Fast and Accurate Estimation of Quality of Results in High-Level Synthesis with Machine Learning

Steve Dai[1], Yuan Zhou[1], Hang Zhang[1], Ecenur Ustun[1], Evangeline F.Y. Young[2], Zhiru Zhang[1]

[1]*School of Electrical and Computer Engineering, Cornell University, Ithaca, NY, USA*
[2]*Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shatin, NT, Hong Kong*
{hd273,yz882,hz459,eu49}@cornell.edu, fyyoung@cse.cuhk.edu.hk, zhiruz@cornell.edu

Abstract—**While high-level synthesis (HLS) offers sophisticated techniques to optimize designs for area and performance, HLS-estimated resource usage and timing often deviate significantly from actual quality of results (QoR) achieved by FPGA-targeted designs. Inaccurate HLS estimates prevent designers from performing meaningful design space exploration without resorting to the time-consuming downstream implementation process. To address this challenge, we first build a large collection of C-to-FPGA results from a diverse set of realistic HLS applications and identify relevant features from HLS reports for estimating post-implementation metrics. We then leverage these features and data to train and compare a number of promising machine learning models to effectively and efficiently bridge the accuracy gap. Experiments demonstrate that our proposed approach is able to dramatically reduce the estimation errors for different families of FPGA devices. By extracting domain-specific insights from our experiments, we explore the implications of our models and predictive influence of various features for enabling fast and accurate QoR estimation in HLS. We have released our dataset to springboard future efforts in this area.**

I. Introduction

High-level synthesis (HLS) provides the capability to automatically convert untimed software descriptions into optimized cycle-accurate hardware models. As a result, it has been recognized as an effective approach for improving the productivity of hardware design. With HLS, designers no longer need to constantly wrestle with low-level hardware description language (HDL) details, and can instead focus on making the best algorithmic and microarchitectural tradeoff, all from a single software design source.

While HLS-generated register-transfer-level (RTL) models are in general not human-readable, HLS tools provide a set of reports to quantitatively convey the expected performance, timing, resource usage, and composition of the synthesized RTL design. These reports represent the crucial, and oftentimes, sole evidence based on which the design is iteratively modified to achieve more desirable results. Despite their importance, many reported values are highly inaccurate. In particular, final resource usage and timing depend on multiple downstream implementation stages (e.g., logic synthesis, place and route) beyond HLS and are therefore difficult to estimate even by state-of-the-art HLS tools.

Analysis of our data from a large set of designs reveals that a commercial HLS tool targeting FPGAs incurs a relative error (defined in Section V) of 125% in estimating the number of look-up tables (LUTs). Similarly, the error for flip-flop (FF) estimation stands at 98%. Such inaccurate estimates on quality of results (QoR) prevent designers and even the tool itself from applying the appropriate set of optimizations, resulting in designs with the wrong tradeoff. To obtain more accurate QoR estimates, designers spend an enormous amount of time iterating the downstream implementation flow for each design point. However, doing so is impractical and nullifies the productivity advantage for which HLS is known. The magnitude of the error indicates the lack of sufficiently accurate models for these estimation tasks.

To address this challenge, we propose to leverage 87 features that can be readily extracted from the HLS reports to accurately predict post-implementation results without actually running the implementation flow. Using these features, we train a number of promising machine learning models, including linear regression, artificial neural network, and gradient tree boosting, to achieve high accuracy for the estimation tasks. We select models whose features can be directly interpreted to gain meaningful insights on top of accurate estimates. Our major contributions include:

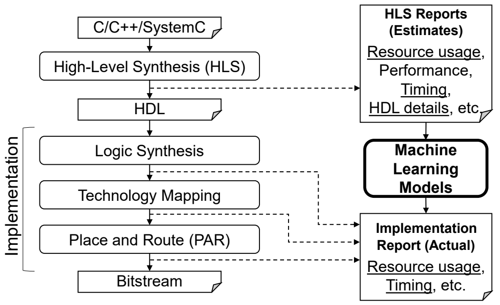

Figure 1: **FPGA tool flow with HLS and proposed machine learning models (in bold) –** The models use underlined metrics in the HLS reports to predict underlined post-implementation metrics.

1) We train a set of machine learning models that reduce the errors of HLS estimations by up to 138% using features extracted from HLS reports.
2) We comparatively study our trained models and employ domain-specific knowledge to explore model implications and predictive influence of various features.
3) Our dataset generated from realistic HLS benchmarks is publicly available on the authors' website to enable further modeling and knowledge discovery efforts in the community.

Some existing work extracts the number of required functional units from dataflow graphs and applies analytical models to approximate resource usage [1], [2], [3]. While these techniques enable fast early-stage design studies prior to completing the HLS process, they are not designed to grasp the intricate effects of the implementation process. As a result, their estimations are only competitive against the crude HLS estimates we aim to tackle. In contrast, we leverage a data-driven approach to holistically model the combined effects of implementation to enable fast estimates that are competitive even against final implementation results.

The rest of the paper is organized as follows: Section II provides motivation for the problem; Section III describes our dataset and data processing; Section IV illustrates machine learning models used to address the problem; Section V presents experimental results and insights; Section VI reviews related work, followed by a summary of our ongoing work in Section VII.

II. Motivation

As shown in Figure 1, an HLS-based design flow starts with a high-level software program, typically in C, C++, or SystemC, that is automatically synthesized into HDL models in Verilog or VHDL. HLS reports are generated alongside the models to indicate the expected performance, estimated resource usage and timing, as well as certain HDL details of the design. For an FPGA flow, the HDL models then run through logic synthesis, technology mapping, and place and route to generate a bitstream for the target FPGA. This HDL-to-bitstream process is collectively known as implementation as indicated in Figure 1. Implementation reports are generated to detail the actual resource usage and timing of the design on-chip.

Accurate estimations of post-implementation results at the HLS stage is difficult because the final implemented design reflects the cumulative effects of many non-trivial transformations through the series of implementation stages shown in Figure 1. Moreover, final resource usage and timing depend on constraints imposed by the target FPGA device, specifically the number, structure, and interconnection of device resources such as LUTs, FFs, DSP blocks (i.e., hardened multipliers), and block RAMs (BRAMs). To enable fast resource and timing estimation, HLS tools pre-characterize different functional units ahead of time and sum up the contributions of instantiated functional units during the synthesis of each design. However, such additive estimation approach fails to correctly capture the effects of post-HLS optimizations across functional units and neglects to consider limitations imposed by finite compute and routing resources on-chip.

As machine learning gains traction in design automation, we believe that it provides the means to holistically and precisely capture the multitude of factors affecting estimation accuracy. With the appropriate dataset on-hand, we can model the intricacies of the implementation process as a practical solution to the HLS estimation problem. As shown in Figure 1, we propose to apply machine learning (regression and classification) models to predict actual resource usage and timing from estimates in the HLS reports.

III. Data Processing and Analysis

To enable machine learning for HLS estimation, we build a dataset of HLS and implementation results consisting of over 1300 samples across 65 individual designs. To ensure quality, we leverage designs from well-known HLS benchmark suites, including CHStone [4], Machsuite [5], and S2CBench [6]. To increase diversity, we complement these benchmark suites with Rosetta benchamarks [7], which include machine learning and real-time video processing applications. These additional designs differentiate from conventional benchmarks because they represent large fully developed applications instead of small kernel programs. They are implemented under realistic design constraints and reflect the latest application trends. We run each design through the complete C-to-bitstream flow for various clock periods (1, 2, 3, 5, 10ns) targeting different FPGA devices (Xilinx Zynq, Artix7, Kintex7, and Virtex7). Table I summarizes the overall characteristics of the designs. The dataset can be further augmented by synthesizing the designs with different combinations of HLS optimization directives.

Table I: **Summary of designs –** Post-implementation resource usages and worst negative slack (WNS) are shown. A negative WNS indicates that timing is not met.

	#LUT	**#FF**	**#DSP**	**#BRAM**	**WNS (ns)**
Max	63645	115452	795	350	8.4
Min	34	0	0	0	-39.7
Mean	5791	7395	25	19	-0.2

To construct our dataset, we first identify features of the HLS designs useful for predicting implementation results. For this purpose, we limit ourselves to features that can be readily extracted from the HLS reports. As such, feature extraction incurs trivial computation overhead, and our approach is generally applicable to different HLS tools. Similarly, we extract implementation results, known as the targets in our machine learning problem, from the implementation reports. After extraction, our dataset contains features and targets for each design sample and can be used to develop estimation models that map from features to targets.

1) Feature Extraction: While we can leverage domain knowledge to hypothesize the relevance of different features to our estimation tasks, it is impossible to ascertain the predictive abilities of and relationship among the different features in advance. As a result, we extract as many relevant features as possible first and apply feature selection techniques (to be discussed in Sections III-2 and III-3) later to systematically remove any unimportant features. Feature extraction results in a total 234 features, all of which represent estimates from HLS reports.

2) Removing Redundant Features: Our effort in building a comprehensive feature set may result in features that are statistically correlated and can be predicted with sufficient accuracy by other features. While this phenomenon of collinearity does not typically degrade the accuracy of estimation models, it nevertheless limits conclusions one can make about the predictive influence of a particular feature because the marginal contribution of the feature depends on which other correlated features are also present in the model. To overcome the effect of collinearity, we compute the Pearson's correlation coefficient for each pair of features on our dataset and select only one feature from each group of correlated features to be included for subsequent modeling.

3) Eliminating Irrelevant Features: Features that exert little influence on the targets should be eliminated to reduce the dimensionality of the data. Having fewer features leads to simpler models that require shorter training time, reduce the chance of overfitting, and are easier to interpret. To eliminate irrelevant features, we fit our data to a linear model with L1 regularization. As described later in Section IV-1, an L1-regularized linear model induces a sparse estimator that zeros out the coefficients of unimportant features and thus selects the important features (with non-zero coefficients). We apply L1 feature selection in conjunction with the correlation technique in Section III-2 to reduce the number of features from 234 to 87. Table II describes categories of our selected features. For dimensionality reduction, we choose L1 feature selection over matrix factorization approaches such as principal component analysis to preserve the original components of the feature set so that we can directly interpret the importance of each feature.

Table II: **Descriptions of categories of selected features**

Category	**Brief Description**
Resource #	Usage & available number of each resource type.
Clock periods	Target clock period; achieved clock period & its uncertainty.
Logic ops	Bitwidth/resource statistics of logic operations (e.g., or, shift).
Arithmetic ops	Bitwidth/resource statistics of arithmetic operations (e.g., add, mul).
Memory	Number of memory words/banks/bits; resource usage for memory.
Multiplexer	Resource usage for multiplexers; multiplexer input size/bitwidth.

IV. Estimation Models

We train regression models to estimate post-implementation resource usages for LUT, FF, DSP, and BRAM, as well as classification models to predict whether the target clock period is met for the implemented design. Because the same concepts carry over from regression to classification, we will focus on describing models in the context of regression and resource estimations.

In general, regression is a supervised machine learning technique that infers a function from features to targets in the training set. For our study, we have a set of n training samples $\{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^{n}$, where $\mathbf{x}_i = [x_i^1, x_i^2, ..., x_i^p]^\top \in \mathbb{R}^p$ is the input vector of feature values for the ith sample, and $\mathbf{y}_i = [y_i^1, y_i^2, ..., y_i^q]^\top \in \mathbb{R}^q$ is the corresponding vector of target values. Here p denotes the number of input features (e.g., LUT count and clock period estimated by HLS), and q denotes the number of output targets (i.e., actual LUT, FF, DSP, and BRAM counts post-implementation). We further define $\mathbf{X} = [\mathbf{x_1}, ..., \mathbf{x_n}]^\top$ to denote feature values for all samples and $\mathbf{y^k} = [y_1^k, y_2^k, ..., y_n^k]^\top$ to denote values of target k for all samples.

Each learning task corresponds to one target estimation. For single-task learning, we train a separate model f_k for each target k, resulting in a set of mapping functions $\{f_k : \mathbb{R}^p \to \mathbb{R}\}_{k=1}^{q}$. For multi-task learning, we train a single model $f : \mathbb{R}^p \to \mathbb{R}^q$ that predicts all targets at the same time. Multi-task learning is a type of inductive transfer learning which improves training efficiency and prediction accuracy by exploiting information from the training of related tasks [8]. While we experiment with both single-task and multi-task models when applicable, we will discuss only single-task versions of our selected machine learning models in this section.

Table III: **Resource estimation errors –** RAE incurred in the estimation of the usage of each resource type is shown for four different devices (ordered by increasing size). HLS Estimate: Models built-in to the HLS tool. Lasso, ANN, and XGB: Models described in Sections IV-1, IV-2 and IV-3, respectively. 2-ANN and 2-Lasso: Corresponding multi-task models trained simultaneously for LUT and FF tasks. 4-ANN and 4-Lasso: Corresponding multi-task models trained simultaneously for all four tasks. n/a: Not applicable.

Device		**XC7Z020 (Zynq-7000)**				**XC7A100T (Artix-7)**				**XC7K160T (Kintex-7)**				**XC7V585T (Virtex-7)**			
Resource		**LUT**	**FF**	**DSP**	**BRAM**	**LUT**	**FF**	**DSP**	**BRAM**	**LUT**	**FF**	**DSP**	**BRAM**	**LUT**	**FF**	**DSP**	**BRAM**
HLS Estimate		141.2%	84.3%	22.1%	14.9%	99.6%	89.1%	19.8%	9.2%	112.5%	73.5%	15.8%	12.2%	107.0%	95.4%	14.3%	6.9%
Single-Task	**XGB**	3.7%	2.5%	0.2%	0.1%	2.6%	3.3%	0.4%	0.2%	3.7%	3.8%	0.9%	0.3%	6.2%	5.6%	10.0%	0.2%
	ANN	6.7%	5.6%	2.7%	2.3%	3.7%	3.9%	4.0%	3.4%	5.3%	4.7%	3.8%	3.9%	7.9%	8.5%	9.5%	4.6%
	Lasso	16.2%	19.6%	9.7%	7.1%	9.2%	10.2%	13.7%	6.7%	11.4%	11.0%	15.4%	10.5%	15.7%	18.7%	18.0%	9.0%
Multi-Task	**2-ANN**	8.0%	4.8%	n/a	n/a	3.5%	3.6%	n/a	n/a	7.4%	4.3%	n/a	n/a	7.0%	7.0%	n/a	n/a
	4-ANN	7.8%	5.1%	3.6%	3.3%	4.2%	4.0%	5.9%	4.6%	7.3%	4.7%	6.0%	5.8%	8.3%	8.4%	11.6%	4.3%
	2-Lasso	15.9%	19.5%	n/a	n/a	8.1%	9.6%	n/a	n/a	12.1%	11.3%	n/a	n/a	15.3%	17.6%	n/a	n/a
	4-Lasso	16.0%	19.5%	9.8%	6.8%	7.9%	9.7%	13.8%	8.0%	11.7%	11.5%	15.7%	12.2%	15.6%	18.4%	17.1%	10.3%

1) Linear Model: We start with the classic linear regression model, $\widehat{y_i} = \mathbf{x_i}^\top \mathbf{w}$, which models the target $\widehat{y_i}$ as a linear combination of features $\mathbf{x_i}$. Linear regression fits this model onto the training data to determine the $\mathbf{w}$ such that a loss function is minimized, where $\mathbf{w}$ represents the vector of coefficients for the learned model. In our case, we use the Lasso linear model with a loss function of $\left\|\mathbf{X}\mathbf{w} - \mathbf{y^k}\right\|_2^2 + \gamma \left\|\mathbf{w}\right\|_1$ to train a linear model that minimizes the least-square penalty on the training data with L1 regularization. By tuning the hyperparameter γ, the L1 regularization term $\gamma \left\|\mathbf{w}\right\|_1$ allows us to induce various degree of sparsity into $\mathbf{w}$ and in turn regulate the complexity of the model.

2) Neural Network: Unlike linear models, artificial neural network (ANN) is able to capture non-linearity in the data [9]. An ANN consists of an input layer, followed by a series of hidden layers, and an output layer. Each hidden layer contains a set of neurons, each of which transforms values from the previous layer using a linear model followed by a non-linear activation function. While deep neural networks can represent complicated non-linear functions, a large amount of training data is needed for the model to converge. For our estimation problem, the number of features is relatively small, and the amount of training data is limited. Therefore, we use ANNs with only a few fully-connected hidden layers. We choose to include ANN to validate our hypothesis that the mapping from features to targets is non-linear. Compared to linear models, ANN requires tuning more hyperparameters (e.g., number of layers, neurons per layer) and results in non-convex loss functions that require more effort in training.

3) Gradient Tree Boosting: Based on building a "strong" regression tree by combining a series of "weak" ones, tree boosting represents another promising non-linear technique [10]. It models the target as the sum of regression trees, each of which maps the features to a score for the target. Target estimation is determined by accumulating scores across all trees. Gradient tree boosting implements gradient descent that optimizes the loss over the space of regression trees by repeatedly selecting the tree that points in the negative gradient direction. For our model, we apply XGBoost [11], a recent gradient tree boosting algorithm that enhances scalability using sparsity-aware approximate split finding. XGBoost has demonstrated accuracy competitive to neural networks while attaining better efficiency in both training and inference. However, there is no existing multi-task models available for XGBoost.

V. Experiments

We implement and train the models described in Section IV in Python leveraging the scikit-learn [12] and XGBoost [11] libraries. All designs in the dataset are synthesized and implemented with Xilinx Vivado 2017.1 targeting Zynq-7000, Artix-7, Kintex-7, and Virtex-7. Experiments are performed on an Intel Xeon processor running at 2.5GHz. Regardless of the design, all models are able to complete the estimation tasks within milliseconds, compared to minutes or hours that each implementation stage typically incurs.

For regression, we compute the relative absolute error (RAE), defined as $\epsilon = |\hat{\mathbf{y}} - \mathbf{y}|/|\mathbf{y} - \bar{\mathbf{y}}|$, to evaluate and compare the accuracy of different models. $\hat{\mathbf{y}}$ is a vector of values predicted by the model for a particular target, and $\mathbf{y}$ is a vector of actual ground truth values in the testing set for that target. $\bar{\mathbf{y}}$ denotes the mean value of $\mathbf{y}$. For classification, we compute the percentage of incorrectly classified samples out of the total number of samples. We randomly select 20% of our data as the testing set and perform random permutation cross-validation over 10 iterations on the remaining training/validation set. In each iteration, we randomly select 75% of the training/validation set for training and 25% for validation. While the validation set is used for parameter tuning to locate better models, the testing set remains isolated until the end only to evaluate the accuracy of the finalized models. This ensures that our models are not tuned for the testing set. We employ grid search to find the best set of hyperparameters (e.g., γ in linear model, number of hidden layers for ANN) for each model.

1) Resource Estimation: Table III lists the estimation errors incurred by the HLS tool in comparison to those of both single-task and multi-task versions of our regression models. Based on this table, we observe that the HLS tool suffers from severe estimation errors for LUT and FF while attaining reasonable accuracy for DSP and BRAM. Diving further into our dataset, we observe HLS estimated LUT counts that are on average 4.5x and up to 40x of actual LUT counts even for designs that utilize no BRAMs. This suggests that the disparity stems from the ineffectiveness of HLS additive estimation models (Section II) in capturing the effects of logic synthesis and LUT/FF mapping and is not the result of failure in predicting whether BRAMs will be inferred.

Table III demonstrates that the non-linear models (XGBoost and ANN) are able to achieve sizable reduction in estimation error for all resources on all four devices. While the linear model (Lasso) can also reduce the errors significantly for LUT and FF, it experiences difficulty with DSP and BRAM and are worse than HLS estimates in a few cases. The observation that non-linear model performs significant better than linear model validates our hypothesis that the data are non-linear. DSP and BRAM are generally easier to estimate than LUT and FF for both the HLS tool and our non-linear models because LUT and FF bound operations experience more complicated transformations than DSP-bound operations and memories, making them more susceptible to the simplicity of built-in HLS estimation models. In general, XGBoost stands out as the most competitive model with less than 5% error for LUT and FF and less than 1% error for DSP and BRAM in a majority of single-task cases. It is especially competitive for estimating DSP and BRAM usages because these usages tend to be stepwise functions of features such as operator bitwidths and memory sizes. Splits learned by regression trees correspond precisely to stepwise functions.

Our results indicate that there is no apparent benefit for multi-task models over single-task models, as multi-task models achieve lower errors than their single-task counterparts for only limited cases in Table III. The two-task ANNs (estimating LUT and

Table IV: **Timing classification errors –** Error rate for corresponding models and devices following the same notations as Table III.

Device	XC7Z020	XC7A100T	XC7K160T	XC7V585T
HLS Estimate	21.3%	19.8%	19.1%	16.1%
XGB	1.6%	4.6%	10.5%	8.8%
ANN	3.2%	6.2%	10.5%	13.2%

Table V: **Important categories of features for each estimation task in XGBoost –** Ranked by combined importance of features in each category. #LUT, #FF, #DSP, and #BRAM: HLS estimated resource counts. Mux: Multiplexer-related. Est_CP and Target_CP: Estimated and target clock periods. Logic_Ops: Logic operations.

Task	LUT	FF	DSP	BRAM	Timing
Important Feature Categories	#FF	#FF	#DSP	#BRAM	Logic_Ops
	#LUT	#LUT	#FF	Mux	Target_CP
	Mux	Mux	#LUT	#FF	Est_CP
	Est_CP	Est_CP	#BRAM	#LUT	#FF
	#BRAM	#BRAM	Est_CP	Est_CP	#LUT

FF simultaneously) generally perform better than four-task ANNs (estimating all resources simultaneously) because the LUT and FF tasks are more correlated to each other than to either the DSP or BRAM task. However, the correlation between LUT and FF tasks is still not sufficiently strong for meaningful inductive knowledge transfer. In addition, training all four tasks simultaneously increases errors in most cases for DSP and BRAM, revealing evidence of negative knowledge transfer because the estimation tasks for DSP and BRAM are not closely related to those for LUT and FF.

2) Timing Classification: Table IV lists the errors on classifying whether a design meets the target clock period. While the built-in HLS estimates are reasonably good in the first place, all of our implemented models are able to achieve further reduction in error. XGBoost performs more competitively, attaining 7.3% to 19.7% error reduction in comparison to ANN with 2.9% to 18.1%. We conjecture that XGBoost's split-fining approach also maps better to the timing classification problem.

3) Model Interpretation: It is often desirable to interpret the trained models to understand key features that lead to good estimation. In fact, we include linear model and gradient tree boosting because both provide a weight for each feature indicating the feature's importance in the models. For example, we can determine the importance of each feature in gradient tree boosting by the number of times that the feature is used as a split across all trees. Because of careful feature selection (Sections III-2 and III-3), we can interpret and discover knowledge from these models.

For clarity, Table V lists the most important categories of features for each of our estimation tasks in XGBoost. Not surprisingly, the post-implementation usage of each resource depends heavily on the corresponding built-in HLS estimates. This is expected because our approach uses HLS report as the starting point and is essentially "recalibrating" HLS report to match corresponding implementation report. Multiplexer-related features affect the number of LUTs and FFs because these resources are typically used for multiplexer implementation. Estimated clock period plays a role in estimating each resource because timing slack is a good indicator of how resources are mapped, placed, and routed. For timing classification, it is reasonable to see estimated and target clock periods as top features because they intuitively provide good indication of whether timing is met. Features of logic operations are also important because they reflect the amount of inaccuracy introduced by the additive timing estimation model built-in to the HLS tool.

VI. Related Work

Machine learning has been successfully applied within autotuning frameworks to effectively explore the large, high-dimensional space of tool-specific parameters controlling FPGA synthesis and implementation [13], [14]. For HLS, it has been leveraged for design space exploration to reduce the number of design candidates that need to run through the downstream implementation flow [15]. For resource estimation specifically, Koeplinger et al. learn three-layer ANN models to predict post-implementation resource usages from pre-characterized area models of a small set of architectural templates [16]. Instead of template-based designs, our techniques work for general HLS designs which are significantly more difficult to model. In addition, we employ a larger and more complex set of designs in our dataset and comparatively study both regression and classification models with more rigorous training, validation, and testing. Our work explores correlations and non-linearity within our data as well as both single-task and multi-task learning models.

VII. Conclusions

This work demonstrates that popular machine learning models can be trained to enable fast and accurate resource and timing estimations for HLS designs. The associated dataset is publicly available on the authors' website to allow interested members of the community to develop better models immediately without engaging in the notoriously slow implementation process for hundreds of design points. Our dataset can be further extended and improved with additional designs and data submitted by the community.

Acknowledgements

We thank Dr. Taemin Kim and Dr. Aravind Dasu from Intel for their helpful feedback. We also thank the anonymous reviewers for their insightful comments. This research was supported in part by NSF/Intel CAPA Award #1723715, NSF Award #1512937, ISRA Program under Intel Corp., and a research gift from Xilinx, Inc.

References

[1] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, "Aladdin: A Pre-RTL, Power-Performance Accelerator Simulator Enabling Large Design Space Exploration of Customized Architectures," *Int'l Symp. on Computer Architecture (ISCA)*, 2014.

[2] M. Makni, M. Baklouti, S. Niar, and M. Abid, "Hardware Resource Estimation for Heterogeneous FPGA-based SoCs," *Symp. on Applied Computing (SAC)*, 2017.

[3] J. Zhao et al., "COMBA: A Comprehensive Model-Based Analysis Framework for High Level Synthesis of Real Applications," *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2017.

[4] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii, "CHStone: A Benchmark Program Suite for Practical C-Based High-Level Synthesis," *Int'l Symp. on Circuits and Systems (ISCAS)*, 2008.

[5] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks, "Machsuite: Benchmarks for Accelerator Design and Customized Architectures," *Int'l Symp. on Workload Characterization (IISWC)*, 2014.

[6] B. C. Schafer and A. Mahapatra, "S2CBench: Synthesizable SystemC Benchmark Suite for High-Level Synthesis," *IEEE Embedded Systems Letters (ESL)*, 2014.

[7] Y. Zhou et al., "Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software Programmable FPGAs," *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2018.

[8] S. J. Pan and Q. Yang, "A Survey on Transfer Learning," *IEEE Trans. on Knowledge and Data Engineering*, 2010.

[9] K. Hornik, M. Stinchcombe, and H. White, "Multilayer Feedforward Networks are Universal Approximators," *Neural networks*, 1989.

[10] L. Mason, J. Baxter, P. L. Bartlett, and M. R. Frean, "Boosting Algorithms as Gradient Descent," *Advances in Neural Information Processing Systems (NIPS)*, 2000.

[11] T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System," *Int'l Conf. on Knowledge Discovery and Data Mining (KDD)*, 2016.

[12] F. Pedregosa et al., "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research (JMLR)*, 2011.

[13] Q. Yanghua, N. Kapre, H. Ng, and K. Teo, "Improving Classification Accuracy of a Machine Learning Approach for FPGA Timing Closure," *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2016.

[14] C. Xu, G. Liu, R. Zhao, S. Yang, G. Luo, and Z. Zhang, "A Parallel Bandit-Based Approach for Autotuning FPGA Compilation," *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2017.

[15] D. Liu and B. C. Schafer, "Efficient and Reliable High-Level Synthesis Design Space Explorer for FPGAs," *Int'l Conf. on Field Programmable Logic and Applications (FPL)*, 2016.

[16] D. Koeplinger et al., "Automatic Generation of Efficient Accelerators for Reconfigurable Hardware," *Int'l Symp. on Computer Architecture (ISCA)*, 2016.