Machine-learning-driven Architectural Selection of Adders and Multipliers in Logic Synthesis

JIAWEN CHENG, Tsinghua University, China
YONG XIAO, YUN SHAO, and GUANGHAI DONG, Giga Design Automation Co., Ltd., China
SONGLIN LYU and WENJIAN YU, Tsinghua University, China

Designing high-performance adders and multiplier components for diverse specifications and constraints is of practical concern. However, selecting the best architecture for adder or multiplier, which largely affects the performance of synthesized circuits, is difficult. To tackle this difficulty, a machine-learning-driven approach is proposed for automatic architectural selection of adders and multipliers. It trains a machine learning model for classification through learning a number of existing design schemes and their performance data. Experimental results show that the proposed approach based on a multi-perception neural network achieves as high as 94% prediction accuracy with negligible inference time. On a CPU server, the proposed approach runs about 4× faster than a brute-force approach trying four candidate architectures and consumes 10%~20% less runtime than the DesignWare datapath generator for obtaining the optimal adder/multiplier circuit. The adder (multiplier) generated with the proposed approach achieves performance metrics close to the optimal and has 1.6% (5.2%) less area and 2.2% (7.1%) more worst negative slack averagely than that generated with the DesignWare datapath generator. Our experiment also shows that the proposed approach is not sensitive to the size of training subset.

CCS Concepts: • **Hardware → Circuit optimization**; • **Computing methodologies → Machine learning approaches;**

Additional Key Words and Phrases: Logic synthesis, datapath, adder/multiplier, machine learning, architectural selection

ACM Reference format:
Jiawen Cheng, Yong Xiao, Yun Shao, Guanghai Dong, Songlin Lyu, and Wenjian Yu. 2023. Machine-learning-driven Architectural Selection of Adders and Multipliers in Logic Synthesis. *ACM Trans. Des. Autom. Electron. Syst.* 28, 2, Article 20 (March 2023), 16 pages.
https://doi.org/10.1145/3560712

1 INTRODUCTION

With the outbreak of the **internet of things (IoT)** and embedded **artificial intelligence (AI)** applications, various low-power and high-performance **integrated circuits (ICs)** are greatly

This work is partially supported by the National Natural Science Foundation of China (NSFC) Research Projects under Grant 62090025.
Authors' addresses: J. Cheng, S. Lyu, and W. Yu (corresponding author), Tsinghua University, Beijing, China, 100084; emails: {cjw21, lvst17}@mails.tsinghua.edu.cn, yu-wj@tsinghua.edu.cn; Y. Xiao, Y. Shao, and G. Dong, Giga Design Automation Co., Ltd., Shenzhen, Guangdong, China, 518055; emails: {yxiao, yshao, ghdong}@giga-da.com.

1084-4309/2023/03-ART20 $15.00
https://doi.org/10.1145/3560712

demanded. Various **electronic design automation (EDA)** technologies have been employed to build the flow for designing, analyzing, and verifying these ICs. Logic synthesis is an important part of EDA that translates the high-level description of a circuit (such as in Verilog) into a gate-level circuit and imposes some optimizations on it.

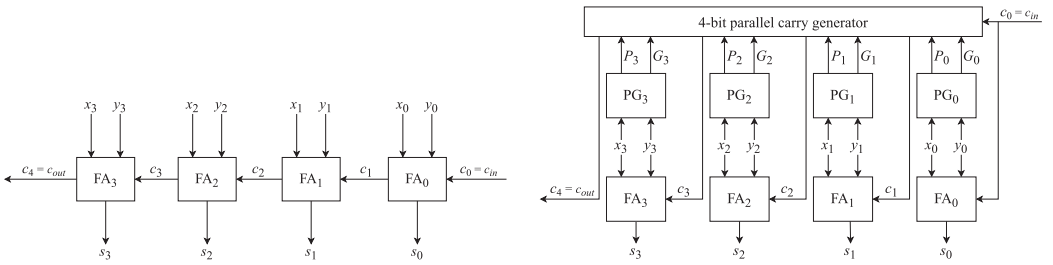
Datapath, which consists of arithmetic operation units, registers, and buses, and performs data processing operations [11], is a key module in ICs. Addition and multiplication are the most fundamental and frequently used arithmetic operations in the datapath, and the optimization for the basic adder and multiplier components affects the datapath's cost and performance significantly. The logic synthesis tool performs circuit optimization iteratively, where other parts of the circuit impose constraints on the adder and multiplier components in terms of bit width, working frequency, input and output delay, and so on. These constraints may change during the iterations. Therefore, how to implement the optimal adder or multiplier (in terms of a certain pre-defined cost function) under a wide variety of constraints is of concern.

Different architectures for adders and multipliers have been proposed, which have different costs or performances under different scenarios. The basic adder architecture is the carry ripple adder [32], which connects several full adders in series. To reduce the timing delay, a variety of parallel adder architectures were proposed, such as carry lookahead adder [18] and parallel prefix adders, including Sklansky adder [29], Brent-Kung adder [4], and Kogge-Stone adder [13]. The basic multiplier is shift add multiplier [1]. Booth proposed the idea of properly using addition and subtraction to accelerate multiplication [2], which derives the Booth multipliers. Several tree architectures were also proposed to further improve the efficiency of multiplication, such as Wallace tree multiplier [31] and Dadda tree multiplier [7].

There are some works for optimizing the design of prefix adder for specific performance objectives, solely at the stage of logic synthesis or considering physical design metrics [14–17, 24, 25, 35, 36]. These works are based on a certain kind of adder architecture, instead of choosing the architecture that usually has a larger impact on the circuit's performance in the iterative logic-synthesis optimization process. For example, an algorithm was proposed to pursue area minimization for parallel prefix adders under bitwise delay constraints [17] and a polynomial time algorithm was proposed to synthesize parallel prefix adders with maximum fan-out restriction [24]. They do not consider non-parallel prefix adder architectures such as carry select adder, carry skip adder, and carry lookahead adder. How to automatically select the architecture of an adder or a multiplier in a larger design space is the concern of our work. And, it is orthogonal to the work for optimizing a certain kind of adder or multiplier architecture.

However, the great progress of **machine learning (ML)** promotes its application to logic synthesis [23] and other EDA problems [6, 33]. For example, a cross-layer optimization approach for prefix adder design was recently proposed in Reference [16] that employs machine learning-based design space exploration to predict the Pareto frontier of adders in the physical domain. And in Reference [8], the **decision tree (DT)**-based approach was proposed for fast logic minimization of Boolean functions. Further, Geng et al. employed **graph convolutional networks (GCN)** to do multi-objective design space exploration for optimizing high-speed prefix adders [9].

In this work, we investigate the architecture selection problem of adder and multiplier design with the help of machine learning technology. A machine-learning-driven approach is proposed to select adder/multiplier architecture among several candidates of architecture for a given set of specifications/constraints. The proposed approach consists of the following three steps: First, different adders (multipliers) following the candidate architectures are synthesized under a set of various specifications/constraints. According to the specified selection criterion or performance metric, the best architecture is selected (labeled) for each specification/constraint. This results in a dataset for training, where each data includes the correspondence between the specification/constraint and



(a) Carry ripple adder (CRA) (b) Overall architecture of SA, BKA, and KSA

Fig. 1. Different architectures of an adder in 4-bit width.

the best architecture. Then, with the training set, a supervised ML method, e.g., **support vector machine (SVM), multi-layer perception neural network (MLP-NN)**, or **decision tree (DT)**, is applied to train a multi-class classifier for selecting the architecture. Finally, the trained ML classifier is used to predict the architecture of the adder/multiplier for a tested specification/constraint. It should be pointed out that the proposed machine-learning-driven approach is not restricted to the optimization of adder and multiplier design. It can be easily extended to the problem of divider design and would help synthesize better datapath circuits more efficiently.

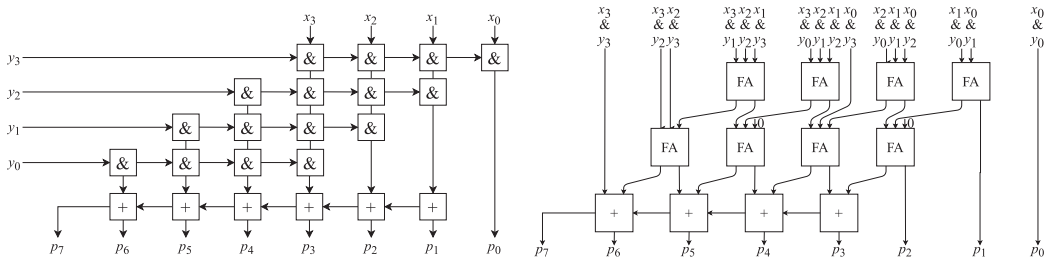
The proposed approach is validated with the experiments based on the results obtained with a commercial logic synthesis tool. We consider the variable inputs of bit widths of operands, clock period of the component, input delay, and output delay, set four candidate architectures for adder and multiplier individually, and evaluate the performance of SVM, MLP-NN, and DT as a classifier. The results show that the MLP-NN-based classifier outperforms the SVM and DT models and achieves as high as 94% prediction accuracy and a 0.9489 weighted F1-score. The proposed approach is compared with a brute-force approach trying four candidate architectures and the DesignWare datapath generator, which is the component inside Design Compiler (a commercial logic synthesis tool) for automatically selecting adder/multiplier architecture. The proposed approach has negligible runtime penalty on inference using the trained classifier, enabling about 4× speedup over the brute-force approach and 10%~20% time saving over DesignWare datapath generator, on the same CPU machine. The adder (multiplier) generated with the proposed approach achieves performance metrics close to the optimal and has 1.6% (5.2%) less area and 2.2% (7.1%) more **worst negative slack (WNS)** averagely than that generated with the DesignWare datapath generator. In addition, our experiment shows that the proposed approach is not sensitive to the size of training subset.

2 BACKGROUND

In this section, we first briefly introduce the widely used architectures of adder and multiplier. Without loss of generality, we suppose the bit widths of two operands of an adder/multiplier are both n. Then, several machine learning techniques for classification are reviewed.

2.1 Architecture of Adder

The input of an adder includes an n-bit operand x, an n-bit operand y, and a 1-bit carry c_{in}. The output includes an n-bit sum s and a 1-bit carry c_{out}. The simplest way to implement an adder is to connect n **full adders (FAs)** in series. In this way, the carry propagates from the lowest significant bit to the highest significant bit like a ripple. So, it is called **carry ripple adder (CRA)** [32]. Figure 1(a) illustrates the architecture of a typical 4-bit CRA.



(a) Shift add multiplier (SAM) (b) Wallace tree multiplier (WTM)

Fig. 2. Different architectures of a multiplier in 4-bit width.

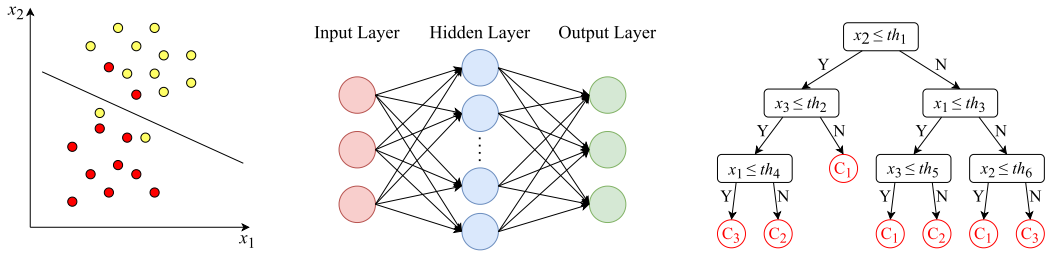
Carry generation is the main source of timing delay in CRA. For reducing the delay, a variety of parallel adder architectures were proposed to accelerate carry generation, by generating each carry c_i in parallel. Among them, **carry lookahead adder (CLA)** [18] and parallel prefix adders such as Sklansky adder (SA) [29], **Brent-Kung adder (BKA)** [4], and **Kogge-Stone adder (KSA)** [13] are commonly used. Because c_i only depends on the two operands' low $i-1$ bits $x_{i-1}x_{i-1}\ldots x_0$, $y_{i-1}y_{i-1}\ldots y_0$ and the input carry c_{in}, each FA$_i$ has all the information for calculating s_i and c_{i+1} at the same time. Based on this, CLA, SA, BKA, and KSA all utilize a parallel carry generator and n **propagation generators (PG$_i$)** to generate carries in parallel. They have similar overall architecture, which of 4-bit width is illustrated in Figure 1(b). The only difference among them is the structure of the parallel carry generator. The overhead of these architectures is larger area compared to CRA.

2.2 Architecture of Multiplier

The input of a multiplier includes two n-bit operands x, y and the output includes an $(2n)$-bit product p. **Shift add multiplier (SAM)** [1] simulates the process of long multiplication, using several adders to add each line of n-bit summands. Figure 2(a) illustrates the architecture of 4-bit SAM, where the "&" block is an AND gate and the "+" block is an adder that generates the sum and carry of input bits (the same below). SAM only multiplies one bit and the multiplicand at a time, which is a lack of efficiency. Therefore, a technique multiplying multiple bits can be adopted to reduce the number of additions [19]. The SAM that multiplies m bits at a time is called Radix-2^{m-1} SAM.

To accelerate the multiplication, another architecture, i.e., **Booth multiplier (BM)** [2], was proposed. BM properly utilizes addition and subtraction to reduce the number of additions in SAM. It is based on the fact that sequential 0s in the multiplicator require no addition but only shifting of the partial product, and a string of 1 s in the multiplicator from the kth bit to the mth bit can be treated as the subtraction of 2^k from 2^{m+1}. According to the number of bits checked in one iteration of accumulating, BM adopts different recoding schemes for partial product generation. The BM that checks m bits in one iteration is called Radix-2^{m-1} Booth multiplier.

Another kind of multiplier, **Wallace tree multiplier (WTM)** [31], was proposed to reduce the amount of additions by adding three bits at one time instead of two bits in SAM. It first generates all sums simultaneously, then groups every three sums of one bit together into an FA to generate carry out and sum for the next level. If there are only two remaining sums, then one input bit of the FA will be replaced by zero. Otherwise, the remaining sum is directly outputted to the next level. WTM keeps grouping three sums to carry out and sum until reaching the last level where high-order bits can only generate two sums. Finally, they are put into an adder to generate the final product. The architecture of 4-bit WTM is illustrated in Figure 2(b).



(a) SVM (b) MLP-NN (c) DT

Fig. 3. The classifiers based on SVM, MLP-NN, and DT.

2.3 Machine Learning Techniques for Classification

In ML terminology, classification is the process of predicting the category (label) of a new observation (data) based on a training set where data has known labels. Many ML techniques can be used to do classification. Below, we briefly introduce a few of them.

Support vector machine (SVM) [5] is a classic ML technique for binary classification. In SVM, a hyperplane is constructed to separate two groups of labeled data, as shown in Figure 3(a). The special properties of the classification hyperplane ensure the high generalization ability of SVM. If the sets of data to discriminate are not linearly separable, the kernel trick can be used to map the original data to a higher dimensional space, presumably making the separation easier. After this mapping, the inner product in the original space needs to be replaced by a new one called *kernel function,* which is then used for training the classifier. An example of kernel function is the Gaussian **radial basis function (RBF)**:

$$K(x_1, x_2) = e^{-\gamma \|x_1 - x_2\|^2}, \tag{1}$$

where x_1 and x_2 are two data in the original space, and $\gamma > 0$ is a given parameter. With the **One-vs-One (OvO)** or **One-vs-Rest (OvR)** strategy, SVM can be extended to do multi-class classification. The OvO strategy splits a multi-class classification problem into the binary classification problem for all pairs of classes, while the OvR strategy splits the problem into the binary classifications for each class versus the remaining classes.

Multi-layer perceptron neural network (MLP-NN) is a kind of simple artificial neural network that can also be used in classification. With linear transformations and a nonlinear activation function, MLP-NN maps inputs into outputs, as shown in Figure 3(b). Similar to other deep learning methods, MLP-NN uses the error back-propagation to train its parameters [26]. At the output layer, the softmax function is often used as the activation function for classification.

Decision tree (DT) has a binary tree structure, as shown in Figure 3(c), where each non-leaf node represents a test on a feature. An incoming observation goes to the left child node if and only if the condition is satisfied. When the observation reaches the leaf node, it is classified as the category belonging to that node. Several algorithms have been proposed to generate a DT, among which the most commonly used are CART [3], ID3 [21], and C4.5 [22].

3 METHODOLOGY

In this section, we present the approach for selecting the architecture for the logic synthesis of adders (multipliers) based on machine learning, along with implementation details.

Table 1. The Specifications/Constraints Used in This Work

Specifications/Constraints	Description
w_1	Bit width of operand 1
w_2	Bit width of operand 2
T	Clock period
d_{i1}	Input delay of operand 1
d_{i2}	Input delay of operand 2
d_o	Output delay

3.1 The Overall Idea

The idea is that we can select the optimal architecture of the adder/multiplier via a well-trained classifier. The optimality refers to the best among several candidate architectures mentioned in Sections 2.1 and 2.2 and is in terms of certain predefined performance metric. This turns into a classification problem that requires a dataset for training and a proper ML technique.

The gate-level circuit of an adder/multiplier component can be synthesized from its high-level architecture with logic synthesis tools under a certain standard cell library and specifications/constraints. Suppose that the standard cell library is fixed, the specifications/constraints such as clock period are the data input to the classifier. In this work, the specifications/constraints used are listed in Table 1.

These parameters also form the features of each data, while the classifier's output (label for each data) is the selected adder/multiplier category. The data representation of this problem is relatively simple, so the basic ML techniques in Section 2.3 are suitable for this classification problem.

3.2 Generation of Parameter Samples for Training Data

We consider the sample values for each input feature or specification/constraint. For w_1 and w_2, we set the sample value to all multiples of 4 ranging from 4 to 64. This covers various adders (multipliers) in the arithmetic systems using at most 64-bit numbers. Besides, due to the symmetry of the two operands of an adder/multiplier, we further specify that w_1 is not less than w_2. Since input and output delays tend to increase from the **least significant bit (LSB)** to the **most significant bit (MSB)** in practice, we set them similarly. Given a slope parameter s and $0 \leq s \leq 1$, the input/output delay of the kth ($0 \leq k < n$) bit is set to $s \cdot \frac{k}{n-1} \cdot T$, where n is the bit width of the input/output. Namely, the input/output delay of LSB is set to 0 and that of MSB is set to sT. For d_{i1}, d_{i2}, and d_o, their slope parameters s are all sampled within 0, 0.2, 0.4, and 0.6, which guarantees that the timing constraints can be satisfied.

For other specifications/constraints, their value range for sampling should be properly determined. Take the clock period T as an example. If its sample value is too small to be satisfied, then the result from the logic synthesis tool will be the component with minimized timing delay and relatively large area. And, this result will keep unchanged for smaller clock period. However, if the clock period increases beyond an upper bound, then the synthesized result will converge to a component with a smaller area and relatively large delay. Thus, we first determine the lower and upper bounds of the reasonable clock period for each set of bit widths. Because the clock period should be larger than the delay, we consider the delay of the synthesized component for setting the bounds of the clock period. Two extreme conditions with 0 and an infinitely large clock period are input to the logic synthesis tool. Then, the delays of the synthesized components in these two extreme scenarios are set as the lower bound and upper bound of the clock period, respectively. The experimental results show that both bounds are approximately proportional to the bit width, as shown in Figure 4. The linear regression formulas obtained with our experiments are:

$$T_{min}(w) = 0.004w + 0.246,\ T_{max}(w) = 0.091w + 0.044 \tag{2}$$

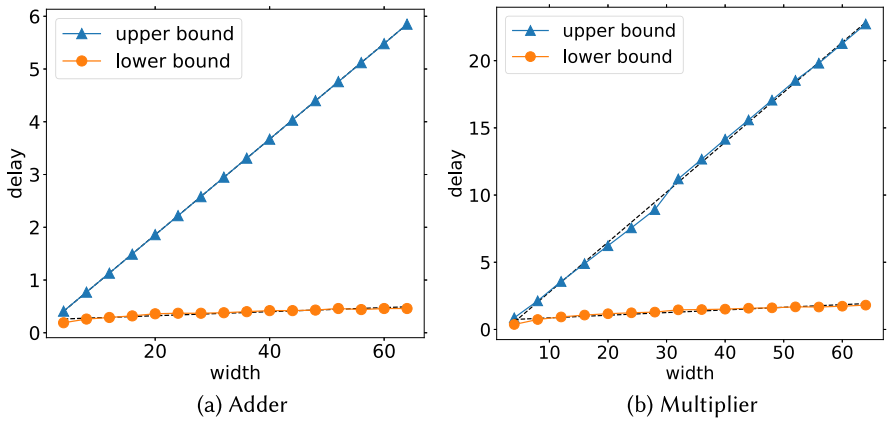


Fig. 4. The trends of lower and upper bounds of delay of the synthesized component.

for adder and

$$T_{min}(w) = 0.010w + 0.664,\ T_{max}(w) = 0.372w - 0.940 \tag{3}$$

for multiplier, where $w = (w_1 + w_2)/2$. The unit of clock period is nanosecond.

Multiple sample values of the clock period should be set. With the experiments, we see that the synthesized result (e.g., area) is more sensitive to the clock period when the clock period is smaller. It suggests that we should set more samples at the smaller-value range, instead of set samples uniformly in the whole value range. Supposing the number of discrete values is N, we can generate the sample values of clock periods with the following quadratic formula:

$$T_i = T_{\text{min}} + (T_{\text{max}} - T_{\text{min}}) \left(\frac{i}{N-1} \right)^2, 0 \le i < N, \tag{4}$$

where T_i is the ith sample of the clock period, and T_{min} and T_{max} are the lower and upper bounds of the clock period obtained with Equations (2) or (3).

For other specifications/constraints, a similar approach can be applied for generating the sample values. With the sample values for each input parameter, a set of training data can be generated representing various specifications/constraints for which the optimized adder/multiplier design is needed.

3.3 Selection Criterion of Architecture

The selection criterion of architecture is related to the design target. The optimal architecture may be different under different targets (performance metrics). The actual performance metric for designing a good adder/multiplier is usually a combination of area, delay, and power consumption and depends on the application scenario of the whole circuit. In this work, we consider the following two selection criteria for different performance preferences in actual scenarios:

(1) **Delay first, area second (DFAS)**. This criterion is for first considering the satisfaction of timing constraint and then minimizing the area. It selects the architecture with the minimum area if satisfying the timing constraint, i.e., the WNS t_s is not less than 0. Otherwise, the one with the highest WNS will be selected. The corresponding cost function for minimization is

$$f_{\text{cost}} = \begin{cases} \text{Area} - \lambda \cdot t_s, & t_s < 0, \\ \text{Area}, & t_s \ge 0, \end{cases} \tag{5}$$

where Area is the area of the synthesized adder/multiplier, λ is a large positive number.

(2) **Balanced delay and area (BDA)**. This is for the scenario where the timing constraint is not a dominant factor for design. The corresponding cost function for minimization is

$$f_{\text{cost}} = \frac{\text{Area}}{\text{Area}_{\min}} - \lambda' \cdot \min\left(\frac{t_s}{T}, 0\right), \tag{6}$$

where $\text{Area}_{\min}$ is the minimum area of all candidate architectures, λ' is the weight of delay, and T is the clock period. $\text{Area}/\text{Area}_{\min}$ and t_s/T represent the normalized area and WNS, respectively. With different values of λ', the selected architecture brings more optimized delay or area.

3.4 Implementation Details

Each entry of data has six features and a label, which are the input to the classifiers based on SVM, MLP-NN, or DT. For SVM, the RBF in Equation (1) with $\gamma = 1/(5\text{Var}_x)$ is used as the kernel function, where Var_x denotes the variance of the input data. To do multi-class classification, we employ the OvO strategy. For MLP-NN, a four-layer neural network including two 16-node hidden layers is built. Batch normalization [10] is used and the activation function for the hidden layers is ReLU:

$$\text{ReLU}(x) = \max(0, x). \tag{7}$$

The Adam optimizer [12] is applied to training the weights and biases of linear layers. The weight decay of the L2 penalty and the initial learning rate are both set to 1e-3. The maximum epoch is 1,000 and an early-stop technique with a patience of 50 epochs is used. For DT, the CART algorithm [3] is applied to construct the tree structure.

4 EXPERIMENTAL RESULTS

To validate the proposed approach, we consider the designs of adder and multiplier with varied bit widths of operands and clock period. Their sample values are set as described in Section 3.2. Besides, the number of samples for the clock period T is $N_T = 11$. Considering all parameter combinations, we generate $N_T \cdot (\frac{N_w(N_w+1)}{2}) \cdot N_{i1} \cdot N_{i2} \cdot N_o = 11 \times 16 \times 17 \times 4 \times 4 \times 4/2 = 95,744$ items of data for adder and multiplier individually, where $N_w = 16$ is the number of sample values for bit width w. N_{i1}, N_{i2}, and N_o (all equal 4) represent the number of sample values for the slope parameters s of d_{i1}, d_{i2}, and d_o, respectively. Design Compiler (v2016) [30] from Synopsys, Inc. is used as the logic synthesis tool to generate the circuits of the adder and multiplier components. The architectures of CLA, SA, BKA, and KSA for adder and SAM, Radix-4 SAM (R4SAM), Radix-4 BM (R4BM), and Radix-8 BM (R8BM) for multiplier are the candidates for selection. For each combination of a data item and an architecture, the component is synthesized with the Nangate 45 nm standard cell library [28] under the specifications/constraints represented by the data. In terms of the two criteria (DFAS and BDA), the architecture resulting in the lowest cost is regarded as the best and is attached to the data as the label or ground truth.

The distributions of the best architecture for adder and multiplier datasets are shown in Table 2. The difference in the distributions shows the influence of selection criteria on adder/multiplier selection.

Each dataset is split into a training subset and a test subset. Then, we train the classifier using the training subset with SVM, MLP-NN, and DT techniques. The test subset is used to evaluate the performance of the resulted classifiers. For this purpose, the weighted F1-score metric [34] is employed. The definition of F1-score is

$$\text{F1} = \frac{2P \cdot R}{P + R}, \text{ with } P = \frac{\text{TP}}{\text{TP} + \text{FP}}, \; R = \frac{\text{TP}}{\text{TP} + \text{FN}}, \tag{8}$$

Table 2. The Distributions of the Best Architecture in Four Categories under Two Selection Criteria

#Data of Adder	Criterion		#Data of Multiplier	Criterion	
	DFAS	BDA		DFAS	BDA
CLA	17,818	15,847	SAM	10,034	10,168
SA	23,598	27,942	R4SAM	13,448	10,708
BKA	34,356	37,602	R4BM	50,915	52,139
KSA	19,972	14,353	R8BM	21,347	22,729
Total	**95,744**	**95,744**	**Total**	**95,744**	**95,744**

Table 3. Weighted F1-Scores of SVM-based, MLP-NN-based, and DT-based Classifiers on the Test Subsets of Two Datasets

F1-score (Dataset 1)	Criterion		F1-score (Dataset 2)	Criterion	
	DFAS	BDA		DFAS	BDA
SVM	0.9222	0.8830	SVM	0.9125	0.9255
MLP-NN	**0.9347**	**0.9160**	MLP-NN	**0.9409**	**0.9489**
DT	0.9062	0.8928	DT	0.9160	0.9202

where P is the precision, R is the recall, and TP, FP, and FN are the numbers of true positive, false positive, and false negative samples, respectively. F1-score $\in[0, 1]$ is a harmonic average of precision and recall. The weighted F1-score is

$$\mathrm{F1}_{\mathrm{weighted}} = \frac{\sum_{i=1}^{k} N_i \cdot \mathrm{F1}_i}{\sum_{i=1}^{k} N_i}, \tag{9}$$

where k is the number of categories, N_i is the number of samples in category i, and $\mathrm{F1}_i$ is the F1-score of category i.

The classifiers are implemented in Python. Specifically, SVM-based and DT-based classifiers are implemented with the Scikit-Learn package [27], while MLP-NN-based classifier is with the PyTorch package [20]. All experiments are carried out on a Linux server with two 8-core Intel Xeon E5-2620 CPUs @ 2.10GHz.

4.1 Results of the Classifiers for Architectural Selection

We refer to the dataset of adder as Dataset 1 and the dataset of multiplier as Dataset 2. For each of them, 67,021 entries of data (70% of the whole dataset) are used as the training subset and the remainder as the test subset. The DFAS or BDA ($\lambda' = 8$) selection criterion is set as the selection criterion.

With the test subset, we evaluate the SVM-based, MLP-NN-based, and DT-based classifiers. The confusion matrices for selecting adder and multiplier architecture are shown in Figures 5 and 6, respectively. From it, we see that the prediction accuracy ranges from 88% to 94%.

The weighted F1-scores are listed in Table 3. From the results, we observe that the classifiers overall have good performance, among which the MLP-NN-based classifier remarkably outperforms the other two classifiers. Besides, the classifier for predicting adder architecture with DFAS criterion has better performance, with up to 0.9489 F1-score.

4.2 Comparison with DesignWare Datapath Generator

DesignWare datapath generator is the component inside Design Compiler that automatically selects adder/multiplier architecture. Therefore, it is necessary to compare the proposed approach

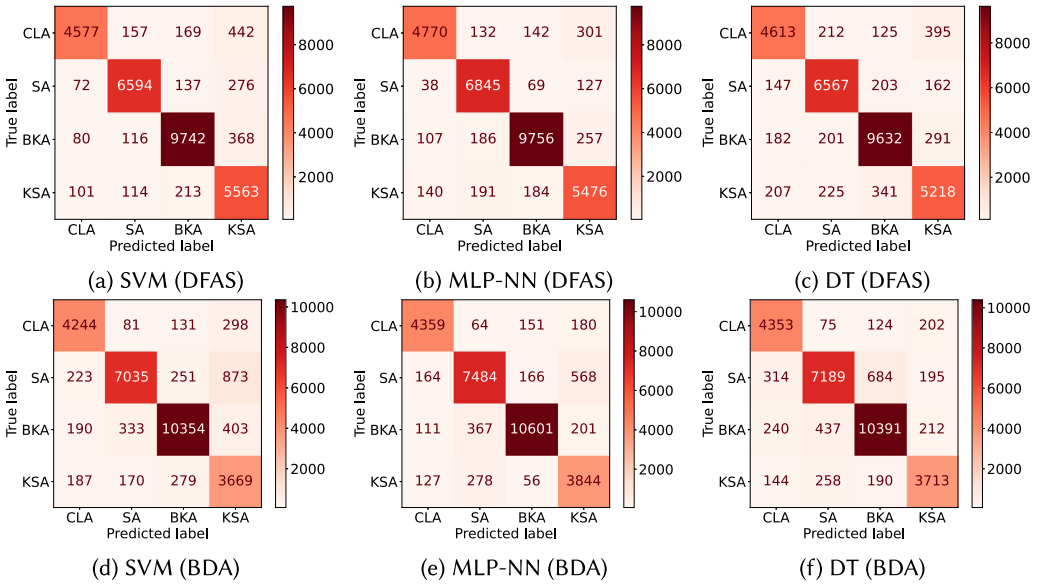


Fig. 5. The inference results of SVM-based, MLP-NN-based, and DT-based classifiers on the test subset of Dataset 1, under different selection criteria.

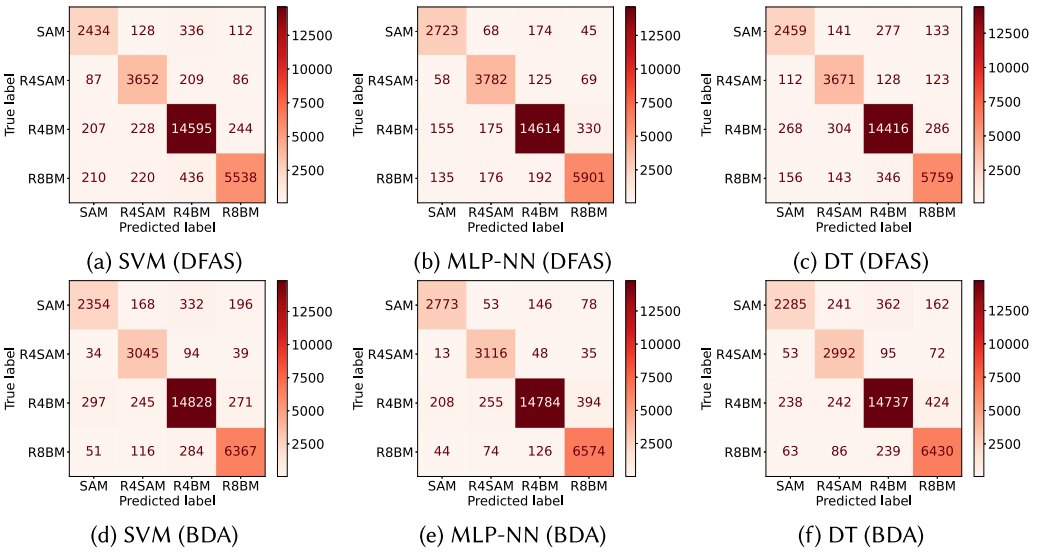


Fig. 6. The inference results of SVM-based, MLP-NN-based, and DT-based classifiers on the test subset of Dataset 2, under different selection criteria.

with it. We use the "set_dp_smartgen_options" command in Design Compiler to control the architecture used by DesignWare datapath generator in the process of logic synthesis.

To compare the efficiency of different methods, we record the inference time of the classifiers and the time for synthesizing the component with the selected architecture. The average times for synthesizing a component (including architectural selection) are listed in Table 4. DesignWare

Table 4. Average Time for Synthesizing a Component with Difference Approaches

Time (s)	SVM	MLP-NN	DT	DesignWare Datapath Generator	Brute-force
Adder	5.42	6.12	5.29	6.75	20.98
Multiplier	21.98	22.54	21.33	28.34	85.64

Table 5. Average Cost, Area, Worst Negative Slack, and Critical Path Delay of the Synthesized Components with Different Approaches

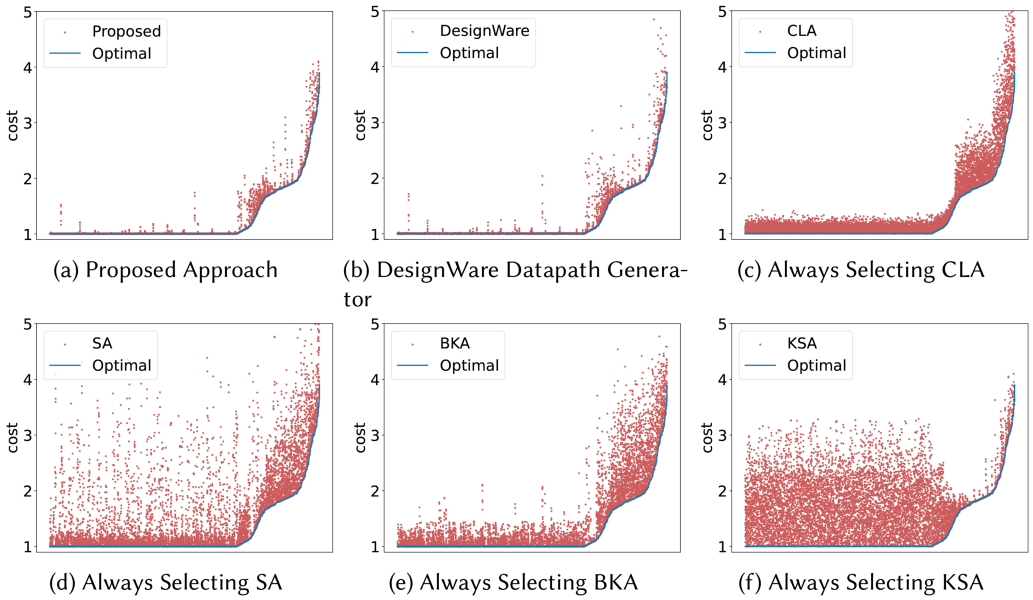
Dataset 1	Metric				Dataset 2	Metric			
	Cost	Area (μm^2)	WNS (ns)	CPD (ns)		Cost	Area (μm^2)	WNS (ns)	CPD (ns)
Optimal	1.263	296.0	0.138	1.140	Optimal	3.121	4,342.8	0.183	4.294
Proposed	1.265	297.1	0.137	1.141	Proposed	3.128	4,348.2	0.180	4.297
DesignWare	1.276	302.0	0.134	1.144	DesignWare	3.209	4,586.3	0.168	4.309
CLA	1.913	403.5	0.078	1.200	SAM	3.703	5,900.6	0.124	4.353
SA	1.499	378.9	0.032	1.246	R4SAM	3.435	6,291.3	0.065	4.412
BKA	1.421	367.2	0.098	1.180	R4BM	3.289	4,729.7	0.161	4.316
KSA	1.805	541.0	0.084	1.194	R8BM	3.721	4,872.9	0.082	4.395

datapath generator and a brute-force approach, which synthesizes the components of four architectures and then picks the best one, are also tested as the baselines. From Table 4, we see that the proposed ML-driven approach saves 10%~20% time compared to DesignWare datapath generator and only costs about one-fourth of the time by the brute-force approach, as the proposed approach only synthesizes one component and the inference time of the trained classifier is marginal.

To compare the performance of components obtained with the proposed approach and the baselines, we draw the cost functions of BDA criterion for all 28,723 test data of two datasets in Figure 7(a) and 8(a). "Optimal" and "Proposed" denote the values of cost function derived from the brute-force approach (reordered to exhibit a non-declining curve) and our MLP-NN-based approach, respectively. From the figure, we see that in most cases our approach achieves the best performance of the synthesized component. As a comparison, the BDA costs of the components derived from DesignWare datapath generator and always selecting a single architecture are shown in Figures 7(b)–(f) and 8(b)–(f). They highlight that the proposed approach leads to much better circuit performance with a little sacrifice on the time for ML model inference. In addition, the components selected by the proposed approach have better performance than those by DesignWare datapath generator while costing less time. The average cost, area, WNS, and **critical path delay (CPD)** of the synthesized components with different approaches are listed in Table 5. This also demonstrates that the performance of the components obtained by the proposed approach is distinctly better than those by DesignWare datapath generator and close to the optimal. For a better demonstration of the proposed approach's benefits, we group the area and CPD of the synthesized components by 11 different clock period settings described in Equation (4) and then calculate the average. The results are listed in Tables 6–9 and show that the proposed approach is consistently better than DesignWare datapath generator with different clock periods. We also group the area and CPD by bitwidths of the adders/multipliers and list the averaged metrics in Table 10 for adders/multipliers with both bitwidths not less than 32 and Table 11 for those with at least one bitwidth below 32. The results show that the proposed approach is suitable for both higher bitwidth adders/multipliers and the other cases.

4.3 Sensitivity of the ML Techniques to Training Subset's Size

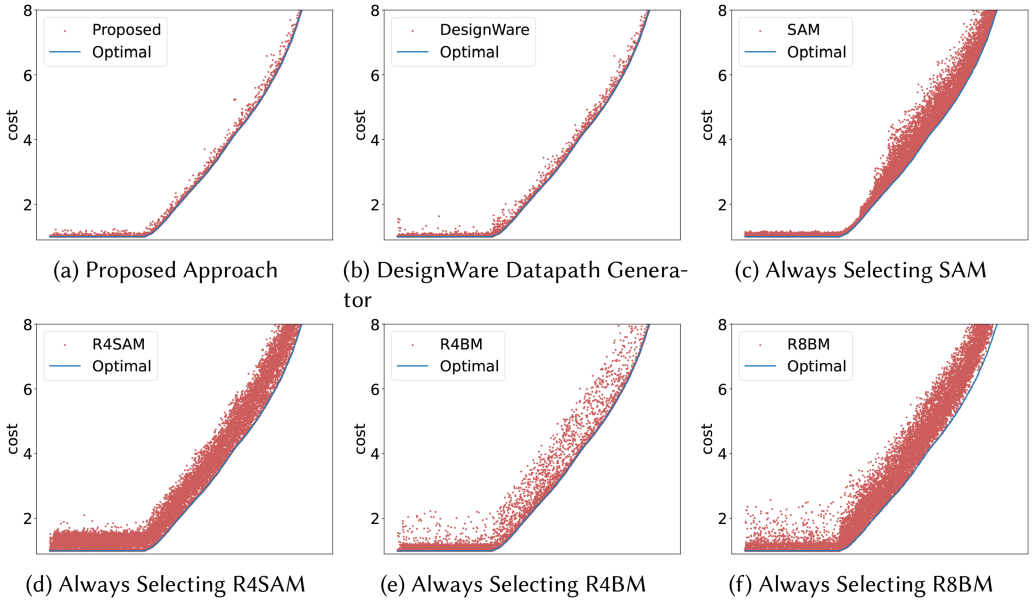
To evaluate how the size of training subset affects the performance of the classifiers, more datasets with different sizes of training subsets are constructed. The smallest training subset only contains



(a) Proposed Approach (b) DesignWare Datapath Generator (c) Always Selecting CLA

(d) Always Selecting SA (e) Always Selecting BKA (f) Always Selecting KSA

Fig. 7. The cost values of the synthesized adders derived from all 28,723 test data of Dataset 1 and different architectural selection strategies.



(a) Proposed Approach (b) DesignWare Datapath Generator (c) Always Selecting SAM

(d) Always Selecting R4SAM (e) Always Selecting R4BM (f) Always Selecting R8BM

Fig. 8. The cost values of the synthesized multipliers derived from all 28,723 test data of Dataset 2 and different architectural selection strategies.

10% of the whole dataset (i.e., 9,574 data). Then, for each one the classifiers based on SVM, MLP-NN, and DT are trained and evaluated for classification accuracy on the corresponding test subsets. The results of weighted F1-scores are plotted in Figure 9, which illustrates that the classifiers' performance decreases very slowly as the training set becomes smaller and the MLP-NN-based

Table 6. Average Area of the Synthesized Components in Dataset 1 (Adder) Grouped by 11 Clock Period Settings with Different Approaches (μm^2)

Clock Period	T_{min}	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{max}
Optimal	345.6	335.6	325.2	316.6	306.1	295.6	285.8	276.5	265.2	256.8	246.6
Proposed	346.8	336.7	326.3	318.0	307.3	297.3	287.3	276.6	267.7	257.7	247.0
DesignWare	350.8	339.7	330.8	323.2	316.7	299.9	291.1	281.8	273.8	260.5	253.7
CLA	453.1	447.6	434.6	426.5	412.4	398.2	392.6	381.1	377.6	361.4	352.8
SA	427.2	420.1	410.4	402.1	385.4	380.6	369.2	361.5	350.6	337.1	323.5
BKA	411.6	410.9	400.4	385.7	378.0	370.5	355.8	341.7	338.0	330.9	315.3
KSA	592.9	579.7	572.8	561.6	548.1	542.5	526.1	518.0	512.0	503.6	494.2

Table 7. Average Critical Path Delay of the Synthesized Components in Dataset 1 (Adder) Grouped by 11 Clock Period Settings with Different Approaches (ns)

Clock Period	T_{min}	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{max}
Optimal	0.231	0.261	0.337	0.470	0.651	0.882	1.168	1.497	1.885	2.329	2.822
Proposed	0.231	0.261	0.338	0.470	0.652	0.885	1.169	1.499	1.887	2.331	2.825
DesignWare	0.233	0.263	0.342	0.472	0.655	0.887	1.169	1.504	1.896	2.333	2.827
CLA	0.294	0.321	0.402	0.530	0.710	0.941	1.225	1.559	1.955	2.386	2.879
SA	0.338	0.373	0.445	0.575	0.752	0.987	1.272	1.613	2.000	2.432	2.920
BKA	0.281	0.300	0.379	0.510	0.687	0.918	1.207	1.538	1.931	2.368	2.862
KSA	0.288	0.315	0.393	0.520	0.705	0.934	1.223	1.555	1.945	2.384	2.870

Table 8. Average Area of the Synthesized Components in Dataset 2 (Multiplier) Grouped by 11 Clock Period Settings with Different Approaches (μm^2)

Clock Period	T_{min}	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{max}
Optimal	4,591.7	4,562.1	4,500.0	4,440.4	4,387.8	4,348.4	4,307.7	4,249.1	4,184.8	4,118.9	4,079.7
Proposed	4,594.4	4,569.9	4,505.9	4,448.2	4,394.0	4,351.6	4,311.6	4,255.9	4,194.3	4,121.8	4,082.1
DesignWare	4,859.4	4,779.0	4,715.1	4,672.1	4,618.9	4,575.2	4,540.3	4,501.9	4,438.1	4,396.8	4,352.2
SAM	6,170.0	6,099.4	6,050.2	6,003.1	5,969.0	5,903.9	5,840.1	5,773.5	5,728.2	5,712.0	5,657.7
R4SAM	6,553.0	6,481.6	6,419.7	6,410.3	6,339.7	6,270.3	6,236.9	6,208.0	6,131.8	6,106.0	6,047.9
R4BM	4,999.3	4,949.5	4,858.6	4,827.2	4,769.2	4,743.9	4,687.9	4,608.5	4,589.7	4,527.2	4,465.6
R8BM	5,118.9	5,054.2	5,026.7	4,977.0	4,908.7	4,865.6	4,827.6	4,779.6	4,741.3	4,674.7	4,627.5

Table 9. Average Critical Path Delay of the Synthesized Components in Dataset 2 (Multiplier) Grouped by 11 Clock Period Settings with Different Approaches (ns)

Clock Period	T_{min}	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{max}
Optimal	0.802	0.901	1.197	1.704	2.400	3.293	4.396	5.695	7.185	8.881	10.777
Proposed	0.803	0.902	1.200	1.706	2.403	3.294	4.399	5.698	7.192	8.888	10.784
DesignWare	0.811	0.911	1.218	1.716	2.412	3.312	4.410	5.701	7.201	8.904	10.798
SAM	0.855	0.961	1.264	1.756	2.461	3.351	4.448	5.751	7.245	8.942	10.844
R4SAM	0.921	1.017	1.316	1.817	2.519	3.418	4.510	5.807	7.306	9.006	10.900
R4BM	0.825	0.924	1.218	1.718	2.418	3.316	4.412	5.718	7.214	8.907	10.804
R8BM	0.906	1.007	1.303	1.800	2.497	3.394	4.493	5.793	7.288	8.984	10.879

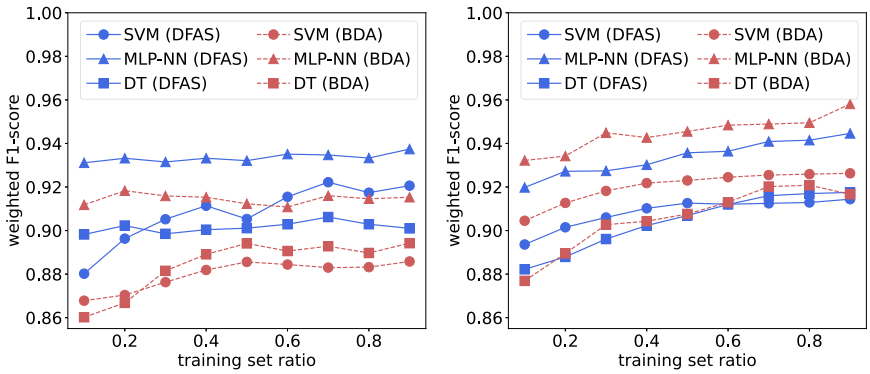
model always outperforms the other two models. Even trained with only 9,574 data, the MLP-NN-based approach can achieve a high weighted F1-score, which is about 0.92 for both adder and multiplier. This means that the proposed approach for selecting adder/multiplier architecture is not sensitive to the size of training subset.

Table 10. Average Area and Critical Path Delay of the Synthesized Components (Both Bitwidths Not Less Than 32) with Different Approaches

Dataset 1	Metric		Dataset 2	Metric	
	Area (μm^2)	CPD (ns)		Area (μm^2)	CPD (ns)
Optimal	401.1	1.622	Optimal	5,710.7	6.208
Proposed	403.1	1.624	Proposed	5,717.6	6.210
DesignWare	407.2	1.628	DesignWare	5,898.4	6.226
CLA	509.9	1.681	SAM	7,009.1	6.261
SA	476.1	1.728	R4SAM	7,549.8	6.323
BKA	464.1	1.663	R4BM	6,182.9	6.228
KSA	637.9	1.678	R8BM	6,275.8	6.307

Table 11. Average Area and Critical Path Delay of the Synthesized Components (at Least One Bitwidth Below 32) with Different Approaches

Dataset 1	Metric		Dataset 2	Metric	
	Area (μm^2)	CPD (ns)		Area (μm^2)	CPD (ns)
Optimal	257.8	0.965	Optimal	3,845.4	3.598
Proposed	258.5	0.965	Proposed	3,850.2	3.602
DesignWare	263.7	0.968	DesignWare	4,109.2	3.612
CLA	364.8	1.025	SAM	5,497.5	3.659
SA	343.6	1.071	R4SAM	5,833.6	3.717
BKA	332.0	1.004	R4BM	4,201.2	3.621
KSA	505.8	1.018	R8BM	4,362.8	3.700



(a) Adder (b) Multiplier

Fig. 9. The weighted F1-scores of testing with different ML classifiers for architectural selection, which are trained with varied-size training sets.

5 CONCLUSIONS

In this work, a machine-learning-driven approach is proposed to automatically select the architecture of the adder or multiplier components for achieving a good performance of the synthesized circuit. The approach is realized by transforming the problem into a multi-class classification problem with several architecture candidates and a run-only-once procedure of data preparation and model training. The experiments with different bit widths of operands and clock periods demonstrate that the MLP-NN-based classifier stably achieves better performance than the classifiers based on SVM and DT. It can accurately predict (with up to 94% accuracy and 0.9489 weighted

F1-score) the best architecture of adder or multiplier for a given set of specifications/constraints. Compared with a brute-force approach trying four candidate architectures, the proposed approach costs only one-fourth of runtime and sacrifices little on the performance of the synthesized circuit. Furthermore, the adder (multiplier) generated with the proposed approach achieves performance metrics close to the optimal and has 1.6% (5.2%) less area and 2.2% (7.1%) more worst negative slack averagely than that generated with the DesignWare datapath generator. And, the proposed approach is insensitive to the size of training subset.

In the future, we will apply the proposed approach to the datapath design in larger and more realistic circuits. Considering the adaptability of machine learning libraries in a GPU computing environment, the proposed approach could show more runtime and performance advantages on the GPU server, over the existing approach for architecture selection.

REFERENCES

[1] Charles R. Baugh and Bruce A. Wooley. 1973. A two's complement parallel array multiplication algorithm. *IEEE Trans. Comput.* C-22, 12 (1973), 1045–1047. DOI: https://doi.org/10.1109/T-C.1973.223648

[2] Andrew D. Booth. 1951. A signed binary multiplication technique. *Quart. J. Mech. Appl. Math.* 4, 2 (1951), 236–240. DOI: https://doi.org/10.1093/qjmam/4.2.236

[3] Leo Breiman, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone. 1984. *Classification and Regression Trees.* Chapman & Hall, Boca Raton, FL.

[4] Richard P. Brent and Hsiang T. Kung. 1982. A regular layout for parallel adders. *IEEE Trans. Comput.* C-31, 3 (1982), 260–264. DOI: https://doi.org/10.1109/TC.1982.1675982

[5] Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. *Mach. Learn.* 20, 3 (1995), 273–297. DOI: https://doi.org/10.1007/BF00994018

[6] Ganqu Cui, Wenjian Yu, Xin Li, Zhiyu Zeng, and Ben Gu. 2019. Machine-learning-driven matrix ordering for power grid analysis. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'19).* 984–987. DOI: https://doi.org/10.23919/DATE.2019.8715086

[7] Luigi Dadda. 1965. Some schemes for parallel multipliers. *Alta Frequenza* 34 (1965), 349–356.

[8] Brunno A. de Abreu, Augusto Berndt, Isac S. Campos, Cristina Meinhardt, Jonata T. Carvalho, Mateus Grellert, and Sergio Bampi. 2021. Fast logic optimization using decision trees. In *Proceedings of the International Symposium on Circuits and Systems (ISCAS'21).* 1–5. DOI: https://doi.org/10.1109/ISCAS51556.2021.9401664

[9] Hao Geng, Yuzhe Ma, Qi Xu, Jin Miao, Subhendu Roy, and Bei Yu. 2021. High-speed adder design space exploration via graph neural processes. *IEEE Trans. Comput.-aid. Des. Integ. Circ. Syst.* (2021). DOI: https://doi.org/10.1109/TCAD.2021.3114262

[10] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the International Conference on Machine Learning (ICML'15).* 448–456.

[11] Marellasv Kamaraju, Kondepudi Lal Kishore, and A. V. N. Tilak. 2010. Power optimized ALU for efficient datapath. *Int. J. Comput. Applic.* 11 (2010), 39–43.

[12] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. DOI: https://doi.org/10.48550/arXiv.1412.6980

[13] Peter M. Kogge and Harold S. Stone. 1973. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Trans. Comput.* C-22, 8 (1973), 786–793. DOI: https://doi.org/10.1109/TC.1973.5009159

[14] Jianhua Liu, Shuo Zhou, Haikun Zhu, and Chung-Kuan Cheng. 2003. An algorithmic approach for generic parallel adders. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'03).* 734–740. DOI: https://doi.org/10.1109/ICCAD.2003.159758

[15] Jianhua Liu, Yi Zhu, Haikun Zhu, Chung-Kuan Cheng, and John Lillis. 2007. Optimum prefix adders in a comprehensive area, timing and power design space. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC'07).* 609–615. DOI: https://doi.org/10.1109/ASPDAC.2007.358053

[16] Yuzhe Ma, Subhendu Roy, Jin Miao, Jiamin Chen, and Bei Yu. 2019. Cross-layer optimization for high speed adders: A Pareto driven machine learning approach. *IEEE Trans. Comput.-aid. Des. Integ. Circ. Syst.* 38, 12 (2019), 2298–2311. DOI: https://doi.org/10.1109/TCAD.2018.2878129

[17] Taeko Matsunaga and Yusuke Matsunaga. 2007. Area minimization algorithm for parallel prefix adders under bitwise delay constraints. In *Proceedings of the Great Lakes Symposium on VLSI (GLSVLSI'07).* 435–440. DOI: https://doi.org/10.1145/1228784.1228886

[18] Yu-Ting Pai and Yu-Kumg Chen. 2004. The fastest carry lookahead adder. In *Proceedings of the International Workshop on Electronic Design, Test and Applications (DELTA'04).* 434–436. DOI: https://doi.org/10.1109/DELTA.2004.10071

[19] Behrooz Parhami. 2010. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, New York, NY.
[20] Pytorch. 2022. PyTorch. Retrieved from https://pytorch.org/.
[21] John R. Quinlan. 1986. Induction of decision trees. *Mach. Learn.* 1 (1986), 81–106. DOI: https://doi.org/10.1007/BF00116251
[22] John R. Quinlan. 1993. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA.
[23] Shubham Rai, Walter Lau Neto, Yukio Miyasaka et al. 2021. Logic synthesis meets machine learning: Trading exactness for generalization. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'21)*. 1026–1031. DOI: https://doi.org/10.23919/DATE51398.2021.9473972
[24] Subhendu Roy, Mihir Choudhury, Ruchir Puri, and David Z. Pan. 2016. Polynomial time algorithm for area and power efficient adder synthesis in high-performance designs. *IEEE Trans. Comput.-aid. Des. Integ. Circ. Syst.* 35, 5 (2016), 820–831. DOI: https://doi.org/10.1109/TCAD.2015.2481794
[25] Subhendu Roy, Yuzhe Ma, Jin Miao, and Bei Yu. 2017. A learning bridge from architectural synthesis to physical design for exploring power efficient high-performance adders. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'17)*. 1–6. DOI: https://doi.org/10.1109/ISLPED.2017.8009168
[26] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. 1985. *Learning Internal Representations by Error Propagation*. Technical Report. California University.
[27] Scikit-Learn. 2022. Scikit-learn: Machine Learning in Python. Retrieved from https://scikit-learn.org/stable/.
[28] Silicon Integration Initiative, Inc. 2022. Open-Cell Library - Silicon Integration Initiative. Retrieved from https://si2.org/open-cell-library/.
[29] Jack Sklansky. 1960. Conditional-sum addition logic. *IRE Trans. Electron. Comput.* EC-9, 2 (1960), 226–231. DOI: https://doi.org/10.1109/TEC.1960.5219822
[30] Synopsys, Inc. 2022. Design Compiler NXT. Retrieved from https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/design-compiler-nxt.html.
[31] Christopher S. Wallace. 1964. A suggestion for a fast multiplier. *IEEE Trans. Electron. Comput.* EC-13, 1 (1964), 14–17. DOI: https://doi.org/10.1109/PGEC.1964.263830
[32] A. Weinberger and J. L. Smith. 1958. A logic for high-speed addition. *Nat. Bur. Stand. Circul.* 591 (1958), 3–12.
[33] Dingcheng Yang, Wenjian Yu, Yuanbo Guo, and Wenjie Liang. 2021. CNN-Cap: Effective convolutional neural network based capacitance models for full-chip parasitic extraction. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD'21)*. IEEE, 1–9.
[34] Yiming Yang. 1999. An evaluation of statistical approaches to text categorization. *Inf. Retr.* 1 (1999), 69–90.
[35] Yi Zhu, Jianhua Liu, Haikun Zhu, and Chung-Kuan Cheng. 2008. Timing-power optimization for mixed-radix Ling adders by integer linear programming. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC'08)*. 131–137. DOI: https://doi.org/10.1109/ASPDAC.2008.4483926
[36] Reto Zimmermann. 1996. Non-heuristic optimization and synthesis of parallel-prefix adders. In *Proceedings of the International Workshop on Logic and Architecture Synthesis (IWLAS'96)*.

Received 18 March 2022; revised 13 July 2022; accepted 19 August 2022