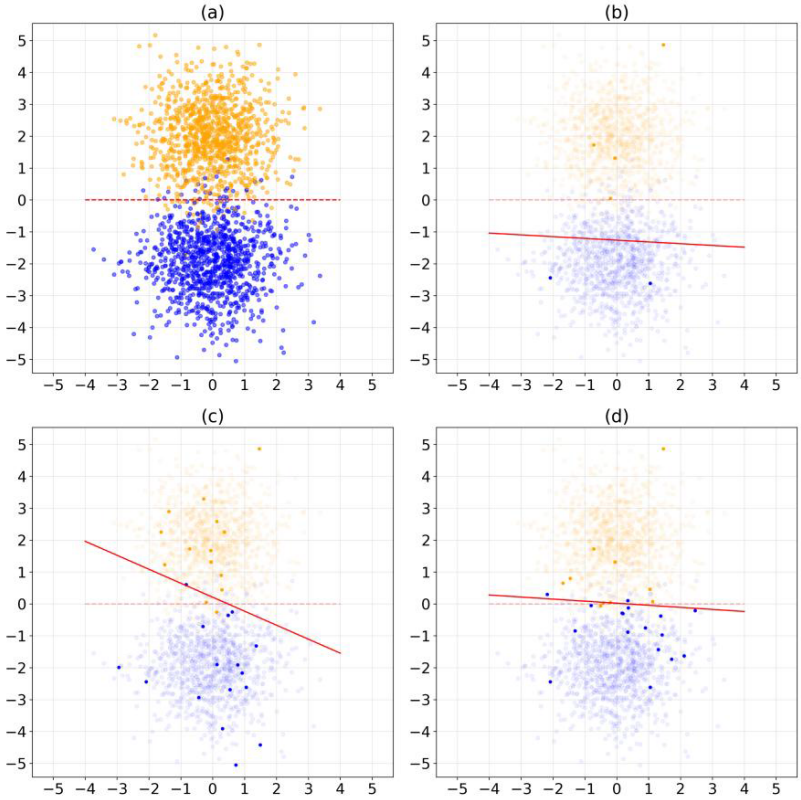


FIGURE 3. Classification task using active learning: (a) Initial distribution; (b) stage 1, learning using random sampling; (c) stage 2, learning using random sampling; (d) stage 2, active learning using uncertainty sampling.

does not present differences between the ASIC and the FPGA flows. It is followed by technology-dependent optimizations where the Boolean network is transformed into a network of gates derived from the technology library. Usually, this transformation involves the use of mapping and packing algorithms aimed at the concurrent reduction of cell area and power, where the output netlist is predominantly composed of lookup tables (LUTs) and flip-flops (FFs). The logic implementation space is very large and there is a wide margin of potential optimizations. Typical metrics targeted during the optimization process are performance and cost. The former is targeted by improving the hardware's latency for increased speed while the latter is achieved by reducing the number of mapped devices. The large optimization space also requires solvers with very high accuracy and complexity since the synthesis and optimization process often must obey many input constraints.

D. PLACEMENT

The technology-mapped netlist is ready to be placed on the FPGA. This step aims to place the logic cell instances while optimizing the routing resources [35]. The connected blocks are placed within the FPGA core area to minimize wire density, timing, and routing congestion. Historically many placement algorithms in the literature can be divided into three large categories based on their underlying methods: partitioning-based [36], analytical [37], [38], and simulated annealing placement techniques [39], [40]. Partition-based techniques recursively divide the circuit, reducing the problem dimension to smaller placement areas. This approach shows better runtime, sometimes impacting negatively on QoR. Analytical placement reduces the problem to a function minimization setting, where the solution is represented by the block locations that minimize a set of goals under user-provided constraints. Simulated annealing is a stochastic global search optimization algorithm. It tends to achieve good quality results at the expense of larger runtime.

E. ROUTING

This step represents a combinatorial optimization problem. It is especially complex for modern FPGA architectures where routing resources are limited and discrete. After placement, cell instances need to be connected through physical routes. To perform the routing task, the underlying architecture is modeled as a weighted graph. Each logic edge is associated with a single routing resource on the FPGA [41]. The edge weight represents the congestion present on its associated physical connection. Routing approaches can be classified into sequential routing where nets are routed following a pre-defined sequence, and concurrent routing where nets are routed at the same time. One famous example of sequential approaches is Dijkstra's shortest path algorithm,

as applied in PathFinder [42], where signal routing follows an iterative negotiation-based strategy. The negotiation happens when multiple signals share the same wire in the initial routing stage. These signals would go through a rip-up and re-route process, where assignment conflicts would be resolved. Concurrent routers are often implemented using network flow [43], linear assignment [44], and Boolean satisfiability [45].

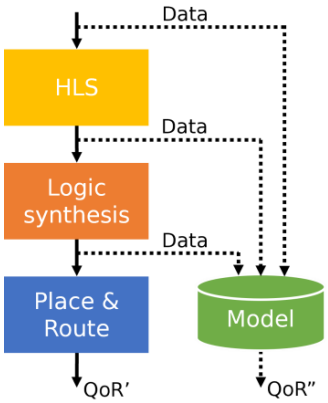


FIGURE 4. QoR prediction.

III. ML IN THE FPGA DOMAIN

ML systems aim to create a functional mapping between input and output based on available labeled samples. These data can be available beforehand and can be used to create an ML system to operate under an independent and identically distributed assumption (IID). In the context of FPGA, each design may consist of characteristics specific to its architecture. In these cases, the availability of past data may provide little usefulness for the design at hand since the IID assumption for a given representation may hold no more. In this situation, active learning is performed where an ML algorithm learns the design-specific mapping from scratch by guiding the optimizer on what data should be selected and subjected to labeling.

The active learning strategy facilitates building an ML solution when labeled data is not available beforehand or such data is not relevant or when labeling the data is very expensive. The active learning cycle, as depicted in Fig. 2, may start with a small, labeled training dataset, on which the initial model is built. This model is then used to guide the selection of unlabeled samples that can be subjected to an annotator to get the labels. The sampling is designed in a way to maximize the gain. The sampling algorithm learns from the acquired labels and leverages them to query the next batch of samples. The ML algorithm simultaneously proceeds with its (re-)training process supervised.

Fig. 3 depicts an intuitive illustration of active learning. Fig. 3(a) shows a data set generated from two Gaussians centered at $(0, -2)$ and $(0, 2)$ with a standard deviation $\sigma = 1$, representing two classes. The full available data set is composed of 2000 samples represented as points in a 2D feature space. The dashed red line represents the Bayes optimal decision boundary that separates the two classes. Let us assume we have a limited querying budget of 30 samples. Fig. 3(b) presents stage 1 of supervised learning: we randomly select 6 instances for initial labeling from the unlabeled pool. The line shows the decision boundary of a linear support vector machine (SVM) classifier trained using these 6 points. The model achieves a classification accuracy of 87.4% on the remaining unlabeled population. In Fig. 3(c) and 3(d), we show stage 2 of the learning process where we query for training an additional batch of 24 samples using two different strategies: random sampling and active learning. Notice that in the random sampling case in Fig. 3(c) most of the labeled instances selected for incremental training are far from zero on the vertical axis, which is where the optimal decision boundary should be (dashed red line). As a result, the SVM classifier trained on 30 samples altogether achieves the test accuracy of 87.5%. On the other hand, the active learner in Fig. 3(d) uses uncertainty sampling to improve the position of the decision boundary, focusing on samples closest to the boundary. As a result, it avoids requesting labels for irrelevant samples. The SVM model trained on ''actively'' selected 30 samples, achieves a test accuracy of 97.3%.

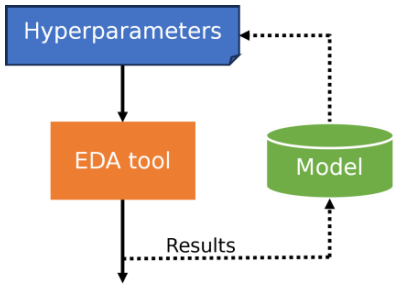


FIGURE 5. Hyperparameter optimization.

Active and passive ML learning methodologies are quite common in the FPGA domain to learn different mappings. The main use cases of the learned mapping are the following:

A. QoR PREDICTION

These models serve the purpose of swiftly evaluating the QoR of a design in the context of power, area, and delay, helping engineers efficiently filter out unsatisfactory designs. This eliminates the need for time-consuming simulations and traditional EDA runs (Fig. 4), typically run using timing or congestion estimations engines. As a result, the design cycle is typically reduced with the possibility of trading runtime for prediction accuracy.

B. HYPERPARAMETER OPTIMIZATION

The setup of EDA tools, including decisions related to algorithms and hyperparameters, significantly shapes their performance and outcomes. ML models aim to replace exhaustive or experimental approaches during configuration exploration. It may be used to assist or augment traditional optimization methods to perform hyperparameter searches

to find a better or optimal set of solutions quicker than the classical approaches (Fig. 5).

C. EXPLORATION GUIDANCE

Numerous tasks within EDA revolve around DSE, aiming to locate optimal design points within a large design space. This area is predominantly dominated by active learning techniques. Usually, surrogate ML models are used to guide exploration, as opposed to relying on precise analytical models or traditional hill-climbing methodologies. The model learns from past design cases, infers potentially better exploration directions, and predicts outcomes for new design points (Fig. 6). After pruning sub-optimal design options a subset of optimal design points, the Pareto front [46], is found with considerable runtime speed-up.

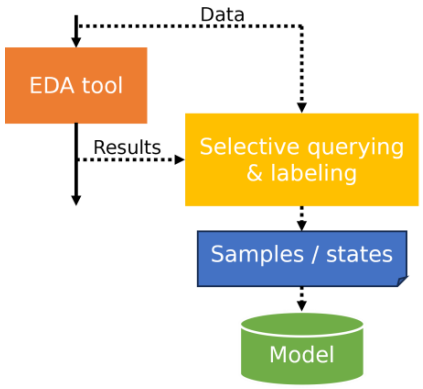


FIGURE 6. Exploration guidance.

D. AUTOMATION

Certain EDA tasks can benefit from the use of AI for the automation of design tasks that traditionally rely heavily on human involvement. In this context, deep learning and reinforcement learning (RL) find their best application. AI algorithms aim to replace human intervention creating no-human-in-the-loop flows and fully automating the design and verification cycle for FPGA (Fig. 7).

In the following sections, we analyze ML techniques typically used in the major steps of the EDA flow for FPGA. We focus on highlighting input-output ML components (features-targets relationship) for each of the papers and the results obtained compared to state-of-the-art techniques. In Table 1 we report a classification of reviewed publications based on their EDA stage and application task.

IV. HLS

A. QoR PREDICTION

The ability to perform early and accurate congestion estimation would greatly benefit the degree of architectural optimization in HLS. This would allow higher implementation efficiency and faster design cycles. However, existing HLS design tools are unable to predict routing congestion. Routability can be a serious concern in FPGA designs since it can directly impact the implementation feasibility of a design. It is difficult to evaluate in HLS without analyzing results after the full place and route implementation. Supervised ML models have been exploited to predict routing congestion.

Zhao et al. [47] built a congestion prediction model in FPGA HLS to help designers detect and eliminate congestion issues in the HLS code. The authors extracted 302 related features from the HLS and intermediate representation codes to capture the characteristics of each low-level HLS operator. Specifically, the features aimed to characterize interconnection, local circuit complexity, resource, and timing-related information of the low-level operators. A feature importance study demonstrated that congestion metrics were highly influenced by a subset of operators features such as resource utilization and global interconnection to/from multiplexers and memories. The framework aimed to predict horizontal, vertical, and average routing congestion. The authors trained and compared various regression models, including linear regression (LR), artificial neural network (ANN), and gradient-boosted decision trees (GBDTs). GBDT regression models outperformed the other models. In addition, they demonstrated that by discovering the bottlenecks in high-level source code with ML, routing congestion could be resolved easier and quicker, compared to the efforts involved in RTL-level implementation and design feedback.

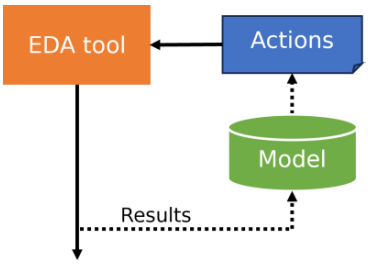


FIGURE 7. Automation.

Various ML research efforts aim at estimating area vs. performance tradeoffs. HLS code offers sophisticated techniques to optimize designs for area and performance using pragmas. However, resource usage and timing estimated by the HLS tools are often inaccurate and significantly different from the actual results achieved after the place and route implementation on FPGAs. These estimation inaccuracies prevent designers from performing valid DSE without going through the full implementation process.

Several works [48], [49], [50], [51], and [52] studied the suitability of learning from past designs and transferring such knowledge to a new design, for the purpose of DSE.

Wang et al. [48], [49] showed an ML-based methodology to perform area vs. performance tradeoff estimation. They extracted features such as synthesis directives of loops and arrays, target synthesis frequency, delay of functional units, bit-width of input and output signals, and loop iterations of all the loops in the HLS code. They used these features to train several ML models to estimate area and latency targets for a single design. The models used ranged from LR to more sophisticated random forest (RF), based on the input data

TABLE 1. Classification of the reviewed publications by EDA stage and application task.

	QoR prediction	Hyperparameter optimization	Exploration guidance	Automation
HLS	[47], [48], [49], [50], [51], [52], [60]	[54], [62]	[64], [65]	[70]
RTL development	[73], [74], [75], [76]	[77]	[78], [79]	[80], [83], [84], [85], [87]
Logic synthesis	[88]	[90], [91], [93]	[96], [100]	[102], [103], [105], [107], [110]
Placement	[112], [114], [115], [117], [120], [121], [122]	[123]	-	[124], [126], [127], [128], [129]
Routing	[131], [132], [133], [134]	-	[135]	[137], [139]

complexity. The training was performed on data extracted exhaustively by exploring the search space of all possible synthesis attributes. One of the main contributions consisted of leveraging the compositional aspect of the models at inference time by identifying previously pre-characterized ones and mixing them to output predictions on unseen designs. The identification of the models was enabled by the generation of hash values for each model after it was translated to an abstract syntax tree [53] and encoded into a multi-dimensional vector. The proposed method was shown to be very effective in identifying the Pareto front in a DSE task and 3× faster when compared to a meta-heuristics-based method [54] and a probabilistic multi-knob explorer [55].

Goswami et al. [50] demonstrated the performance of GBDT models to predict post-synthesis hardware resources and latency QoR metrics. They extracted a total of 66 features from the four different sources: HLS code (for loops, arrays, function, and datatype-related features), intermediate representation code (features describing basic block information such as number and type of instructions), control and dataflow graph (features representing the basic blocks interactions like variables propagation and dependencies), and call graph (features representing the relationship of functions in the code). Comparing several regression methods GBDT led to the best accuracy. When coupled with a simulated annealer setup for DSE, the model achieved similar results as a full DSE that performed logic synthesis for each newly generated design, with a reported runtime improvement of more than 12×.

Similarly to [50], Ferretti et. al [51] used ML to address the prediction of latency and resource utilization at the HLS stage, using graph neural networks (GNNs). They extracted features to capture elements of the HLS process and the data-processing tools. These features were extracted from the HLS code into a hybrid control and dataflow graph representation. The nodes of the graph represented different constructs of the program, such as loops, functions, and arrays. The edges encoded the data and control dependencies between the nodes. The aggregated node features consisted of 27 attributes describing the node type (loop, read, load, write, and function blocks), node attributes (the number of instructions in a block, number of iterations of a loop block, loop stride, and depth, loop trip count, number of function invocations, etc.), and associated pragma values (resource type, array partitioning, loop unrolling and function inlining pragmas). The edge feature vectors captured information regarding their type (control, data, or parameter). The prediction was achieved by training a message-passing GNN that would encode the graph structure into a single vector using a final global pooling operation. Regression heads would then predict the latency and resource utilization targets. The proposed framework outperformed the baseline represented by an ANN-based alternative [56]. The performance levels obtained were comparable to an HLS process simulator based on analytical models [57] for the latency prediction task, and to an estimation framework based on GBDT [58] for the resource prediction task.

ML was utilized to re-calibrate the results of HLS reports and provide an accurate prediction of post-implementation resource utilization and timing performance [52]. A total of 183 features were extracted from the HLS reports capturing performance information (requested clock period, estimated clock period, uncertainty), resources (utilization and availability of hardware blocks), operations (bit-width/resource statistics of operations), memory (number of memory words/banks/bits), and multiplexers (multiplexer input size/bit-width). Redundant and irrelevant features were identified and removed using Pearson's correlation analysis. The authors aimed to predict 5 targets: LUT, FF, digital signal processor (DSP), random access memory (RAM) utilization, and maximum supported frequency of an HLS design. They employed a stacked regression ensemble technique [59] to exploit complementary information coming from different base estimators used such as LR, ANN, SVM, and RF, to improve the overall model accuracy. Reported results showed that the average error improved when compared with the best base model alone.

In the early design stages, power prediction represents a challenge for designers. It is difficult to predict the correlation between power consumption and HLS-based applications. The current practice to obtain power consumption is through power measurement after hardware implementation or estimation through gate-level simulation, both of which require designers to spend substantial effort.

Lin et al. [60] implemented a framework for power modeling using ML. An initial set of 256 features was extracted from 3 main sources: HLS report, the intermediate representation code, and the finite state machine with a datapath model. Collected features could be categorized into architecture-related and activity-related. While architecture features provided information on the overall design structure, as estimated by the HLS tools, activity features represented the switching activities of different hardware components in the target design. The framework aimed to predict the power

consumption target in *mW* units of a design. The authors reported performance using various models based on Lasso, SVM, GBDT, and convolutional neural network (CNN) architectures. The CNN model achieved the best overall performance among all the learning models, followed closely by the GBDT model. The experimental results demonstrated that the framework could achieve accurate power modeling which is only 4% away from ground-truth power measurements. The framework also achieved lower prediction error when compared with the state-of-the-art work [61].

B. HYPERPARAMETER OPTIMIZATION

Hyperparameter tuning is a fundamental step involved in meta-heuristics approaches used for HLS DSE. Due to the large exploration space, meta-heuristics are often a preferred approach compared to exhaustive searches. In this area, techniques have emerged in supervised learning and have been used to explore the best combination of meta-heuristics hyperparameters.

Wang and Schafer [54] presented an ML-based approach for automated setup of search parameters for HLS exploration meta-heuristics. The input features were extracted from the HLS code and represented the number of explorable operations (arrays, loops, and functions) and the number of functional units present in the code. Prediction targets included the meta-heuristic parameters to be set for search optimization algorithms studies such as simulated annealing (e.g. temperature descent increment, mutation rate, etc.), genetic algorithm (e.g. cross-over rate, exit condition, etc.) and ant colony optimization (number of parallel ants, decrement % of pheromone, etc.). Different predictive approaches were tried ranging from LR to GBDT regression models. The GBDT-based solution achieved the best results and was selected as the final model. When compared to meta-heuristic methods with default parameters, the proposed method led to a 2× higher average accuracy in identifying the Pareto front in a DSE task with a negligible runtime overhead.

Mahapatra and Shafer [62] developed an ML algorithm for the selection of promising attribute combinations for simulated annealing-assisted DSE. They used a set of directives as features, such as loop unrolling, folding, pipelining, function expansion, and arrays implementation style. Their framework aimed to obtain attribute combinations having the most impact on the simulated annealing cost function (linear composition of area and latency metrics). After sorting the attributes based on their capacity to minimize the cost function, they used a decision tree-based algorithm [63] as an ML backbone. Experimental results confirmed that the approach was 30% faster than the standard simulated annealing algorithm while achieving comparable results in the identification of the Pareto front dominant designs.

C. EXPLORATION GUIDANCE

Manual DSE in HLS requires expert knowledge and represents a time-consuming phase. Various tools emerged to facilitate the design of hardware accelerators in achieving high-performance results in HLS. Most of these tools have leveraged the power of ML and heavily relied on the use of active learning to achieve the retrieval of better training data samples and the selection of more accurate models.

Yu et al. [64] presented a tool to predict performance and resource usage for a design. Each design was characterized by input features comprising design knobs such as array configurations and loop unrolling/pipelining. The proportions of hardware resource utilization consumed on the target FPGA, in terms of RAM, LUT, FF, and DSP blocks, were used as the targets. RFs were used to establish the non-linear mapping between the design characteristics and the targets. The targets were initially obtained from the HLS ground truth data through a first random sampling stage. Then the framework used the predictions from the model to refine the training dataset. Active learning was utilized to determine the most relevant samples to be evaluated and added to the training data. The model was constructed by interleaving the process of evaluating new samples and re-training the models with the updated dataset. The authors reported computing savings while achieving better results in terms of resource usage and latency with respect to hand-tuned designs. With the developed framework, the authors constructed a Pareto curve to help select the optimal trade-off between latency and resource usage.

Gautier et al. [65] designed a multi-objective DSE framework for optimizing FPGA HLS using active learning techniques. The framework aimed to create surrogate models of the design space. These models were then used to predict the performance of a given design without running it on the FPGA. To achieve that, features based on HLS loop and array directives were used to represent the design space. These features proved to be effective design knobs, due to their fine-grained control over the synthesized hardware. Throughput and latency were defined as prediction targets. ML algorithms, such as RF, gaussian process (GP), and radial basis function interpolator were explored to model the complex relationships between input characteristics and targets. A multi-armed bandit algorithm was utilized to balance exploration and exploitation of the design space while using active learning. The algorithm rewarded the models that provided the most accurate estimates of the Pareto front. When compared to the four selected algorithms in the literature [66], [67], [68], [69], results indicated better or similar performance levels with significantly reduced computation costs.

D. AUTOMATION

Scheduling is a crucial step of HLS. It takes care of assigning each operation to a specific cycle or control state. This process's output would be a finite-state machine representing the full implementation of the synthesized application. However, scheduling often depends heavily on heuristics that aim to improve the performance and flexibility of the order

of operations. At the same time, heuristics are designed by humans and prone to human bias/expertise level. ML aims to tackle scheduling automation using RL techniques.

Chen and Shen [70] proposed an RL-based scheduling agent interacting with the environment, where the observed scheduling state was extracted using three features: the current schedule, current feasible operation movements, and all possible operation movements. The state was represented using several 2D figures in a matrix form, where the rows represent different operations, and the columns represent different control steps. The agent took an action based on the policy and received a reward signal from the environment, which corresponded to a reduction in the total resource usage. It then used this feedback to update the policy, and this process was repeated until the policy converged, or the maximum iteration threshold was reached. The policy network was trained initially using supervised learning and further updated using RL. When compared to the ''as soon as possible'' scheduler [71] and to the optimal integral linear programming solution [72], comparable results were reported demonstrating the effectiveness of the proposed method in solving time-constrained scheduling problems.

V. RTL DEVELOPMENT

A. QoR PREDICTION

Rapid estimation of FPGA on-chip resource utilization for individual sub-circuit blocks early in the design process is essential for aiding designers in optimizing circuits at the RTL level. Similarly, power estimation is critical not only for validating hardware design concepts during development but also for optimizing and managing power, energy, and thermal performance in real time. As a result, significant research has been conducted on both area and power estimation methodologies using ML. Both supervised and unsupervised techniques have seen widespread adoption.

Zennaro et al. [73] developed a supervised learning approach to predict the area of hardware components based on abstract specifications. The study utilized both register-specific features, such as total contained size and bitfield size, and bitfield-specific features, including the number of bitfields read/written by peripheral devices and CPU instructions. Regression models, including RF, GBDT, and ANN were employed to predict the number of LUTs and slice registers in the design. The results demonstrated high prediction accuracy with a runtime improvement of 600× compared to traditional synthesis methods. However, the study primarily focused on specific hardware components, such as control and status register interfaces, and the generalizability of this approach to more complex designs remained to be evaluated.

Li et al. [74] developed a predictive model to estimate the on-chip resource utilization of RTL designs, facilitating their partitioning and integration into multi-FPGA systems. They identified and extracted numerous features, ultimately selecting 46 of them. These features encompassed the number of modules, registers, memories, counts of *always*, *case*, and *if* statements, bit width statistics, and the number of various operation types. The authors employed several ML models such as LR, RF, SVM, ANN, and GBDT, configured in both single-stage and multi-stage ensemble approaches. The prediction targets included estimations of LUT, FF, RAM, DSP, and CARRY block counts. The performance of their tool was evaluated by comparing its predictions with post-synthesis ground-truth data generated by Vivado in default mode. The results demonstrated that the ensemble techniques and the GBDT model in isolation achieved the lowest prediction error, with a minimum runtime speed improvement of 25×.

Kim et al. [75] proposed a real-time power modeling approach for RTL designs, which automatically identified key power-dissipating signals via clustering. An activity-based power model was constructed, and FPGA simulations were instrumented for rapid runtime power estimation. Using signal toggle densities as features, the model employed LR with polynomial terms to predict power dissipation across hardware modules. Compared to reference power traces from a commercial EDA tool, the method achieved less than 10% error, enabling power simulations to run in minutes rather than weeks.

Parthasarathy et al. [76] proposed an ML-based method to optimize regression test selection for RTL functional verification. The approach employed an ensemble of models, including logistic regression, RF, GBDT, ANN, and Naïve Bayes, to predict test failure likelihoods based on code changes. The primary goal was to reduce the test suite while maintaining validation accuracy. Features used included aggregate code abstraction metrics such as lines of code, complexity, and test coverage, collected less frequently, along with checksums for each code abstraction to efficiently identify code changes. Experimental results on multiple designs demonstrated high accuracy in detecting change-based failures, with up to a 10× reduction in recommended test sets.

B. HYPERPARAMETER OPTIMIZATION

As designs have grown, RTL verification has become significantly more complex. Researchers have sought to leverage RL and active learning techniques to replace human intuition in the hyperparameter tuning of simulations.

Hughes et al. [77] proposed an ML approach to optimize the design verification process by enhancing constrained-random testing with supervised and RL techniques. They aimed to improve the verification process by accelerating the coverage of hard-to-reach design states. Their method utilized features such as input control parameters, environment settings, design configurations, and coverage results indicating whether each functional coverage statement was hit during the simulation. These features were input into an ANN model to predict the optimal knob settings for subsequent simulation iterations. The approach was tested on a system aimed at improving hardware cache occupancy, demonstrating that their ML-based method consistently achieved higher

occupancy rates and superior coverage compared to the traditional constrained-random approach.

C. EXPLORATION GUIDANCE

DSE in RTL traditionally demands expert knowledge and is a time-intensive process. To streamline this phase, several tools used RL and active learning to optimize the design of processors and memory and maximize their performance.

Paletti et al. [78] introduced an open-source methodology for DSE that employed online learning to approximate costly black-box function evaluations during RTL synthesis. The framework combined a controller and an estimator working together. The controller, built on a multi-armed bandit architecture, optimized the trade-off between time and accuracy by deciding dynamically at each step whether to execute the exact or the estimated function. The estimator, leveraging decision trees, predicted design objectives, such as resource usage and throughput, from a feature vector representing the RTL context. This dual approach enabled simultaneous knowledge acquisition and decision refinement, enhancing overall performance. The proposed method achieved over a $2\times$ speedup while maintaining near Pareto-optimal solutions across various hardware design scenarios.

Ghaffari et al. [79] introduced a multi-model active learning framework aimed to assist RTL design. The system utilized Bayesian models to predict hardware performance and conduct DSE. It incorporated 12 micro-architecture design parameters as input features, including control latency, floating-point and fixed-point latency, number of physical registers, and sizes of L1 and L2 caches. It selected iteratively informative samples to develop Bayesian models of objective functions such as power consumption and latency. Previously acquired knowledge was leveraged using transfer learning techniques, enabling the modeling process for novel applications. Gaussian regression bootstrapping methods were used to reduce the number of samples needed for prediction. This approach led to a 65% reduction in sample requirements during the optimization of micro-architecture designs.

D. AUTOMATION

RTL debugging requires designers to analyze and troubleshoot hardware behavior to verify that the RTL code accurately represents the intended circuit functionality. The introduction of large language models (LLMs) has significantly improved this process. Their continuous advancements accelerate progress and drive innovation in the field.

Tsai et al. [80] proposed a framework that leveraged pre-trained large language models to automate the identification and correction of syntax errors in Verilog code. Acting as autonomous agents, LLMs iteratively interacted with error logs and human guidance through a feedback loop to debug code. Using prompt engineering techniques [81], the framework integrated reasoning and action steps in an interleaved fashion, while external retrieval mechanisms [82] provided access to relevant knowledge, such as databases or documents, for more accurate responses. This approach achieved a 98% error correction rate, notably improving syntax pass rates and reducing manual intervention.

A similar methodology to that described in [80] was employed by Yao et al. [83]. Their HDL debugging framework was enhanced through the incorporation of three key components: data generation, a search engine, and retrieval-augmented LLM fine-tuning. Data generation involved creating faulty HDL code via reverse engineering, which in turn produced error messages used for fine-tuning the LLM. The search engine extracted pertinent information from a document and code database to improve bug detection and code repair processes. The fine-tuning step further refined the LLM capability to produce accurate code solutions. This improved framework demonstrated at least an 80% performance increase in terms of pass rate over state-of-the-art LLM baselines, including those discussed in [80].

Thakur et al. [84] investigated the use of LLMs for Verilog code generation to enable hardware design automation. Unlike most prior research, which focused on models trained primarily on software code, their work centered on hardware design. They achieved this by assembling the largest curated dataset of Verilog code, sourced from GitHub repositories and textbooks, and fine-tuning pre-existing open-source LLMs. In contrast to earlier studies [80], [83], they further performed in [85] a comprehensive evaluation of the syntactical and functional correctness of the generated code using verification test benches. Their model showed a 40% improvement over the pre-trained version and outperformed GPT-3.5-turbo [86] in generating functionally accurate Verilog code.

Lu et al. [87] advanced the research beyond basic HDL syntax evaluation by presenting an open-source benchmark for hardware RTL generation with LLMs, which evaluated both functionality and post-synthesis PPA design metrics. They also proposed a ''self-planning'' prompt engineering technique that divided the prompting process into two steps: the first involved reasoning, which mirrored human problem-solving by planning and decomposing tasks into logical steps. The second stage generated the final RTL code. Experimental results demonstrated a significant improvement in GPT-3.5's RTL generation performance using this approach.

VI. LOGIC SYNTHESIS AND OPTIMIZATION

A. QoR PREDICTION

QoR prediction is important in the early stages of logic synthesis. Predicting accurate area and delay results is fundamental as calls to static timing analyzers are very expensive to perform at this stage.

Hu et al. [88] proposed an ML-based framework to predict the depth of the LUT logic after technology mapping and to estimate the delay of a gate-level circuit. The topology of the pre-mapped netlist and the behavior of technology mapping of each design were characterized as a set of 12 input features that aim to reflect the scale, size, complexity, fanout, and depth of a circuit, and to capture the nature of LUT mapping. The features comprised the number of primary inputs, the

number of gates with different numbers of primary inputs, the number of paths, and path length. A GBDT model [89] was used to learn the non-linear mapping between the input features and target logic depth. When compared to the traditional technology mapping algorithm, results indicated negligible estimation error with 8× runtime improvement.

B. HYPERPARAMETER OPTIMIZATION

ML finds application in the hyperparameter selection of logic synthesis meta-heuristics. By learning from past data, supervised learning techniques enhanced these algorithms by providing higher-quality parameters.

Kapre et al. [90] applied ML techniques to tune CAD tool parameters for the designer. The framework built the initial database of results from a series of preliminary EDA runs, based on pre-determined configurations. The authors leveraged learnings from these past results to iteratively adjust the tool settings and help the design meet timing. Features were extracted at the pre-synthesis stage and included information on the target device family, logic structure, and compiler settings like synthesis and place-and-route options. The feature space was mapped to QoR targets such as timing, area, and power metrics. A Naive Bayesian classifier was used for predicting good tool strategies. Principal component analysis was then applied to prune redundant parameters and focus only on influential ones. Towards the end of the optimization process, it applied statistical sampling on the selected parameters to further refine the results. The authors showed a significant improvement in timing QoR when compared both to non-optimized and optimized industrial benchmarks.

Xu et al. [91] devised an efficient parallelization scheme that enabled ML-based auto-tuners to perform multiple concurrent parameter exploration. The feature space included the various configuration options available in an FPGA CAD tool from logic synthesis to routing, such as clustering, timing criticality, max router iterations, etc. The sample labels represented QoR metrics such as timing slack, resource usage, or power consumption. The authors proposed to partition the global solution space at the pre-synthesis stage into promising subspaces. They iteratively constructed a space partitioning tree, where the root node of the tree represented the initial solution space, and each intermediate node represented a subspace. The computational resources would then be allocated to each subspace using the multi-arm bandit-based search approach [92]. Experimental results demonstrated promising improvements in terms of timing QoR and runtime over static partitioning and commercial DSE tools.

Ustun et al. [93] presented an ML-driven autotuning framework for design timing closure. In contrast to previous work [90], [91], [94], [95], where features were collected at the pre-synthesis level, the authors included features extracted from the stage of technology mapping and packing. The features captured information about resources such as the number of LUTs, registers, I/O pins, DSP, etc., and timing such as worst and total negative timing slack. They used a GBDT regressor to model the quality of the design points using respective post-PnR total negative timing slack as the target. The framework would run in iterations where a set of parameters would be sampled for prediction using the multi-arm bandit-based strategy [92]. Parameter configurations were then ranked by the regressor. Top-ranking configurations were evaluated through the complete FPGA tool flow to obtain the actual QoR results, pruning the remaining samples. If design specifications were not achieved at the current iteration, the model would use online learning to progressively improve the model accuracy. Results were indicative of faster timing closure compared to the literature [91].

C. EXPLORATION GUIDANCE

Synthesis flow exploration requires fast and efficient searches to be performed in large combinatorial spaces. In this area, active learning can provide valid guidance and suggest logic optimizations, efficiently navigating between exploration and exploitation.

Bayesian optimization was used in [96] to optimize the QoR of circuits during the pre-mapping logic optimization stage. The authors used surrogate GP to model the QoR data and exploited calibrated uncertainty estimation to suggest new synthesis operation sequences to evaluate. They used local trust-region acquisition function maximization to effectively handle high-dimensional data. Each data point in the feature space was represented as a sequence of operations (rewrite, resub, refactor, etc.) and encoded into a string where each character represented an operation. To measure the similarity between strings, the authors employed a sub-sequence string kernel technique [97], [98], detecting the number of common sub-strings. The QoR targets used were the relative area and delay improvements (%) compared to the resyn2 heuristics [99]. Results showed that the proposed Bayesian optimization approach outperformed other search strategies in sample efficiency and QoR.

In [100], Chowdhury et al. used active learning techniques to improve QoR prediction in pre-technology mapping optimization. The authors proposed an approach to fine-tune a model trained on past synthesis data to accurately predict the quality of a synthesis recipe for an unseen netlist. They reduced the cost of labeling samples for fine-tuning using a clustering-assisted active learning technique [101]. AIG graph and recipe features were used to represent the netlist. AIG features were extracted using a graph convolutional network (GCN) where each node in the AIG graph was characterized by type (primary input, primary output, or internal node) and the number of inverters in fan-in. Recipe features were extracted using a separate CNN and used to create a synthesis recipe embedding. The prediction target was estimated using an ANN-based regression head and represented the improvement in the number of nodes and logic depth in the post-synthesis logic graph. Results reported more than 2× improvement in run-time and comparable QoR for the fine-tuned predictor when contrasted to conventional techniques using actual synthesis runs.

D. AUTOMATION

ML improved automation at logic optimization and technology mapping stages. The increasing complexity of these steps often requires strong expertise from engineers and heavy reliance on EDA tools. The application of deep RL and supervised learning techniques in this area enabled production cycles with no human in the loop.

In [102], the authors proposed an RL algorithm to explore the solution space for an effective sequence of logic synthesis optimizations that can minimize the number of nodes and depth of the logic graphs. The algorithm included a GCN to extract features from an AIG logic network, which was then used as an input to the RL. The extracted features captured the structural properties of the logic network, such as the connectivity between nodes and the types of logic gates used. The RL algorithm learned a policy that mapped states to actions, where the goal was to maximize a reward signal that reflected the quality of the resulting circuit. The reward signal was defined as a function of the number of nodes and the depth of the logic graphs. Two reward settings were investigated with one focusing on optimizing the number of nodes in the AIG graph, and the other on optimizing the logic depth. Experimental results demonstrated that the proposed RL algorithm outperformed the state-of-the-art heuristics [99] and achieved a significant improvement over the baseline in both reward settings, demonstrating the effectiveness of the proposed approach for optimizing the process of logic synthesis.

Hosny et al. [103] proposed an RL methodology to explore the logic synthesis optimization space. The authors represented the state space using features extracted from an AIG graph such as the number of nodes, the number of primary inputs and outputs, the number of latches, and the number of gates. Their RL agent was able to explore the search space selecting actions from a set of 7 primitive logic transformations, (e.g. resub, rewrite, refactor, etc.). A multi-objective reward function was used to represent changes in design area and delay. The agent would receive the largest reward for reducing the design area while keeping the delay under a prespecified constraint value. The authors used an Advantage Actor-Critic framework [104] where a hybrid policy-based network (actor) and a value-based network (critic) would iteratively interact to navigate the environment. In a delay-constrained setting, the framework achieved an improved design area when compared to greedy exploration and expert-crafted results.

Zhou et al. [105] applied RL to determine a recipe of circuit optimizations for logic synthesis, intending to minimize the number of LUTs in the technology-mapped circuit. A set of 21 features was used to describe the characteristics of the circuit under optimization. These features were then fed as inputs to the RL agent, which learned to choose logic synthesis optimizations. To improve the stability and sample efficiency of the RL algorithm, Advantage Actor-Critic and Proximal Policy Optimization [106] techniques were utilized. The authors used feature-importance analysis to prune the set of features visible to the RL agent to improve its efficiency. When the model was evaluated on unseen circuits, improvements in the area reduction were reported when compared to conventional and previously published approaches [99], [103].

Qian et al. [107] proposed an RL framework for logic synthesis automation. The framework modeled two RL environments to tackle logic optimization and technology mapping problems separately. The logic optimization state space was represented by statistical features extracted from the current AIG graph and its previous experience, such as the number of logic nodes and levels of the logic graph. The agent performed synthesis transformations, altering the status of the AIG graph, and received a reward reflecting the performance improvement as a function of the number of nodes and levels in the logic graph. For technology mapping the authors used the multi-armed bandits as a baseline [108]. They improved it by making the RL agent directly learn the logic optimization and mapping sequence, without splitting it into a series of iterative stages. The state space was the same as the one used for logic optimization while the available actions contained the commands for both AIG and LUT mapping optimization. The adopted learning framework was based on the Proximal policy optimization model, which aimed to improve sample utilization. Experimental results were compared to state-of-the-art RL works in logic optimization [102], [109], and technology mapping [108]. Better QoR was reported in terms of logic depth and the number of nodes with a runtime reduction.

Neto et al. [110] proposed an ML-driven framework to choose the logic optimizer to deliver the best QoR for different portions of the logic netlist. A k-way netlist partitioning method was used to derive partitions and encode them into two-dimensional image-like representations. This approach extracted and projected the low-level Boolean logic features into a Karnaugh map representation [111]. The partition images were then fed to a deep-ANN classifier to predict which optimizer would achieve the best results in terms of the number of nodes and circuit area. Post-synthesis results showed QoR improvements compared to single optimizers used in isolation.

VII. PLACEMENT

A. QoR PREDICTION

Obtaining accurate post-routing timing delay and routing congestion estimations during placement is becoming increasingly challenging, as both the design size and target device complexity increase. Various research studies show how supervised learning can address these issues.

Martin et al. [112] used ML to predict net delays and reduce the mismatch between placement and detailed routing. A total of 20 engineered features were used to represent the characteristics of nets, available routing, and logic resources. These included net fanout, HPWL, width, height, average

congestion, maximum congestion, path length, I/O crossing, DSP crossing, etc. The prediction target was the post-routing delay of a net. Various ML models were utilized to learn the mapping, including RF, ANN, and GBDT. RF showed the best performance when compared to the static delay model in [113], reporting an improvement in critical path delay.

The framework in [114] presented a set of ML probes that could provide early QoR feedback to an open-source placement tool [39], guiding the placement to achieve improved designs at the post-routing stage. A set of 53 features was extracted during placement and represented both direct and indirect measures of congestion, such as wire length, channel width, and the number of moves attempted by the placer. A recursive elimination procedure was performed to discard redundant features. This step highlighted the importance of some features such as local wiring congestion, global wiring demand, and the number of moves as a measure of the effort expended by the placer. The prediction targets were represented by routed wire length, critical path delay, minimum channel width, routability, and short/long wire utilization in horizontal/vertical directions. The authors used a mix of LR and GBDT models for the mentioned regression tasks. Results showed that the learned mapping was helpful in the estimation of wire length, critical path delay, and segmented wire utilization, reducing the routing time of the tool.

Pui et al. [115] used ML to predict routing congestion on UltraScale FPGA placements. A set of 27 features was used to characterize the global routing cells of the designs, capturing information about the number of pins, the overlapping nets bounding box, and their combined effects. To augment the contextual representation, the neighboring routing cells information was also utilized. The congestion prediction target was represented by the percentage of used routing resources in each of the cells. A combination of LR and SVM models was used to learn the mapping in a local, hybrid, and global context. The model led to an improved average routed wire length during placement [116].

Maarouf et al. [117] proposed an ML-based method to estimate congestion using 4 high-quality features composed of net wire length, bounding box, pin count, and net cut on 5×5 and 9×9 neighboring regions. The prediction target was the congestion of each global routing cell in the design. The authors compared several regression models: LR, K-nearest neighbor (KNN), ANN, and RF. Feature engineering improved the accuracy reported in [37], [115], [118], and [119].

A deep learning framework for routability classification at the placement stage was presented by Al-Hyari et al. [120]. The model aimed to provide feedback to avoid costly place-and-route iterations and improve the placer efficiency when dealing with hard-to-route circuits. The same set of high-quality features from [117] was utilized. A CNN model was implemented to estimate the placement routability as a binary target. The authors integrated the model into a placer [37] and used it to optimize both global and detailed placement steps. Results reported high prediction accuracy with a considerable reduction in runtime.

Szentimrey et al. [121] studied the integration of 3 deep-learning models into an open-source analytic placement tool [37]. These models tackled 3 sequential placement steps. Initially, the same set of design features from [117] and [120] was extracted and provided to the first model [117]. This model predicted congestion at the global routing cells level. The congestion maps produced at this stage were fed to a second model that aimed to mitigate congestion by estimating the switch cell area inflation. A convolutional encoder-decoder architecture was used for this model, mapping input congestion maps to output LUT inflation maps. Lastly, the third model presented in [120] was used to assess the design routability. Results indicated a $2\times$ improvement in the placement runtime.

Martin et al. [122] explored the adoption of simple ML models for predicting placement routability and showed performance gains when compared to deep-learning techniques. Simple methods were initially investigated such as LR, KNN, GBDT, Naive Bayes, and SVM. They were further combined using bagging, boosting, and stacking techniques. A total of 46 features was collected to represent information regarding placement (HPWL and utilization), congestion (max, mean, histogram values, etc.), nets (wire length, routing network utilization, etc.), hotspots (their area, max intensity, etc.), and logic (cell counts, pin counts, etc.). Binary routability classification was set as the target for all models. The authors used a feature selection algorithm to identify the most important features and improve the efficiency of each model. SVM was the best-performing model when compared to the more complex deep-learning model in [120]. Ensembling the developed models further improved accuracy.

B. HYPERPARAMETER OPTIMIZATION

Placement parameters have dramatically increased as FPGA technology and tools progress. Researchers tried to leverage RL and active learning techniques to replace human intuition during the hyperparameter tuning step.

Kapre et al. [123] proposed a cloud-based ML engine to evaluate parallel placement runs and recommend multiple CAD parameter combinations to achieve timing closure. The learning started from an initial database of tool parameters with associated timing slack variation labels. Input parameters included information on the target device family, logic structure, and various CAD settings. The authors used a Bayesian learning and classification framework [90] to select promising parameters and principal component analysis to prune irrelevant ones. The exploration was organized in iterations where batches of parallel CAD validation runs were sequentially performed and used for learning. By balancing learning with parallelism, the framework achieved better total negative slack convergence compared to random placement seed exploration.

C. AUTOMATION

RL techniques have been used to improve several steps of simulated annealing placement and find better solutions.

Al-hyari et al. [124] proposed an ML framework to recommend the best placement flow. Since the placement tool performance depends on the characteristics of input designs, the authors approached the selection of the most effective placer as a classification problem. Their framework aimed to recommend between two open-source analytical placers, [37] and [125]. They extracted 28 circuit features such as the total number of LUTs, FFs, IOs, pin count, net statistics, etc. A feature selection study highlighted the importance of routability and congestion in the placement selection process. They compared different supervised learning methods: ANN, SVM, KNN, RF, and various GBDT techniques. Results showed good accuracy when the framework was used to select the best placer in terms of total post-routing wire length.

Murray et al. [126] proposed an RL model to select block types to move during a simulated annealing placement. The authors framed the move type selection problem in a K-armed bandit context, using a single state as input. This simplifying assumption limited the solution approach since the agent was not able to recognize different placement features such as the optimization progress, past moves history, logic structure, etc. The RL agent move decision was based on the selection of different types of blocks (LUT, RAM, DSP, etc.) to be randomly swapped. After selecting a move type, the swap action was performed and evaluated using the wire length and timing rewards extracted from the simulated annealing-based placer. The framework achieved 2× faster runtime when compared to the baseline placer in [40] while achieving the same QoR in terms of wire length and critical path delay.

Elgammal et al. [127], [128] devised an RL framework to achieve efficient placement exploration by enhancing the K-armed bandit problem setup in [126]. The authors used knowledge about the circuit structure, current placement, and timing information to intelligently suggest block moves during a simulated annealing-based placement. The entire placement process was divided into two stages, represented as input state features. The early stage focused on performing large cell movements and achieving a good global placement. The later stage was meant to further optimize timing, having more visibility on the critical paths. Based on the stage the agent was in, it decided whether to use actions more focused on wire length or timing optimization. The reward signal was based on the change in QoR placement, which included both timing, wire length, and runtime components. Experimental results showed superior performance in terms of QoR and runtime when compared to the baseline placer [40].

A wide variety of AI techniques remain currently unexplored due to the difficulty of directly integrating them into the FPGA CAD flow usually based on C++. Every improvement to the agent learning code would require a deep knowledge of the flow and trigger a recompilation to integrate the changes. Chen et al. [129] tackled the issue by presenting a framework based on [130] that simplified the implementation process of RL optimization algorithms into an open-source placer [39]. The RL system was formulated as an agent taking actions and receiving rewards from the placer environment. The user could easily direct the exploration by defining the optimization objective through Python code interfaces. At each placement step, the environment returned a state and a reward derived from an incremental simulated annealing iteration. This solution opened the possibility for researchers to quickly define the agent's behavior and allowed quick development and testing of new RL methodologies. The authors also demonstrated the superior performance of evolution strategy techniques over multi-armed bandit formulations [126], [127], [128], in terms of critical path delay and wire length QoR.

VIII. ROUTING

A. QoR PREDICTION

Congestion, routing iterations, and delay degradation predictions are rising research topics. While heuristics and tool simulations can help to estimate solutions for these problems, they often lead to long and impractical runtimes. Researchers applied ML techniques to find viable solutions.

Siddiqi et al. [131] proposed a supervised learning approach to predict congestion costs for routing resources. The feature space comprised the number of input/output logic pins connected to each channel route and the horizontal/vertical wiring used by each net. The authors trained various regression models such as LR, SVM, GBDT, RF, and KNN. The best-performing model was found to be RF. The models predicted six targets: input pins in horizontal/vertical channels, and wires in north, east, south, and west directions. The model was then used to guide an open-source router [40] and avoid highly congested regions. Results showed a large reduction in the number of routing and rip-up-and-reroute iterations, leading to large runtime gains.

Gunter and Wilton [132], [133] presented an ML framework to save time through the early exit of difficult routing problems. The authors considered the 79 features from [122], representing information about the current routing iteration, grid dimensions, net wire lengths, fanouts, resource, and switch-box utilization. They performed a feature importance analysis to highlight features generalizing well across different architectures, device sizes, and channel widths. A Mixture of Experts technique was used and included various classifiers to predict the success of the routing process and regressors to estimate the number of remaining iterations to completion. Compared to the baseline LR-based predictor in the open-source router [40], the framework had a higher success rate in identifying routable circuits and reducing time wasted on unrouteable circuits.

Ghavami et al. [134] leveraged deep learning techniques to predict the impact of aging-induced delay degradation, raising the modeling abstraction from the transistor level to the logic block level. The input features used for the prediction task included input signal probability, load capacitance, and transistor widths of each FPGA basic block. An ANN

TABLE 2. Benchmark dataset summary.

Benchmark	No. of designs	RTL	HLS	Logic synthesis	Place and Route
HDLbits [143]	182	×			
VerilogEval [144]	156	×			
Chstone [145]	12		×		
MachSuite [146]	19		×		
MediaBench [147]	19		×		
Polybench [148]	30		×		
Rosetta [149]	6		×		
S2CBench [150]	13		×		
Spector [151]	11		×		
MCNC [152]	20			×	
ISCAS 85 [153], [154]	10			×	
ISCAS 89 [155]	31			×	
IWLS 2005 [156]	84			×	
EPFL [157]	23			×	
OPDB [158]	27			×	
VTR 7 [159]	19			×	×
VTR 8 [40]	21			×	×
Titan23 [160]	23			×	×
Koios [161]	19			×	×
Koios 2.0 [162]	40			×	×
SymbiFlow [163]	6			×	×
Guelph Xilinx [164]	360				×
ISPD 2016 [165]	4				×
ISPD 2017 [166]	5				×

architecture was trained for each block type (LUT, FF, RAM, multiplexer, switch, and connections) to learn the mapping between its input aging parameters and output delay degradation. SPICE simulations were used to extract ground truth data for soft blocks (LUTs, RAMs, etc.) while hard blocks (e.g. DSPs), typically requiring larger simulation times due to their size, were modeled using hierarchical static timing analysis. When integrated into an open-source router [40], the framework showed a low prediction error rate and a greatly improved runtime compared to transistor-level simulation.

B. EXPLORATION GUIDANCE

Zheng et al. [135] used Bayesian optimization to guide the DSE of FPGA routing architectures. The authors designed the feature space to capture characteristics of the routing segments (lengths, positions, and directions) and signal-driving information (routing channels, logic, input, and output pins). At each optimization iteration, a test point was sampled by maximizing the expected improvement. After evaluating its objective value, it was used to train a GP. Pruning rules were used to further reduce the number of architecture evaluations required. The output was represented by the Pareto set of architecture parameters minimizing the area and delay objectives. Compared to a simulated annealing-based approach [136], the framework found better area-delay design points.

C. AUTOMATION

RL and unsupervised learning are some of the major approaches used to automate the routing and power-gating optimization processes.

Farooq et al. [137] reframed the routing problem into an RL setting. The authors used Q-learning to model a policy that maps the current routing state to an action that selects the next logic net to route. To deal with the non-stationarity of the problem, the authors used a tabular approach to maintain the action values for each routing node. An ε-greedy behavior policy was used to balance exploration and exploitation during the agent sampling process. The reward was defined as the change in the number of routing conflicts after each action. As a result, the agent tried to minimize the number of conflicts throughout the whole routing process. When compared to negotiation-based routing methods [138], the proposed technique delivered comparable QoR with reduced runtime. It also proved to be effective in handling heterogeneous FPGA architectures.

Seifoori et al. [139] presented an ML-driven approach to power gating in FPGA routing networks. The authors collected the utilization data of each multiplexer across the dataset to represent the features used to learn a partitioning strategy. K-means clustering was used to group available routing multiplexers into contiguous power-gating regions to reduce the total static power consumption of the design. The authors tested different variants of K-means clustering aiming to maximize a custom similarity metric based on common cluster patterns, instead of Euclidean distance. The output was a set of power-gating regions used to guide the power routing assignment. Results showed superior static power consumption results compared to the state-of-the-art heuristics [140], [141], [142].

IX. BENCHMARK DATASETS

Table 2 summarizes the most frequently used public benchmark DBs which are used in the reviewed papers. RTL datasets [143], [144] typically consist of multiple design examples coded in Verilog or VHDL, accompanied by cor-

TABLE 3. ML techniques analysis.

QoR prediction	Hyperparameter optimization	Exploration guidance	Automation
Supervised and unsupervised learning	Supervised and unsupervised learning	Active learning	Reinforcement learning

responding verification test benches, hardware specifications, and supplementary metadata designed to train LLMs for code debugging tasks. Recent initiatives to collect HDL code from literature and online sources have enriched these datasets, ensuring they remain comprehensive and up-to-date. Typical HLS datasets [145], [146], [147], [148], [149], [150], [151] provide just behavioral descriptions in C++ sampled across heterogeneous applications (cryptography engines, signal processing cores, etc.). The resulting circuits post-synthesis tend to be of reasonable dimension (usually less than 100000 LUTs), making them fast to synthesize and ideal for DSE exploration and Pareto front identification. Synthesis benchmarks [40], [152], [153], [154], [155], [156], [157], [158], [159], [160], [161], [162], [163] are larger and provide both Verilog/VHDL and BLIF synthesis files, that act as ground truth. Place and route DBs [40], [159], [160], [161], [162], [163], [164], [165], [166] are suitable for large-scale FPGA CAD research where it is not required the synthesis of new netlist primitives.

X. ML METHODOLOGY

After reviewing the above studies, it is possible to outline a common methodology followed by the authors. This can be useful for new practitioners entering the area and can act as a guideline.

A. PROBLEM FORMULATION

ML involves designing a system that learns a desired mapping from the provided data. This mapping can be in the form of a classification, regression model, or policy to guide an action. To formulate the problem, a researcher needs to visualize an ideal ML solution and how it impacts the area of study. We see this kind of approach being used in all the publications reviewed so far: authors define a problem (congestion estimation at the placement stage), establish a ground truth (congestion map produced by a global router), and identify the way to quantify the benefit of a successful approach (X% QoR and Y% runtime expected improvement).

B. DATA COLLECTION

The next step in the design is to gather the data from which the mapping can be established. The quality of this data determines the accuracy and reliability of the model. In the FPGA field, it is important to ensure coherency in the input samples (e.g. placement DB) and associated metadata (e.g. placer parameters, targeted FPGA architectures, etc.) to keep relevancy and correctness in the predictions. We see a trend from the research community to keep benchmark datasets extended and updated over time to match newer and larger FPGA devices. Strong efforts have been directed to increase data generalization towards different applications such as multimedia processing, cyber-security, signal processing, and cryptography.

C. DATA CONDITIONING

FPGA input data needs to go through a ‘‘cleaning’’ process where eventual discrepancies must be addressed (e.g. missing and duplicate values handling, unwanted data removal, data type conversion, etc.). This step usually leads to a fundamental restructuring of datasets where researchers try to ensure an even label distribution. Class imbalance could be a serious issue in EDA since there is a general tendency to discard negative samples (e.g. failed routed DBs) and keep positive ones (e.g. successful routed DBs). Data visualization is important at this stage since it can help to understand the dataset structures and identify the features-labels relationship.

D. DATA REPRESENTATION

Once the problem is formulated and relevant data is collected, a practitioner must figure out a way to represent the completeness of the data to ensure that the modeling efforts will be successful. Domain expertise plays a fundamental role in this process, guiding feature engineering and model architecture design to achieve better representation. One of the main steps in this process is the feasibility study that determines the learnability of the desired mapping from the given data and representation, the choice of the modeling methodology, and the architecture.

E. MODEL SELECTION AND TRAINING

After securing the learnability of the problem, a practitioner needs to ensure the generalization aspects to make the learned model useful in a real scenario. At this stage, the data is split into two sets, a training set, and a test set. This is to keep the data seen by the model during training separated from the one it is evaluated on. A specific model architecture is selected for training based on the nature of the data and the problem to solve. Every ML system can be completely described by the set of its parameters. Some of these parameters are automatically tuned during the learning process, like the weights of an ANN. Other parameters (hyperparameters) should be selected manually. This involves separating a specific part of the training dataset to be used for the internal validation process for guiding the hyperparameter tuning and model selection. We see the successful use of tree-based methods

TABLE 4. ML algorithms time complexity comparison.

Algorithm	Training time	Testing time
Linear regression	O(n·d)	O(d)
Logistic regression	O(n·d)	O(d)
Naïve Bayes	O(n·d·c)	O(d·c)
KNN	O(n·d)	O(k·d·log(n))
SVM	$O(n^2 \cdot d^2)$	O(k·d)
Decision tree	O(n·log(n·d))	O(log(n))
Random forest	O(k·n·log(n·d))	O(k·log(n))

and neural networks in applications such as QoR prediction and hyperparameter tuning. Active learning and unsupervised learning approaches are mostly used for exploration guidance. Automation is instead fully dominated by reinforcement learning techniques. At this stage, it is fundamental to avoid overfitting/underfitting issues, that would give performance mismatches between training and test datasets.

F. MODEL EVALUATION

Once the model parameters and hyperparameters are tuned, the performance is evaluated on the test set, previously unseen for all purposes. It is important to keep model selection and performance assessment routines independent and isolated. This effort aims to avoid compromising model performance evaluation that would otherwise lead to reports of over-optimistic results, not holding in a real scenario.

XI. DISCUSSION

A. ML TECHNIQUES ANALYSIS

In Table 3, we present a comprehensive classification of ML techniques across the various EDA application fields. Supervised learning methods encompass most of the techniques used in QoR prediction and hyperparameter optimization. Commonly employed techniques include LR, GBDT, SVM, GP, and KNN. Various neural network architectures, such as ANN and CNN, are also applied. Notably, GNN approaches, like GCN, are prevalent when netlist representations are required. Supervised methods typically approximate functions that map inputs to outputs based on input-output pairs, aiming to predict continuous or discrete supervisory signals. Examples of continuous targets include switching activity for dynamic power prediction, timing slack for performance estimation, and the number of logic elements (e.g., LUTs, RAMs) for area prediction. In classification problems, such as logic synthesis or placement hyperparameter prediction, supervised methods are used to predict likelihoods or class memberships. In contrast, unsupervised techniques are used to simplify problems by building concise representations of data. These methods often involve clustering, where the number of classes is unknown and is determined as part of the prediction process. Unsupervised learning is useful in tasks such as identifying key signals for dynamic power prediction. Active learning is particularly prominent in exploration guidance, where interactive querying and data labeling are necessary due to the high cost of ground truth data extraction. This approach is commonly applied in areas such as DSE and model selection for HLS and logic synthesis. Techniques such as Bayesian optimization are more adaptable and achieve better performance by selecting only the most informative samples. The automation domain is predominantly driven by RL, which is widely applied in tasks such as logic optimization and placement, where the goal is to develop no-human-in-the-loop methodologies. In this context, an autonomous agent seeks to identify the optimal approach for achieving a specific objective (e.g., finding the best logic optimization for area reduction). The agent makes decisions by leveraging both past feedback and exploring new strategies that could yield greater rewards. This process involves formulating a long-term strategy aimed at maximizing cumulative reward (e.g. logic area improvement).

B. TIME COMPLEXITY

Time complexity measures the number of operations an algorithm performs from start to finish, typically expressed as a function of input size. It reflects the time required for an algorithm to complete its task. This concept applies to both conventional and ML algorithms. However, ML algorithms differ in that they operate in two distinct phases: training and testing. The time complexity of these phases is analyzed based on dataset size and the number of features. Table 4 provides a comparison of typical time complexities for ML algorithms, as reported in [167], where the number of features is denoted as d, samples as s, classes as c, and trees as k. Algorithms like LR and Naive Bayes are relatively fast in both training and testing, making them suitable when computational efficiency is a priority. On the other hand, more complex models such as SVMs often achieve higher accuracy but require longer training times. A key advantage of ML algorithms is their ability to offload most of the computational complexity to the training phase, where all learning takes place. This one-time computational effort allows for significantly faster performance during testing, with inference times showing gains of 10× to 1000× over traditional algorithms, as demonstrated in various EDA applications discussed earlier.

C. INTEGRATION

The integration of ML into EDA software applications enhances capabilities such as predictions, data preprocess-

ing, and data generation. This requires multiple ML models to process data in real time, which introduces engineering challenges that differ significantly from those encountered in traditional software development. Unlike deterministic software, most ML models are stochastic, leading to uncertainties throughout the project lifecycle. Key challenges include managing and ensuring data quality, integrating ML with conventional code, and navigating the complex dependencies among source code, ML models, and external datasets. Effective integration of ML models needs proper encapsulation, often resulting in excessive glue code and intricate data-processing pipelines. Additionally, current version-control systems, such as Git, fall short in managing diverse ML assets, including datasets and model parameters. Popular ML frameworks like PyTorch, TensorFlow, and Scikit-Learn offer modular approaches for implementing models, allowing developers to treat models as reusable components. Models reuse is a vital strategy that minimizes development effort and improves performance, employing techniques such as transfer learning and the utilization of pre-trained models. The architectural design of ML-enabled systems is crucial for maintaining functionality and optimizing performance and maintainability, with modular architectures facilitating smoother integration.

XII. OUTSTANDING CHALLENGES

Upon reviewing the ML methodologies in the application areas of QoR prediction, hyperparameter optimization, exploration guidance, and automation, a few outstanding challenges can be highlighted:

A. ADAPTABILITY

Adaptability defines the ability of an ML system to adapt to reasonable changes and represents an important challenge in ML model design in the context of EDA for FPGA. Distribution shifts in the training data can degrade the performance of ML models when the input dataset is represented only by limited data gathered from various benchmark sources, as in most of the reviewed publications. These distribution shifts can often go undetected and hinder the final quality of the system, making the applicability of the final models limited to the data the model has been exposed to. This issue is not only present in the data collection and preparation stages but equally affects the training and evaluation steps. Data splits and feature engineering efforts need to be performed to avoid the introduction of any bias in the data and incorporate the required invariances (equivariances).

B. SCALABILITY CONSIDERATIONS

Balancing efficiency and effectiveness in ML workflows could be a major challenge, especially during the final deployment stage and adoption. In the reviewed literature, ML pipeline design efforts were limited to research purposes. Making the pipelines more manageable for real-life applications remains challenging. That would be quite useful not only at the model selection and training stage where frequent retraining of the ML models would be needed for multiple experiments, but also at evaluation time where low latency and high throughput constraints are more stringent.

C. EXPLAINABILITY

Explainability for AI has lately gained more importance. ML learns the mapping between input and output data and builds its decision process around this idea. With the evolving complexity models have acquired, these decisions may become unknown to researchers and engineers working on ML algorithm design. We see that explainability analysis of the results is not always a priority in the reviewed material. More effort should be put into the explainability study by understanding model drifts, biases, and feature impact on the prediction outcomes.

D. NEED FOR UPDATED OPEN-SOURCE DATASETS

Recent FPGA technology trends increase the need for up-to-date datasets. Benchmarks need to match new devices with netlists of increasing size. There is also a need to explore multiple application fields (cryptography engines, signal processing cores, etc.) to improve ML model generalization capabilities. We see an opportunity for generative AI and large language models to tackle this issue. These models can learn structural patterns in the training samples and generate new data that has similar features. Generative adversarial networks [168] and large language models [169] demonstrated their efficacy in other disciplines and could tackle this challenge.

E. FUTURE PERSPECTIVES ON HLS AND RTL FLOWS

With the recent advancements in LLMs and their capability to autonomously generate, analyze, and debug code, engineers are expected to receive enhanced support in designing increasingly novel and complex hardware architectures. The growing efficiency of LLMs, coupled with their ability to produce highly interpretable outputs, may address several limitations inherent to HLS, such as lack of output explainability and challenges in debugging and verification. Without significant innovation in these areas, RTL flows may continue to dominate, potentially at the expense of HLS adoption.

F. TOWARDS FULL END-TO-END ML FLOWS

The publications reviewed in this study solve a limited task (estimate post-routing congestion, tune the placer hyperparameters, etc.) in the larger context of EDA for FPGA. All the advantages gathered so far by the single tools could be compounded in a complete end-to-end flow that would lead to much greater performance and runtime benefits. We see the potential of building an all-around ML-driven EDA suite that would greatly benefit both the open-source academic and industrial worlds.

ABBREVIATIONS

Abbreviation	Full description
AI	Artificial intelligence
AIG	And-Inverter graph
ANN	Artificial neural network
ASIC	Application-specific integrated circuit
CAD	Computer-aided design
CNN	Convolutional neural network
DSE	Design space exploration
DSP	Digital signal processor
EDA	Electronic design automation
FF	Flip-flop
FPGA	Field-programmable gate array
GBDT	Gradient-boosted decision trees
GCN	Graph convolutional network
GNN	Graph neural network
GP	Gaussian process
HDL	High-level description language
HLS	High-level synthesis
IID	Independent and identically distributed
KNN	K-nearest neighbor
LLM	Large language model
LUT	Lookup table
LR	Linear regression
ML	Machine learning
PPA	Power, performance, and area
QoR	Quality of results
RAM	Random access memory
RF	Random forest
RL	Reinforcement learning
RTL	Register transfer level
SVM	Support vector machine

REFERENCES

[1] S. M. Trimberger, "Three ages of FPGAs: A retrospective on the first thirty years of FPGA technology," *Proc. IEEE*, vol. 103, no. 3, pp. 318–331, Mar. 2015, doi: 10.1109/JPROC.2015.2392104.

[2] G. E. Moore, "Cramming more components onto integrated circuits," *Proc. IEEE*, vol. 86, no. 1, pp. 82–85, Jan. 1998.

[3] R. R. Schaller, "Moore's law: Past, present and future," *IEEE Spectr.*, vol. 34, no. 6, pp. 52–59, Jun. 1997, doi: 10.1109/6.591665.

[4] S. Wang, X. Liu, and P. Zhou, "The road for 2D semiconductors in the silicon age," *Adv. Mater.*, vol. 34, no. 48, Dec. 2022, Art. no. 2106886, doi: 10.1002/adma.202106886.

[5] K. Underwood, "FPGAs vs. CPUs: Trends in peak floating-point performance," in *Proc. ACM/SIGDA 12th Int. Symp. Field Program. Gate Arrays*. Monterey, CA, USA: ACM, Feb. 2004, pp. 171–180, doi: 10.1145/968280.968305.

[6] M. J. S. Smith, *Application-Specific Integrated Circuits*. Boston, MA, USA: Addison-Wesley, 2003.

[7] P. H. W. Leong, "Recent trends in FPGA architectures and applications," in *Proc. 4th IEEE Int. Symp. Electron. Design, Test Appl. (delta)*, Jan. 2008, pp. 137–141, doi: 10.1109/DELTA.2008.14.

[8] S. Gandhare and B. Karthikeyan, "Survey on FPGA architecture and recent applications," in *Proc. Int. Conf. Vis. Towards Emerg. Trends Commun. Netw. (ViTECoN)*, Mar. 2019, pp. 1–4, doi: 10.1109/ViTECoN.2019.8899550.

[9] C. Mulpuri and S. Hauck, "Runtime and quality tradeoffs in FPGA placement and routing," in *Proc. ACM/SIGDA 9th Int. Symp. Field Program. Gate Arrays*. New York, NY, USA: ACM, Feb. 2001, pp. 29–36, doi: 10.1145/360276.360294.

[10] H. Bian, A. C. Ling, A. Choong, and J. Zhu, "Towards scalable placement for FPGAs," in *Proc. 18th Annu. ACM/SIGDA Int. Symp. Field Program. Gate Arrays*. New York, NY, USA: ACM, Feb. 2010, pp. 147–156, doi: 10.1145/1723112.1723140.

[11] S. Y. L. Chin and S. J. E. Wilton, "Towards scalable FPGA CAD through architecture," in *Proc. 19th ACM/SIGDA Int. Symp. Field Program. Gate Arrays*. Monterey, CA, USA: ACM, Feb. 2011, pp. 143–152, doi: 10.1145/1950413.1950443.

[12] P. Goswami and D. Bhatia, "Application of machine learning in FPGA EDA tool development," *IEEE Access*, vol. 11, pp. 109564–109580, 2023, doi: 10.1109/ACCESS.2023.3322358.

[13] V. Hamolia and V. Melnyk, "A survey of machine learning methods and applications in electronic design automation," in *Proc. 11th Int. Conf. Adv. Comput. Inf. Technol. (ACIT)*, Deggendorf, Germany, Sep. 2021, pp. 757–760, doi: 10.1109/ACIT52158.2021.9548117.

[14] I. Taj and U. Farooq, "Towards machine learning-based FPGA backend flow: Challenges and opportunities," *Electronics*, vol. 12, no. 4, p. 935, Feb. 2023, doi: 10.3390/electronics12040935.

[15] B. Ghavami and L. Shannon, "Unraveling the integration of deep machine learning in FPGA CAD flow: A concise survey and future insights," 2023, *arXiv:2303.10508*.

[16] D. Pal, C. Deng, E. Ustun, C. Yu, and Z. Zhang, "Machine learning for agile FPGA design," in *Machine Learning Applications in Electronic Design Automation*, H. Ren and J. Hu, Eds., Cham, Switzerland: Springer, 2022, pp. 471–504, doi: 10.1007/978-3-031-13074-8_16.

[17] D. Chen, J. Cong, and P. Pan, "FPGA design automation: A survey," *Found. Trends Electron. Design Autom.*, vol. 1, no. 3, pp. 195–330, 2006, doi: 10.1561/1000000003.

[18] X. Tang, E. Giacomin, B. Chauviere, A. Alacchi, and P.-E. Gaillardon, "OpenFPGA: An open-source framework for agile prototyping customizable FPGAs," *IEEE Micro*, vol. 40, no. 4, pp. 41–48, Jul. 2020, doi: 10.1109/MM.2020.2995854.

[19] I. Kuon, A. Egier, and J. Rose, "Design, layout and verification of an FPGA using automated tools," in *Proc. ACM/SIGDA 13th Int. Symp. Field-Program. Gate Arrays*. New York, NY, USA: ACM, Feb. 2005, pp. 215–226, doi: 10.1145/1046192.1046220.

[20] V. A. Ova and R. Saleh, "A 'Soft++' eFPGA physical design approach with case studies in 180 nm and 90nm," in *Proc. IEEE Comput. Soc. Annu. Symp. Emerg. VLSI Technol. Archit. (ISVLSI06)*, Mar. 2006, pp. 103–108, doi: 10.1109/ISVLSI.2006.1.

[21] J. H. Kim and J. H. Anderson, "Synthesizable standard cell FPGA fabrics targetable by the verilog-to-routing CAD flow," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 10, no. 2, pp. 1–23, Apr. 2017, doi: 10.1145/3024063.

[22] B. Grady and J. H. Anderson, "Synthesizable heterogeneous FPGA fabrics," in *Proc. Int. Conf. Field-Program. Technol. (FPT)*, Dec. 2018, pp. 222–229, doi: 10.1109/FPT.2018.00040.

[23] A. Li and D. Wentzlaff, "PRGA: An open-source FPGA research and prototyping framework," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*. New York, NY, USA: ACM, Feb. 2021, pp. 127–137, doi: 10.1145/3431920.3439294.

[24] Y. Xiao, D. Park, A. Butt, H. Giesen, Z. Han, R. Ding, N. Magnezi, R. Rubin, and A. DeHon, "Reducing FPGA compile time with separate compilation for FPGA building blocks," in *Proc. Int. Conf. Field-Program. Technol. (ICFPT)*, Dec. 2019, pp. 153–161, doi: 10.1109/ICFPT47387.2019.00026.

[25] W. Meeus, K. Van Beeck, T. Goedemé, J. Meel, and D. Stroobandt, "An overview of today's high-level synthesis tools," *Design Autom. Embedded Syst.*, vol. 16, no. 3, pp. 31–51, Sep. 2012, doi: 10.1007/s10617-012-9096-8.

[26] A. Schrijver, *Theory of Linear and Integer Programming*. Chichester, U.K.,: Wiley, 2011.

[27] S. Desale, A. Rasool, S. Andhale, and P. Rane, "Heuristic and meta-heuristic algorithms and their relevance to the real world: A survey," *Int. J. Comput. Eng. Res. Trends*, vol. 351, no. 5, pp. 2349–7084, 2015.

[28] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, May 1983, doi: 10.1126/science.220.4598.671.

[29] M. Srinivas and L. M. Patnaik, "Genetic algorithms: A survey," *Computer*, vol. 27, no. 6, pp. 17–26, Jun. 1994, doi: 10.1109/2.294849.

[30] M. Dorigo, M. Birattari, and T. Stutzle, "Ant colony optimization," *IEEE Comput. Intell. Mag.*, vol. 1, no. 4, pp. 28–39, Nov. 2006, doi: 10.1109/MCI.2006.329691.

[31] Z. Sun, K. Campbell, W. Zuo, K. Rupnow, S. Gurumani, F. Doucet, and D. Chen, "Designing high-quality hardware on a development effort budget: A study of the current state of high-level synthesis," in *Proc. 21st Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2016, pp. 218–225, doi: 10.1109/ASPDAC.2016.7428014.

[32] S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen, "Are we there yet? A study on the state of high-level synthesis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 5, pp. 898–911, May 2019, doi: 10.1109/TCAD.2018.2834439.

[33] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. New York, NY, USA: McGraw-Hill, 1994.

[34] A. Turing, "Intelligent machinery (1948)," in *The Essential Turing*, B. J. Copeland, Ed., London, U.K.: Oxford Univ. Press, 2004, pp. 395–432, doi: 10.1093/oso/9780198250791.003.0016.

[35] W. Wang, Q. Meng, and Z. Zhang, "A survey of FPGA placement algorithm research," in *Proc. 7th IEEE Int. Conf. Electron. Inf. Emergency Commun. (ICEIEC)*, Jul. 2017, pp. 498–502, doi: 10.1109/ICEIEC.2017.8076614.

[36] A. Khatkhate, C. Li, A. R. Agnihotri, M. C. Yildiz, S. Ono, C.-K. Koh, and P. H. Madden, "Recursive bisection based mixed block placement," in *Proc. Int. Symp. Phys. Design*. New York, NY, USA: ACM, Apr. 2004, pp. 84–89, doi: 10.1145/981066.981084.

[37] Z. Abuowaimer, D. Maarouf, T. Martin, J. Foxcroft, G. Gréwal, S. Areibi, and A. Vannelli, "GPlace3.0: Routability-driven analytic placer for ultrascale FPGA architectures," *ACM Trans. Design Autom. Electron. Syst.*, vol. 23, no. 5, pp. 1–33, Oct. 2018, doi: 10.1145/3233244.

[38] W. Li, Y. Lin, and D. Z. Pan, "ElfPlace: Electrostatics-based placement for large-scale heterogeneous FPGAs," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2019, pp. 1–8, doi: 10.1109/ICCAD45719.2019.8942075.

[39] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," in *Field-Programmable Logic and Applications*, vol. 1304, W. Luk, P. Y. K. Cheung, and M. Glesner, Eds., Berlin, Germany: Springer, 1997, pp. 213–222, doi: 10.1007/3-540-63465-7_226.

[40] K. E. Murray, O. Petelin, S. Zhong, J. M. Wang, M. Eldafrawy, J.-P. Legault, E. Sha, A. G. Graham, J. Wu, M. J. P. Walker, H. Zeng, P. Patros, J. Luu, K. B. Kent, and V. Betz, "VTR 8: High-performance CAD and customizable FPGA architecture modelling," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 13, no. 2, pp. 1–55, Jun. 2020, doi: 10.1145/3388617.

[41] S. Trimberger, "Effects of FPGA architecture on FPGA routing," in *Proc. 32nd ACM/IEEE Conf. Design Autom. Conf. (DAC)*. San Francisco, CA, USA: ACM, Jul. 1995, pp. 574–578, doi: 10.1145/217474.217592.

[42] L. Mcmurchie and C. Ebeling, "PathFinder: A negotiation-based performance-driven router for FPGAs," in *Proc. ACM 3rd Int. Symp. Field-Program. Gate Arrays*. Monterey, CA, USA: ACM, Feb. 1995, pp. 111–117, doi: 10.1145/201310.201328.

[43] S. Lee, H. Xiang, D. F. Wong, and R. Y. Sun, "Wire type assignment for FPGA routing," in *Proc. ACM/SIGDA 11th Int. Symp. Field Program. gate arrays*. New York, NY, USA: ACM, Feb. 2003, pp. 61–67, doi: 10.1145/611817.611828.

[44] Y.-W. Chang, K. Zhu, and D. F. Wong, "Timing-driven routing for symmetrical array-based FPGAs," *ACM Trans. Design Autom. Electron. Syst.*, vol. 5, no. 3, pp. 433–450, Jul. 2000, doi: 10.1145/348019.348101.

[45] H. Fraisse, A. Joshi, D. Gaitonde, and A. Kaviani, "Boolean satisfiability-based routing and its application to Xilinx ultrascale clock network," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*. New York, NY, USA: ACM, Feb. 2016, pp. 74–79, doi: 10.1145/2847263.2847342.

[46] J. Teich, "Pareto-front exploration with uncertain objectives," in *Evolutionary Multi-Criterion Optimization*, E. Zitzler, L. Thiele, K. Deb, C. A. C. Coello, and D. Corne, Eds., Berlin, Germany: Springer, 2001, pp. 314–328, doi: 10.1007/3-540-44719-9_22.

[47] J. Zhao, T. Liang, S. Sinha, and W. Zhang, "Machine learning based routing congestion prediction in FPGA high-level synthesis," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Florence, Italy, Mar. 2019, pp. 1130–1135, doi: 10.23919/DATE.2019.8714724.

[48] Z. Wang, J. Chen, and B. C. Schafer, "Efficient and robust high-level synthesis design space exploration through offline micro-kernels pre-characterization," in *Proc. Design, Autom. Test Eur. Conf. Exhibit. (DATE)*, Mar. 2020, pp. 145–150, doi: 10.23919/DATE48585.2020.9116309.

[49] Z. Wang and B. C. Schafer, "Learning from the past: Efficient high-level synthesis design space exploration for FPGAs," *ACM Trans. Design Autom. Electron. Syst.*, vol. 27, no. 4, pp. 1–23, Feb. 2022, doi: 10.1145/3495531.

[50] P. Goswami, B. C. Schaefer, and D. Bhatia, "Machine learning based fast and accurate high level synthesis design space exploration: From graph to synthesis," *Integration*, vol. 88, pp. 116–124, Jan. 2023, doi: 10.1016/j.vlsi.2022.09.006.

[51] L. Ferretti, A. Cini, G. Zacharopoulos, C. Alippi, and L. Pozzi, "Graph neural networks for high-level synthesis design space exploration," *ACM Trans. Design Autom. Electron. Syst.*, vol. 28, no. 2, pp. 1–20, Mar. 2023, doi: 10.1145/3570925.

[52] H. M. Makrani, F. Farahmand, H. Sayadi, S. Bondi, S. M. P. Dinakarrao, H. Homayoun, and S. Rafatirad, "Pyramid: Machine learning framework to estimate the optimal timing and resource usage of a high-level synthesis design," in *Proc. 29th Int. Conf. Field Program. Log. Appl. (FPL)*, Barcelona, Spain, Sep. 2019, pp. 397–403, doi: 10.1109/FPL.2019.00069.

[53] G. Fischer, J. Lusiardi, and J. W. von Gudenberg, "Abstract syntax trees-and their role in model driven software development," in *Proc. Int. Conf. Softw. Eng. Adv. (ICSEA)*, Aug. 2007, p. 38, doi: 10.1109/icsea.2007.12.

[54] Z. Wang and B. C. Schafer, "Machine leaming to set meta-heuristic specific parameters for high-level synthesis design space exploration," in *Proc. 57th ACM/IEEE Design Autom. Conf. (DAC)*, Jul. 2020, pp. 1–6, doi: 10.1109/DAC18072.2020.9218674.

[55] B. C. Schafer, "Probabilistic multiknob high-level synthesis design space exploration acceleration," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 3, pp. 394–406, Mar. 2016, doi: 10.1109/TCAD.2015.2472007.

[56] M. Zaheer, S. Kottur, S. Ravanbakhsh, B. Poczos, R. Salakhutdinov, and A. Smola, "Deep Sets," 2018, *arXiv:1703.061143*.

[57] Y.-K. Choi, P. Zhang, P. Li, and J. Cong, "HLScope+: Fast and accurate performance estimation for FPGA HLS," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Irvine, CA, USA, Nov. 2017, pp. 691–698.

[58] G. Zhong, A. Prakash, S. Wang, Y. Liang, T. Mitra, and S. Niar, "Design space exploration of FPGA-based accelerators with multi-level parallelism," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE),*, Mar. 2017, pp. 1141–1146, doi: 10.23919/DATE.2017.7927161.

[59] L. Breiman, "Stacked regressions," *Mach. Learn.*, vol. 24, no. 1, pp. 49–64, Jul. 1996, doi: 10.1007/bf00117832.

[60] Z. Lin, J. Zhao, S. Sinha, and W. Zhang, "HL-pow: A learning-based power modeling framework for high-level synthesis," in *Proc. 25th Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Beijing, China, Jan. 2020, pp. 574–580, doi: 10.1109/ASP-DAC47756.2020.9045442.

[61] W. Zuo, W. Kemmerer, J. B. Lim, L.-N. Pouchet, A. Ayupov, T. Kim, K. Han, and D. Chen, "A polyhedral-based SystemC modeling and generation framework for effective low-power design space exploration," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2015, pp. 357–364, doi: 10.1109/ICCAD.2015.7372592.

[62] A. Mahapatra and B. C. Schafer, "Machine-learning based simulated annealer method for high level synthesis design space exploration," in *Proc. Electron. Syst. Level Synth. Conf. (ESLsyn)*, San Francisco, CA, USA, May 2014, pp. 1–6, doi: 10.1109/ESLSYN.2014.6850383.

[63] J. R. Quinlan, "Induction of decision trees," *Mach. Learn.*, vol. 1, no. 1, pp. 81–106, Mar. 1986, doi: 10.1007/bf00116251.

[64] M. Yu, S. Huang, and D. Chen, "Chimera: A hybrid machine learning-driven multi-objective design space exploration tool for FPGA high-level synthesis," in *Intelligent Data Engineering and Automated Learning—IDEAL* (Lecture Notes in Computer Science), H. Yin, D. Camacho, P. Tino, R. Allmendinger, A. J. Tallón-Ballesteros, K. Tang, S.-B. Cho, P. Novais, and S. Nascimento, Eds., Cham, Switzerland: Springer, 2021, pp. 524–536, doi: 10.1007/978-3-030-91608-4_52.

[65] Q. Gautier, A. Althoff, C. L. Crutchfield, and R. Kastner, "Sherlock: A multi-objective design space exploration framework," *ACM Trans. Design Autom. Electron. Syst.*, vol. 27, no. 4, pp. 1–20, Jul. 2022, doi: 10.1145/3511472.

[66] P. Meng, A. Althoff, Q. Gautier, and R. Kastner, "Adaptive threshold non-Pareto elimination: Re-thinking machine learning for system level design space exploration on FPGAs," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2016, pp. 918–923, doi: 10.3850/9783981537079_0350.

[67] M. Zuluaga, A. Krause, and M. Püschel, "ϵ-PAL: An active learning approach to the multi-objective optimization problem," *J. Mach. Learn. Res.*, vol. 17, no. 1, pp. 3619–3650, Jan. 2016.

[68] K. Yu, J. Bi, and V. Tresp, "Active learning via transductive experimental design," in *Proc. 23rd Int. Conf. Mach. Learn. (ICML)*, Pittsburgh, PA, USA, 2006, pp. 1081–1088, doi: 10.1145/1143844.1143980.

[69] V. Nair, Z. Yu, T. Menzies, N. Siegmund, and S. Apel, "Finding faster configurations using FLASH," *IEEE Trans. Softw. Eng.*, vol. 46, no. 7, pp. 794–811, Jul. 2020, doi: 10.1109/TSE.2018.2870895.

[70] H. Chen and M. Shen, "A deep-reinforcement-learning-based scheduler for FPGA HLS," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design (ICCAD)*, Nov. 2019, pp. 1–8, doi: 10.1109/ICCAD45719.2019.8942126.

[71] R. A. Walker and S. Chaudhuri, "Introduction to the scheduling problem," *IEEE Design Test Comput.*, vol. 12, no. 2, pp. 60–69, Feb. 1995, doi: 10.1109/54.386007.
[72] J. Forrest and R. Lougee-Heimer, "CBC user guide," *INFORMS TutORials in Operations Research*, vol. 2005, pp. 257–277, Sep. 2005, doi: 10.1287/educ.1053.0020.
[73] E. Zennaro, L. Servadei, K. Devarajegowda, and W. Ecker, "A machine learning approach for area prediction of hardware designs from abstract specifications," in *Proc. 21st Euromicro Conf. Digit. Syst. Design (DSD)*, Aug. 2018, pp. 413–420, doi: 10.1109/DSD.2018.00076.
[74] B. Li, X. Zhang, H. You, Z. Qi, and Y. Zhang, "Machine learning based framework for fast resource estimation of RTL designs targeting FPGAs," *ACM Trans. Design Autom. Electron. Syst.*, vol. 28, no. 2, pp. 1–16, Dec. 2022, doi: 10.1145/3555047.
[75] D. Kim, J. Zhao, J. Bachrach, and K. Asanović, "Simmani: Runtime power modeling for arbitrary RTL with automatic signal selection," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*. New York, NY, USA: ACM, Oct. 2019, pp. 1050–1062, doi: 10.1145/3352460.3358322.
[76] G. Parthasarathy, A. Rushdi, P. Choudhary, S. Nanda, M. Evans, H. Gunasekara, and S. Rajakumar, "RTL regression test selection using machine learning," in *Proc. 27th Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2022, pp. 281–287, doi: 10.1109/ASP-DAC52403.2022.9712550.
[77] W. Hughes, S. Srinivasan, R. Suvarna, and M. Kulkarni, "Optimizing design verification using machine learning: Doing better than random," 2019, *arXiv:1909.13168*.
[78] D. Paletti, F. Peverelli, and D. Conficconi, "Online learning RTL synthesis for automated design space exploration," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, May 2022, pp. 69–76, doi: 10.1109/IPDPSW55747.2022.00021.
[79] A. Ghaffari, M. Asgharian, and Y. Savaria, "Statistical hardware design with multimodel active learning," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 43, no. 2, pp. 562–572, Feb. 2024, doi: 10.1109/TCAD.2023.3320984.
[80] Y.-D. Tsai, M. Liu, and H. Ren, "RTLFixer: Automatically fixing RTL syntax errors with large language models," 2023, *arXiv:2311.16543*.
[81] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "ReAct: Synergizing reasoning and acting in language models," 2022, *arXiv:2210.03629*.
[82] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-T. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive NLP tasks," 2020, *arXiv:2005.11401*.
[83] X. Yao, H. Li, T. Ho Chan, W. Xiao, M. Yuan, Y. Huang, L. Chen, and B. Yu, "HDLdebugger: Streamlining HDL debugging with large language models," 2024, *arXiv:2403.11671*.
[84] S. Thakur, B. Ahmad, Z. Fan, H. Pearce, B. Tan, R. Karri, B. Dolan-Gavitt, and S. Garg, "Benchmarking large language models for automated verilog RTL code generation," in *Proc. Design, Autom. Test Eur. Conf. Exhibit. (DATE)*, Apr. 2023, pp. 1–6, doi: 10.23919/date56975.2023.10137086.
[85] S. Thakur, B. Ahmad, H. Pearce, B. Tan, B. Dolan-Gavitt, R. Karri, and S. Garg, "VeriGen: A large language model for verilog code generation," *ACM Trans. Design Autom. Electron. Syst.*, vol. 29, no. 3, pp. 1–31, Apr. 2024, doi: 10.1145/3643681.
[86] J. Ye, X. Chen, N. Xu, C. Zu, Z. Shao, S. Liu, Y. Cui, Z. Zhou, C. Gong, Y. Shen, J. Zhou, S. Chen, T. Gui, Q. Zhang, and X. Huang, "A comprehensive capability analysis of GPT-3 and GPT-3.5 series models," 2023, *arXiv:2303.10420*.
[87] Y. Lu, S. Liu, Q. Zhang, and Z. Xie, "RTLLM: An open-source benchmark for design RTL generation with large language model," in *Proc. 29th Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2024, pp. 722–727, doi: 10.1109/asp-dac58780.2024.10473904.
[88] H. Hu, J. Hu, F. Zhang, B. Tian, and I. Bustany, "Machine-learning based delay prediction for FPGA technology mapping," in *Proc. 24th ACM/IEEE Workshop Syst. Level Interconnect Pathfinding*. San Diego, CA, USA: ACM, Nov. 2022, pp. 1–6, doi: 10.1145/3557988.3569713.
[89] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "LightGBM: A highly efficient gradient boosting decision tree," in *Proc. 31st Int. Conf. Neural Inf. Process. Syst.* Red Hook, NY, USA: Curran Associates, 2017, pp. 3149–3157.
[90] N. Kapre, H. Ng, K. Teo, and J. Naude, "InTime: A machine learning approach for efficient selection of FPGA CAD tool parameters," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*. New York, NY, USA: ACM, Feb. 2015, pp. 23–26, doi: 10.1145/2684746.2689081.
[91] C. Xu, G. Liu, R. Zhao, S. Yang, G. Luo, and Z. Zhang, "A parallel bandit-based approach for autotuning FPGA compilation," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*. Monterey, CA, USA: ACM, Feb. 2017, pp. 157–166, doi: 10.1145/3020078.3021747.
[92] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "OpenTuner: An extensible framework for program autotuning," in *Proc. 23rd Int. Conf. Parallel Archit. Compilation Techn. (PACT)*, Aug. 2014, pp. 303–315, doi: 10.1145/2628071.2628092.
[93] E. Ustun, S. Xiang, J. Gui, C. Yu, and Z. Zhang, "LAMDA: Learning-assisted multi-stage autotuning for FPGA design closure," in *Proc. IEEE 27th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Apr. 2019, pp. 74–77, doi: 10.1109/FCCM.2019.00020.
[94] Q. Yanghua, H. Ng, and N. Kapre, "Boosting convergence of timing closure using feature selection in a learning-driven approach," in *Proc. 26th Int. Conf. Field Program. Log. Appl. (FPL)*, Aug. 2016, pp. 1–9, doi: 10.1109/FPL.2016.7577302.
[95] Q. Yanghua, C. Adaikkala Raj, H. Ng, K. Teo, and N. Kapre, "Case for design-specific machine learning in timing closure of FPGA designs," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*. New York, NY, USA: ACM, Feb. 2016, pp. 169–172, doi: 10.1145/2847263.2847336.
[96] A. Grosnit, C. Malherbe, R. Tutunov, X. Wan, J. Wang, and H. B. Ammar, "BOiLS: Bayesian optimisation for logic synthesis," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2022, pp. 1193–1196, doi: 10.23919/DATE54114.2022.9774632.
[97] H. B. Moss, D. Beck, J. Gonzalez, D. S. Leslie, and P. Rayson, "BOSS: Bayesian optimization over string spaces," in *Proc. 34th Int. Conf. Neural Inf. Process. Syst.* Red Hook, NY, USA: Curran Associates, 2020, pp. 15476–15486.
[98] H. Lodhi, C. Saunders, J. Shawe-Taylor, N. Cristianini, and C. Watkins, "Text classification using string kernels," *J. Mach. Learn. Res.*, vol. 2, pp. 419–444, Feb. 2002.
[99] *ABC: A Simple System for Sequential Synthesis and Verification*. Accessed: Oct. 7, 2024. [Online]. Available: https://people.eecs.berkeley.edu/
[100] A. B. Chowdhury, B. Tan, R. Carey, T. Jain, R. Karri, and S. Garg, "Bullseye: Active few-shot learning guided logic synthesis," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 42, no. 8, pp. 2580–2590, Aug. 2023, doi: 10.1109/TCAD.2022.3226668.
[101] Z. Bodó, Z. Minier, and L. Csató, "Active learning with clustering," in *Proc. Act. Learn. Exp. Design Workshop Conjunct. AISTATS, JMLR Workshop Conf.*, 2011, pp. 127–139. Accessed: Oct. 7, 2024. [Online]. Available: https://proceedings.mlr.press/v16/bodo11a.html
[102] K. Zhu, M. Liu, H. Chen, Z. Zhao, and D. Z. Pan, "Exploring logic optimizations with reinforcement learning and graph convolutional network," in *Proc. ACM/IEEE 2nd Workshop Mach. Learn. CAD (MLCAD)*. Iceland: ACM, Nov. 2020, pp. 145–150, doi: 10.1145/3380446.3430622.
[103] A. Hosny, S. Hashemi, M. Shalan, and S. Reda, "DRiLLS: Deep reinforcement learning for logic synthesis," in *Proc. 25th Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2020, pp. 581–586, doi: 10.1109/ASP-DAC47756.2020.9045559.
[104] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *Proc. The 33rd Int. Conf. Mach. Learn.*, 2016, pp. 1928–1937. Accessed: Oct. 7, 2024. [Online]. Available: https://proceedings.mlr.press/v48/mniha16.html
[105] G. Zhou and J. H. Anderson, "Area-driven FPGA logic synthesis using reinforcement learning," in *Proc. 28th Asia South Pacific Design Autom. Conf. (ASP-DAC)*. Tokyo, Japan: ACM, Jan. 2023, pp. 159–165, doi: 10.1145/3566097.3567894.
[106] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017, *arXiv:1707.06347*.
[107] Y. Qian, X. Zhou, H. Zhou, and L. Wang, "Efficient reinforcement learning framework for automated logic synthesis exploration," in *Proc. Int. Conf. Field-Program. Technol. (ICFPT)*, Dec. 2022, pp. 1–6, doi: 10.1109/ICFPT56656.2022.9974330.
[108] C. Yu, "FlowTune: Practical multi-armed bandits in Boolean optimization," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Design (ICCAD)*. New York, NY, USA: ACM, Nov. 2020, pp. 1–9, doi: 10.1145/3400302.3415615.
[109] Y. V. Peruvemba, S. Rai, K. Ahuja, and A. Kumar, "RL-guided runtime-constrained heuristic exploration for logic synthesis," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Design (ICCAD)*, Nov. 2021, pp. 1–9, doi: 10.1109/ICCAD51958.2021.9643530.

[110] W. L. Neto, M. Austin, S. Temple, L. Amaru, X. Tang, and P.-E. Gaillardon, "LSOracle: A logic synthesis framework driven by artificial intelligence: Invited paper," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design (ICCAD)*, Nov. 2019, pp. 1–6, doi: 10.1109/ICCAD45719.2019.8942145.

[111] M. Karnaugh, "The map method for synthesis of combinational logic circuits," *Trans. Amer. Inst. Electr. Engineers, I, Commun. Electron.*, vol. 72, no. 5, pp. 593–599, Nov. 1953, doi: 10.1109/TCE.1953.6371932.

[112] T. Martin, G. Grewal, and S. Areibi, "A machine learning approach to predict timing delays during FPGA placement," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*. OR, USA: IEEE, Jun. 2021, pp. 124–127, doi: 10.1109/IPDPSW52791.2021.00026.

[113] T. Martin, D. Maarouf, Z. Abuowaimer, A. Alhyari, G. Grewal, and S. Areibi, "A flat timing-driven placement flow for modern FPGAs," in *Proc. 56th ACM/IEEE Design Autom. Conf. (DAC)*. New York, NY, USA: ACM, Jun. 2019, pp. 1–6, doi: 10.1145/3316781.3317743.

[114] T. Martin, C. Barnes, G. Grewal, and S. Areibi, "Integrating machine-learning probes into the VTR FPGA design flow," in *Proc. 35th SBC/SBMicro/IEEE/ACM Symp. Integr. Circuits Syst. Design (SBCCI)*, Porto Alegre, Brazil, Aug. 2022, pp. 1–6, doi: 10.1109/SBCCI55532.2022.9893251.

[115] C.-W. Pui, G. Chen, Y. Ma, E. F. Y. Young, and B. Yu, "Clock-aware ultrascale FPGA placement with machine learning routability prediction: (Invited paper)," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Irvine, CA, USA, Nov. 2017, pp. 929–936, doi: 10.1109/ICCAD.2017.8203880.

[116] C.-W. Pui, G. Chen, W.-K. Chow, K.-C. Lam, J. Kuang, P. Tu, H. Zhang, E. F. Y. Young, and B. Yu, "RippleFPGA: A routability-driven placement for large-scale heterogeneous FPGAs," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design (ICCAD)*, Nov. 2016, pp. 1–8, doi: 10.1145/2966986.2980084.

[117] D. Maarouf, A. Alhyari, Z. Abuowaimer, T. Martin, A. Gunter, G. Grewal, S. Areibi, and A. Vannelli, "Machine-learning based congestion estimation for modern FPGAs," in *Proc. 28th Int. Conf. Field Program. Log. Appl. (FPL)*, Dublin, Ireland, Aug. 2018, pp. 427–4277, doi: 10.1109/FPL.2018.00079.

[118] P. Kannan, S. Balachandran, and D. Bhatia, "On metrics for comparing interconnect estimation methods for FPGAs," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 12, no. 4, pp. 381–385, Apr. 2004, doi: 10.1109/TVLSI.2004.825865.

[119] D. Yeager, D. Chiu, and G. Lemieux, "Congestion estimation and localization in fpgas: A visual tool for interconnect prediction," in *Proc. Int. Workshop Syst. Level Interconnect Predict.* New York, NY, USA: ACM, Mar. 2007, pp. 33–40, doi: 10.1145/1231956.1231963.

[120] A. Al-Hyari, H. Szentimrey, A. Shamli, T. Martin, G. Gréwal, and S. Areibi, "A deep learning framework to predict routability for FPGA circuit placement," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 14, no. 3, pp. 1–28, Aug. 2021, doi: 10.1145/3465373.

[121] H. Szentimrey, A. Al-Hyari, J. Foxcroft, T. Martin, D. Noel, G. Grewal, and S. Areibi, "Machine learning for congestion management and routability prediction within FPGA placement," *ACM Trans. Design Autom. Electron. Syst.*, vol. 25, no. 5, pp. 1–25, Sep. 2020, doi: 10.1145/3373269.

[122] T. Martin, S. Areibi, and G. Grewal, "Effective machine-learning models for predicting routability during FPGA placement," in *Proc. ACM/IEEE 3rd Workshop Mach. Learn.CAD (MLCAD)*, Raleigh, NC, USA, Aug. 2021, pp. 1–6, doi: 10.1109/MLCAD52597.2021.9531243.

[123] N. Kapre, B. Chandrashekaran, H. Ng, and K. Teo, "Driving timing convergence of FPGA designs through machine learning and cloud computing," in *Proc. IEEE 23rd Annu. Int. Symp. Field-Program. Custom Comput. Mach.*, May 2015, pp. 119–126, doi: 10.1109/FCCM.2015.36.

[124] A. Al-hyari, Z. Abuowaimer, D. Maarouf, S. Areibi, and G. Grewal, "An effective FPGA placement flow selection framework using machine learning," in *Proc. 30th Int. Conf. Microelectron. (ICM)*, Sousse, Tunisia, Dec. 2018, pp. 164–167, doi: 10.1109/ICM.2018.8704066.

[125] W. Li, S. Dhar, and D. Z. Pan, "UTPlaceF: A routability-driven FPGA placer with physical and congestion aware packing," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 4, pp. 869–882, Apr. 2018, doi: 10.1109/TCAD.2017.2729349.

[126] K. E. Murray and V. Betz, "Adaptive FPGA placement optimization via reinforcement learning," in *Proc. ACM/IEEE 1st Workshop Mach. Learn. CAD (MLCAD)*, Sep. 2019, pp. 1–6, doi: 10.1109/MLCAD48534.2019.9142079.

[127] M. A. Elgamma, K. E. Murray, and V. Betz, "Learn to place: FPGA placement using reinforcement learning and directed moves," in *Proc. Int. Conf. Field-Program. Technol. (ICFPT)*, Dec. 2020, pp. 85–93, doi: 10.1109/ICFPT51103.2020.00021.

[128] M. A. Elgammal, K. E. Murray, and V. Betz, "RLPlace: Using reinforcement learning and smart perturbations to optimize FPGA placement," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 41, no. 8, pp. 2532–2545, Aug. 2022, doi: 10.1109/TCAD.2021.3109863.

[129] R. Chen, S. Lu, M. A. Elgammal, P. Chun, V. Betz, and D. Niu, "VPR-gym: A platform for exploring AI techniques in FPGA placement optimization," in *Proc. 33rd Int. Conf. Field-Program. Log. Appl. (FPL)*, Sep. 2023, pp. 72–78, doi: 10.1109/fpl60245.2023.00018.

[130] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI gym," 2016, *arXiv:1606.01540*.

[131] U. Siddiqi, T. Martin, S. Van Den Eijnden, A. Shamli, G. Grewal, S. Sait, and S. Areibi, "Faster FPGA routing by forecasting and pre-loading congestion information," in *Proc. ACM/IEEE 4th Workshop Mach. Learn. CAD (MLCAD)*, Sep. 2022, pp. 15–20, doi: 10.1109/MLCAD55463.2022.9900091.

[132] A. D. Gunter and S. Wilton, "Reformulating the FPGA routability prediction problem with machine learning," in *Proc. IEEE 31st Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, May 2023, pp. 230–232, doi: 10.1109/FCCM57271.2023.00057.

[133] A. D. Gunter and S. J. E. Wilton, "A machine learning approach for predicting the difficulty of FPGA routing problems," in *Proc. IEEE 31st Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, May 2023, pp. 63–74, doi: 10.1109/FCCM57271.2023.00016.

[134] B. Ghavami, M. Ibrahimipour, Z. Fang, and L. Shannon, "MAPLE: A machine learning based aging-aware FPGA architecture exploration framework," in *Proc. 31st Int. Conf. Field-Program. Log. Appl. (FPL)*, Aug. 2021, pp. 369–373, doi: 10.1109/FPL53798.2021.00070.

[135] S. Zheng, J. Qian, H. Zhou, and L. Wang, "GRAEBO: FPGA general routing architecture exploration via Bayesian optimization," in *Proc. 32nd Int. Conf. Field-Program. Log. Appl. (FPL)*, Aug. 2022, pp. 282–286, doi: 10.1109/FPL57034.2022.00050.

[136] J. Qian, Y. Shen, K. Shi, H. Zhou, and L. Wang, "General routing architecture modelling and exploration for modern FPGAs," in *Proc. Int. Conf. Field-Program. Technol. (ICFPT)*, Dec. 2021, pp. 1–9, doi: 10.1109/ICFPT52863.2021.9609935.

[137] U. Farooq, N. U. Hasan, I. Baig, and M. Zghaibeh, "Efficient FPGA routing using reinforcement learning," Valencia, Spain, May 2021, pp. 106–111, doi: 10.1109/ICICS52457.2021.9464626.

[138] J. Luu, I. Kuon, P. Jamieson, T. Campbell, A. Ye, W. M. Fang, K. Kent, and J. Rose, "VPR 5.0: FPGA CAD and architecture exploration tools with single-driver routing, heterogeneity and process scaling," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 4, no. 4, pp. 1–23, Dec. 2011, doi: 10.1145/2068716.2068718.

[139] Z. Seifoori, H. Asadi, and M. Stojilovic, "A machine learning approach for power gating the FPGA routing network," in *Proc. Int. Conf. Field-Program. Technol. (ICFPT)*, Dec. 2019, pp. 10–18, doi: 10.1109/ICFPT47387.2019.00010.

[140] A. A. M. Bsoul and S. J. E. Wilton, "An FPGA with power-gated switch blocks," in *Proc. Int. Conf. Field-Program. Technol.*, Dec. 2012, pp. 87–94, doi: 10.1109/FPT.2012.6412117.

[141] C. H. Hoo, Y. Ha, and A. Kumar, "A directional coarse-grained power gated FPGA switch box and power gating aware routing algorithm," in *Proc. 23rd Int. Conf. Field Program. Log. Appl.*, Sep. 2013, pp. 1–4, doi: 10.1109/FPL.2013.6645548.

[142] Z. Seifoori, B. Khaleghi, and H. Asadi, "A power gating switch box architecture in routing network of SRAM-based FPGAs in dark silicon era," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE),*, Mar. 2017, pp. 1342–1347, doi: 10.23919/DATE.2017.7927201.

[143] *HDLBits*. Accessed: Oct. 7, 2024. [Online]. Available: https://hdlbits.01xz.net/wiki/Main_Page

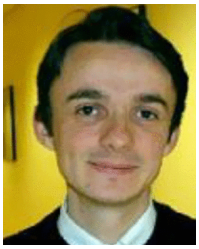
[144] N. Pinckney, C. Batten, M. Liu, H. Ren, and B. Khailany, "Revisiting VerilogEval: Newer LLMs, in-context learning, and specification-to-RTL tasks," 2024, *arXiv:2408.11053*.

[145] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii, "CHStone: A benchmark program suite for practical C-based high-level synthesis," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2008, pp. 1192–1195, doi: 10.1109/ISCAS.2008.4541637.

[146] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks, "MachSuite: Benchmarks for accelerator design and customized architectures," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Oct. 2014, pp. 110–119, doi: 10.1109/IISWC.2014.6983050.

[147] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communications systems," in *Proc. 30th Annu. Int. Symp. Microarchitecture*, Dec. 1997, pp. 330–335, doi: 10.1109/micro.1997.645830.
[148] *PolyBench/C—Homepage of Louis-Noël Pouchet*. Accessed: Oct. 7, 2024. [Online]. Available: https://web.cs.ucla.edu/
[149] Y. Zhou, U. Gupta, S. Dai, R. Zhao, N. Srivastava, H. Jin, J. Featherston, Y.-H. Lai, G. Liu, G. A. Velasquez, W. Wang, and Z. Zhang, "Rosetta: A realistic high-level synthesis benchmark suite for software programmable FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*. New York, NY, USA: ACM, Feb. 2018, pp. 269–278, doi: 10.1145/3174243.3174255.
[150] B. C. Schafer and A. Mahapatra, "S2CBench: Synthesizable SystemC benchmark suite for high-level synthesis," *IEEE Embedded Syst. Lett.*, vol. 6, no. 3, pp. 53–56, Sep. 2014, doi: 10.1109/LES.2014.2320556.
[151] Q. Gautier, A. Althoff, P. Meng, and R. Kastner, "Spector: An OpenCL FPGA benchmark suite," in *Proc. Int. Conf. Field-Program. Technol. (FPT)*, Dec. 2016, pp. 141–148, doi: 10.1109/FPT.2016.7929519.
[152] S. Yang, *Logic Synthesis and Optimization Benchmarks User Guide: Version 3.0*. Princeton, NJ, USA: Citeseer, 1991.
[153] D. Bryan. *The ISCAS '85 Benchmark Circuits and Netlist Format*. Accessed: Oct. 7, 2024. [Online]. Available: https://s2.smu.edu/
[154] M. C. Hansen, H. Yalcin, and J. P. Hayes, "Unveiling the ISCAS-85 benchmarks: A case study in reverse engineering," *IEEE Design Test Comput.*, vol. 16, no. 3, pp. 72–80, Jul. 1999, doi: 10.1109/54.785838.
[155] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," in *Proc. IEEE Int. Symp. Circuits Syst.*, May 1989, pp. 1929–1934, doi: 10.1109/ISCAS.1989.100747.
[156] *IWLS 2005 Benchmarks*. Accessed: Oct. 7, 2024. [Online]. Available: https://iwls.org/iwls2005/benchmarks.html
[157] L. Amarú, P.-E. Gaillardon, and G. D. Micheli, "The EPFL combinational benchmark suite," in *Proc. 24th Int. Workshop Log. Synthesis (IWLS)*, 2015, pp. 1–5. Accessed: Oct. 7, 2024. [Online]. Available: http://infoscience.epfl.ch/record/207551
[158] G. Tziantzioulis, T.-J. Chang, J. Balkind, J. Tu, F. Gao, and D. Wentzlaff, "OPDB: A scalable and modular design benchmark," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 41, no. 6, pp. 1878–1887, Jun. 2022, doi: 10.1109/TCAD.2021.3096794.
[159] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed, K. B. Kent, J. Anderson, J. Rose, and V. Betz, "VTR 7.0: Next generation architecture and CAD system for FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 7, no. 2, pp. 1–30, Jul. 2014, doi: 10.1145/2617593.
[160] K. E. Murray, S. Whitty, S. Liu, J. Luu, and V. Betz, "Titan: Enabling large and complex benchmarks in academic CAD," in *Proc. 23rd Int. Conf. Field Program. Log. Appl.*, Sep. 2013, pp. 1–8, doi: 10.1109/FPL.2013.6645503.
[161] A. Arora, A. Boutros, D. Rauch, A. Rajen, A. Borda, S. A. Damghani, S. Mehta, S. Kate, P. Patel, K. B. Kent, V. Betz, and L. K. John, "Koios: A deep learning benchmark suite for FPGA architecture and CAD research," in *Proc. 31st Int. Conf. Field-Program. Log. Appl. (FPL)*, Aug. 2021, pp. 355–362, doi: 10.1109/FPL53798.2021.00068.
[162] A. Arora, A. Boutros, S. A. Damghani, K. Mathur, V. Mohanty, T. Anand, M. A. Elgammal, K. B. Kent, V. Betz, and L. K. John, "Koios 2.0: Open-source deep learning benchmarks for FPGA architecture and CAD research," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 42, no. 11, pp. 3895–3909, Nov. 2023, doi: 10.1109/TCAD.2023.3272582.
[163] *Benchmarks-Verilog-to-Routing 8.1.0-dev Documentation*. Accessed: Oct. 7, 2024. [Online]. Available: https://docs.verilogtorouting.org/en/latest/vtr/benchmarks/#symbiflow-benchmarks
[164] *Guelph FPGA CAD Group—Benchmarks*. Accessed: Oct. 7, 2024. [Online]. Available: https://fpga.socs.uoguelph.ca/benchmarks
[165] *ISPD 2016 Contest: FPGA Placement*. Accessed: Oct. 7, 2024. [Online]. Available: https://www.ispd.cc/contests/16/ispd2016_contest.html
[166] *ISPD 2017 Contest: Clock-Aware FPGA Placement*. Accessed: Oct. 7, 2024. [Online]. Available: https://www.ispd.cc/contests/17/
[167] J. Singh, "Computational complexity and analysis of supervised machine learning algorithms," in *Next Generation of Internet of Things*, R. Kumar, P. K. Pattnaik, and J. M. R. S. Tavares, Eds., Singapore: Springer, 2023, pp. 195–206, doi: 10.1007/978-981-19-1412-6_16.
[168] A. Creswell, T. White, V. Dumoulin, K. Arulkumaran, B. Sengupta, and A. A. Bharath, "Generative adversarial networks: An overview," *IEEE Signal Process. Mag.*, vol. 35, no. 1, pp. 53–65, Jan. 2018, doi: 10.1109/MSP.2017.2765202.
[169] W. Xin Zhao et al., "A survey of large language models," 2023, *arXiv:2303.18223*.

ARMANDO BISCONTINI received the B.Sc. and M.Sc. degrees in electrical engineering from the Politecnico di Milano, Milan, Italy. He is currently pursuing the Ph.D. degree with the Department of Electrical and Electronic Engineering, University College Cork, Cork, Ireland. He is also a Staff Engineer with QT Technologies Ireland Ltd., Cork. Previously, he was a Research Scholar with The University of Utah, Salt Lake City, USA, and the École Polytechnique Fédérale de Lausanne, Switzerland. His research interests include machine learning applications to electronic design automation and optimization of silicon design methodology, the physical design implementation of high-speed telecommunication modules for system-on-chip, and the study of emerging resistive memories.



E. POPOVICI (Senior Member, IEEE) received the Dipl. (Ing.) degree in computer engineering from the Politehnica University of Timisoara, Romania, in 1997, and the Ph.D. degree in microelectronic engineering from the National University of Ireland, Cork, Ireland, in 2002. He is currently a Senior Lecturer with the Department of Electrical and Electronic Engineering, National University of Ireland. His research interests include electronic design automation, embedded system design for low power, reliable, secure computing, and communications. His teams received more than 50 awards and distinctions at national and international level for co-authored articles or international competitions in these fields.

A. TEMKO (Senior Member, IEEE) received the B.E. degree in computer science from Oles Honchar Dnipro National University, Dnipro, Ukraine, in 2002, and the Ph.D. degree in telecommunication from the Universitat Politècnica de Catalunya, Barcelona, Spain, in 2008. From 2008 to 2018, he was with the Irish Centre for Fetal and Neonatal Translational Research, University College Cork, Ireland. He developed and patented a novel cutting-edge neonatal seizure detection system, which has completed a European multi-centre clinical trial toward its regulatory approval and clinical adoption. In 2017, he was named a Winner of the Kaggle challenge, for his work in predicting seizures in the human brain through long-term EEG recordings, organized by the National Institute of Health, American Epilepsy Society, and Melbourne University. Since 2018, he has been with QT Technologies Ireland Ltd., Cork, Ireland. His research interests include machine learning for signal processing, decision support tools, and ML-based electronic design automation.