

Cross-layer Optimization for High Speed Adders: A Pareto Driven Machine Learning Approach

Yuzhe Ma, Subhendu Roy, Jin Miao, Jiamin Chen, and Bei Yu

Abstract—In spite of maturity to the modern electronic design automation (EDA) tools, optimized designs at architectural stage may become sub-optimal after going through physical design flow. Adder design has been such a long studied fundamental problem in VLSI industry yet designers cannot achieve optimal solutions by running EDA tools on the set of available prefix adder architectures. In this paper, we enhance a state-of-the-art prefix adder synthesis algorithm to obtain a much wider solution space in architectural domain. On top of that, a machine learning-based design space exploration methodology is applied to predict the Pareto frontier of the adders in physical domain, which is infeasible by exhaustively running EDA tools for innumerable architectural solutions. Considering the high cost of obtaining the true values for learning, an active learning algorithm is proposed to select the representative data during learning process, which uses less labeled data while achieving better quality of Pareto frontier. Experimental results demonstrate that our framework can achieve Pareto frontier of high quality over a wide design space, bridging the gap between architectural and physical designs. Source code and data are available at <https://github.com/yuzhe630/adder-DSE>.

I. INTRODUCTION

In the last decades, the industrial EDA tools have advanced towards optimality, especially at the individual stages of VLSI design cycle. Nevertheless, with growing design complexity and aggressive technology scaling, physical design issues have become more and more complex. As a result, the constraints and the objectives of higher layers, such as the system or logic level, are very difficult to be mapped into those of lower layers, such as physical design, and vice-versa, thereby creating a *gap* between the optimality at the logic stage and the physical design stage. This necessitates the innovation of data-driven methodologies, such as machine learning [1]–[5], to bridge this gap.

Adder design is one of the fundamental problems in digital semiconductor industry, and its main bottleneck (in terms of both delay and area) is the carry-propagation unit. This unit can be realized by hundreds of thousands of parallel prefix structures, but it is hard to evaluate the final metrics without running through physical design tools. Historically, regular adders [6]–[9] have been proposed for achieving the corner points in terms of various metrics as shown in Fig. 1

The preliminary version has been presented at the IEEE International Symposium on Low Power Electronics and Design (ISLPED) in 2017. This work is supported in part by The Research Grants Council of Hong Kong SAR (Project No. CUHK24209017) and CUHK Undergraduate Summer Research Internship 2017.

Y. Ma, J. Chen and B. Yu are with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, NT, Hong Kong.

S. Roy is with Intel Corporation, San Jose, CA, USA.

J. Miao is with Cadence Design Systems, San Jose, CA, USA.

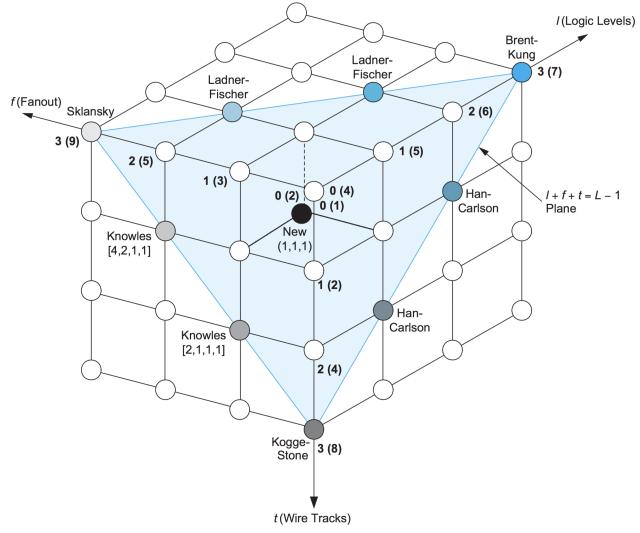


Fig. 1: Regular adders (picture taken from [12]).

in architectural stage. The main motivation for structural regularity was the ease of manual layout, but EDA tools now taking care of all physical design aspects, the regularity is no longer essential. Moreover, the extreme corners do not map well to the physical design metrics after synthesis, placement and routing. To address this gap between prefix adder synthesis and actual physical design of the adders, custom adders are typically designed by tuning parameters, such as gate-sizing, buffering etc., targeting at the optimization of power/performance metrics for a specific technology library [10], [11]. However, this custom approach (i) needs significant engineering effort, (ii) is not flexible to Engineering Change Order (ECO), and (iii) does not guarantee the optimality.

The algorithmic synthesis approach resolves the first two issues of the custom approach, by adding more flexibility to the late ECO changes and reducing the engineering effort. Based on the number of solutions, the existing adder synthesis algorithms can be broadly classified into two categories. The first and the most common approach is to generate a single prefix network for a set of structural constraints, such as the logic level, fan-out etc. Several algorithms have been proposed to minimize the size of the prefix graph (s) under given bit-width (n) and logic-level (L) constraints [13]–[16]. Closed form theoretical bounds for size-optimality are provided by [17] for $L \geq 2 \log_2 n - 2$. [18] has given more general bound for prefix graph size, but when L is reduced to $\log_2 n$, a prerequisite for high-performance adders, there is no closed form bound for s . [19] presents a polynomial-time algorithm for

generating prefix graph structures by restricting both logic-level and fan-out. The limitations in these approaches are two-fold, (i) this restricted set of structures is not capable of exploring the large solution space, and (ii) since it is very hard to analytically model the physical design complexities, such as wire-length and congestion issues, the physical design metrics, such as the area, power, delay etc., may not be mapped well to the prefix structure metrics, such as the size, max-fan-out (*mfo*) etc. This motivates the second category of algorithms where thousands of prefix adder solutions can be generated and explored for synthesis and physical design in the commercial EDA tools.

One such approach is [20] which presents an exhaustive bottom-up enumeration technique with several pruning strategies to generate innumerable prefix structure solutions. However, it has two issues, (i) this approach cannot provide solutions in several cases for restricted fan-out, which can control the congestion and load-distribution during physical design [19]. As a result, it may still miss the *good* solution space to a large extent, and (ii) it is computationally very intensive to run all solutions through synthesis, placement and routing.

In this paper, we enhance the algorithm in [20], [21] to generate adders under any arbitrary *mfo* constraint, which enables a *wider* adder solution space in *logical* form. To tackle the high computational effort during the physical design flow, we further propose to use machine learning to perform the design space exploration in *physical* solution space. We develop Pareto frontier driven machine learning methodologies to achieve rich adder solutions with trade-offs among power, area, and delay. As a passive supervised learning, the proposed *quasi-random* sampling approach is able to select representative prefix adders out of the hundreds of thousands of prefix structures.

It should be noted that various machine learning algorithms have been investigated to explore design space in different design scenarios. Palermo *et al.* [22] deploy both linear regression and artificial neural network for multiprocessor systems-on-chip design. Lin *et al.* [23] present a random forest-based learning model in high level synthesis, which can find an approximate Pareto-optimal designs effectively. Meng *et al.* [24] propose a random forest-based method for Pareto frontier exploration, where non-Pareto-optimal designs are carefully eliminated through an adaptive strategy. Multiple predictions can be obtained through random forest, which can be used for estimating the uncertainty. Superior to the random forest, in this paper we further propose an active learning approach based on Gaussian Process (GP), which by nature can estimate the prediction uncertainty efficiently.

Our main contributions are summarized as follows:

- A comprehensive framework for optimal adder search by machine learning methodology bridging the prefix architecture synthesis to the final physical design;
- An enhancement to a state-of-the-art prefix adder algorithm [20] to optimize the prefix graph size for restricted fan-out and explore a wider solution space;
- A machine learning model for prefix adders, guided by quasi-random data sampling with features considering

architectural attributes and EDA tool settings;

- A design space exploration method to generate the Pareto frontier for delay vs. power/area over a wide design space;
- An active learning approach for the design space exploration, which uses less labeled data and achieves better quality of Pareto frontier.

The rest of the paper is organized as follows. Section II presents the background of prefix adder synthesis, while Section III discusses our prefix graph generation algorithm. Next, two machine learning approaches of design space exploration for high-performance adders are described. Section IV presents the passive supervised learning, while Section V introduces a Pareto frontier driven active learning approach. Section VI lists the experimental results, followed by conclusion in Section VII.

II. PREFIX ADDER SYNTHESIS

In this section, we first provide the background of the prefix adder synthesis problem. Then we present a brief discussion on the algorithm presented in [20], which we enhance to our **Prefix Graph Generation (PGG)** algorithm to synthesize the prefix adder network.

A. Preliminaries

An n bit adder accepts two n bit addends $A = a_{n-1}..a_1a_0$ and $B = b_{n-1}..b_1b_0$ as input, and computes the output sum $S = s_{n-1}..s_1s_0$ and carry out $C_{out} = c_{n-1}$, where $s_i = a_i \oplus b_i \oplus c_{i-1}$ and $c_i = a_i b_i + a_i c_{i-1} + b_i c_{i-1}$. The simplest realization for the adder network is the ripple-carry-adder, but with logic level $n - 1$, which is too slow. For faster implementation, carry-lookahead principle is used to compute the carry bits. Mathematically, this can be represented with bitwise (group) generate function $g(G)$ and propagate function $p(P)$ by the Weinberger's recurrence equations as follows [25]:

- Pre-processing (inputs): Bitwise generation of g, p

$$g_i = a_i \cdot b_i \text{ and } p_i = a_i \oplus b_i. \quad (1)$$

- Prefix processing: This part is the main carry-propagation component where the concept of generate/propagate is extended to multiple bits and $G_{[i:j]}$, $P_{[i:j]}$ ($i \geq j$) are defined as

$$P_{[i:j]} = \begin{cases} p_i, & \text{if } i = j, \\ P_{[i:k]} \cdot P_{[k-1:j]}, & \text{otherwise,} \end{cases} \quad (2)$$

$$G_{[i:j]} = \begin{cases} g_i, & \text{if } i = j, \\ G_{[i:k]} + P_{[i:k]} \cdot G_{[k-1:j]}, & \text{otherwise.} \end{cases} \quad (3)$$

The associative operation \circ is defined for (G, P) as:

$$\begin{aligned} (G, P)_{[i:j]} &= (G, P)_{[i:k]} \circ (G, P)_{[k-1:j]} \\ &= (G_{[i:k]} + P_{[i:k]} \cdot G_{[k-1:j]}, P_{[i:k]} \cdot P_{[k-1:j]}). \end{aligned} \quad (4)$$

- Post-processing (outputs): Sum/Carry-out generation

$$s_i = p_i \oplus c_{i-1}, \quad c_i = G_{[i:0]}, \text{ and } C_{out} = c_{n-1}. \quad (5)$$

The ‘Prefix processing’ or carry propagation network can be mapped to a prefix graph problem with inputs $i_k = (p_k, g_k)$ and outputs $o_k = c_k$, such that o_k depends on all previous inputs i_j ($j \leq k$). Any node except the input nodes is called a *prefix* node. Size of the prefix graph is defined as the number of prefix nodes in the graph. Fig. 2 shows an example of such prefix graph of 6 bit and we can see that $C_{out} = c_5 = o_5$ is given by

$$o_5 = (i_5 \circ i_4) \circ ((i_3 \circ i_2) \circ (i_1 \circ i_0)). \quad (6)$$

Size (s), logic level (L) and maximum-fan-out (mfo) for this network are respectively 8, 3 and 2. Note that here the number of fan-ins for each of the associative operation o is two, thus this is called radix-2 implementation of the prefix graph. However, there exist other options such as radix-3 or radix-4, but the complexity is very high and not beneficial in static CMOS circuits [26]. In this work, the logic levels for all output bits are $\log_2 n$, *i.e.*, the minimum possible, to target high performance adders.

B. Discussion on [20]

Our PGG algorithm to generate the prefix graph structures for physical solution space exploration is based on [20]. So it is imperative to first discuss about [20]. However, we omit the details and only mention the key points of [20] due to space constraint.

[20] is an exhaustive bottom-up and pruning based enumeration technique for prefix adder synthesis. This work presented an algorithm to generate all possible $n + 1$ bit prefix graph structures from any n bit prefix graph. Then this algorithm is employed in a bottom-up fashion (from 1 bit adder to 2 bit adders, then from all 2 bit adders to 3 bit adders, and so on) to synthesize prefix graphs of any bit-width. As a result, scalability issue arises due to the exhaustive nature of the algorithm, which is then tackled by adopting various pruning strategies to scale the approach. However, the pruning strategies are not sufficient to scale the algorithm well for different fan-out constraints. So when it intends to find the solutions for higher bit adders, the intermediate adder solutions that need to be generated are often huge. Consequently, it fails to get fan-out restricted (e.g. when $mfo = 8, 10, 12$ etc. for 64 bit adders) solutions even with 72GB RAM due to the generation of innumerable intermediate solutions [19]. Pruning strategies, such as size-bucketing [20], help to achieve solutions in some cases, but with sub-optimality. So design space-exploration based on this algorithm can miss a significant spectrum of the adder solutions.

C. Our PGG Algorithm

To better explore the wide design space of adders, in this paper we have enhanced [20] for different fan-out constraints by incorporating more pruning techniques.

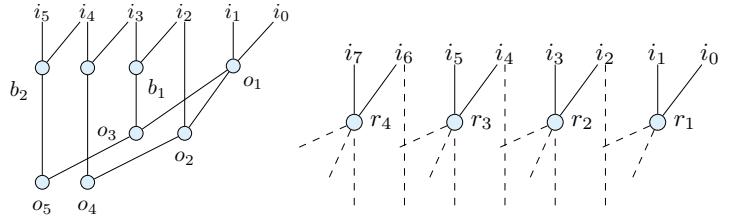


Fig. 2: 6 bit prefix adder

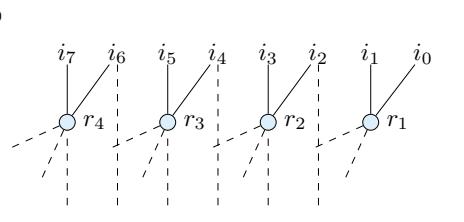


Fig. 3: Imposing semi-regularity.

1) Semi-Regularities in Prefix Graph Structure: The first strategy is to enforce a sort of regularity in the prefix graphs. For instance, regular adders, such as Sklansky, Brent-Kung, have the inherent property that the consecutive input nodes (even and odd) are combined to create the prefix nodes at the logic level 1. In our approach, we constrain this regularity for those prefix nodes (logic level 1). To explain this, let us consider Fig. 3. We can see that prefix nodes r_1, r_2, r_3 and r_4 are constructed by consecutive even-odd nodes. For instance, i_0 and i_1 are used to construct r_1 . But with this structural constraint, we are not allowed to construct any node by combining i_1 and i_2 as done in Kogge-Stone adders. Note that the sub-structure, as shown in Fig. 3, is a part of some regular adders like Sklansky adder, and is imposed in our prefix structure enumeration.

We have run experiments with 16 bit adders, and observed that this pruning strategy (i) does not degrade the solution quality (or size of the prefix graph under same L and mfo), but (ii) able to reduce the search space significantly, in comparison to not using this pruning strategy.

2) Level Restriction in Non-trivial Fan-in: Each of the prefix node $N(a:b)$, where a is the most-significant-bit (MSB) and b is the least-significant-bit (LSB), is constructed by connecting the trivial fan-in $N_{tr}(a:c)$ having same MSB as N , and the non-trivial fan-in $N_{non-tr}(c-1:b)$. For instance, in Fig. 2, o_3 and b_2 are respectively the non-trivial fan-in and the trivial fan-in node for the prefix node o_5 . In the bottom-up enumeration technique, we put another additional restriction that the level of the trivial fan-in node is always less or equal to that of the non-trivial fan-in node, *i.e.*, $\text{level}(N_{tr}) \leq \text{level}(N_{non-tr})$. Note that this sort of structural restriction is also inherent in regular adders, such as Sklansky or Brent-Kung adders.

In a nutshell, our PGG algorithm is a blend of regular adders and [20]. We borrow some properties of regular adders to enforce in [20] for reducing its huge search space without hampering the solution quality. To illustrate this, we have obtained the binary for [20] from the authors, and first compared our result for lower bit adders, such as $n = 16, 32$. We got the solutions with same minimum size, which *proves that our structural constraints have not degraded the solution quality*. However, for higher bit adders, we get better solution quality than [20] as shown in TABLE I.

Column 1 presents the mfo constraint, while columns 2 and 3 respectively show the size and run-time for our enhanced algorithm, and the corresponding entries for [20] are respectively represented in columns 4 and 5. In general,

TABLE I: Comparison with [20] for 64 bit adders

mfo	Our Approach		Approach in [20]	
	size	Run-time (s)	size	Run-time (s)
4	244	302	252	241
6	233	264	238	212
8	222	423	-	-
12	201	193	-	-
16	191	73	192	149
32	185	0.04	185	0.04

when fan-out is relaxed or mfo is higher, the run-time is less due to relaxed size-pruning as explained in [20]. Note that [20] cannot generate solutions for $mfo = 8, 12$ due to generation of innumerable intermediate solutions as explained in [19]. On the contrary, our structural constraints can do a pre-filtering of the potentially futile solutions, thereby allowing relaxed size-pruning and size-bucketing to search for more effective solution space. In terms of run-time, it is slightly worse in a few cases, but importantly, this generation is a one-time process, and this run-time is negligible in comparison to the design space exploration by the physical design tools. So our imposed structural restrictions (i) do not degrade the solution quality, (ii) achieve better solution sizes for all mfo than [20] for higher bit adders which could not even generate solutions in all cases, and (iii) help to obtain wider physical solution space to be demonstrated in Section II-E.

D. Quasi-random Sampling

We have mainly focused on 64 bit adders in this work as this is mostly used in today's microprocessors. From all prefix adder solutions, we sample a set of solutions for building the learning model via the quasi-random approach which is conducted by a two-level binning (mfo, s) followed by random selection. This approach aims to evenly sample the prefix adders covering different architectural bins. The primary level of binning is determined by mfo of the solutions. However, there may be thousands of architectures sharing the same mfo , so the secondary level of binning is based on s . Afterwards, adders are picked randomly from those secondary bins.

We illustrate the quasi-random sampling with the following example: given 5000 solutions with $mfo = 4$, we want to pick 50 solutions from them. Suppose these 5000 solutions have the size distribution from 244 to 258. First a random solution is picked from the bucket of the solutions ($mfo = 4, s = 244$). Then we pick a solution randomly from ($mfo = 4, s = 245$), and so on. After picking 15 solutions from each of those buckets with $mfo = 4$, we again start from the bucket ($mfo = 4, s = 244$). This process is repeated until we get 50 solutions. Similar procedure is done with other mfo values.

E. Physical Solution Space Comparison with [20]

In this subsection, we show the usefulness of our algorithm for obtaining wider solution space in physical design domain in comparison to [20]. Among the prefix adders generated by [20], we randomly sampled 7000 prefix adders. Those prefix adders are fed into the full EDA flow (synthesis, placement and routing) to get their real delay, power and area values

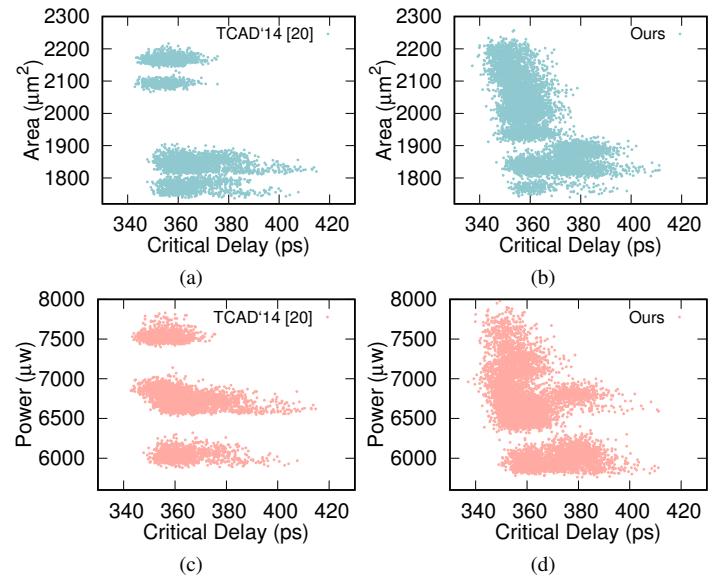


Fig. 4: Quasi-random sampled adders vs. adders from [20]. (a) Solution space in area vs. delay domain from [20]; (b) Solution space in area vs. delay domain from ours; (c) Solution space in power vs. delay domain from [20]; (d) Solution space in power vs. delay domain from ours.

(takes around 700 hours). We plot these adders by [20] and our representative 3000 adders in Fig. 4. It can be seen that, although the numbers of adders by [20] is more than 2 times of our representative adders, our adders still cover *wider* solution space in physical domain, demonstrating the effectiveness of our enhanced algorithm PGG. This is in accordance with the solutions missed by [20] as mentioned in TABLE I. Those availabilities eventually offer more opportunities for our machine learning methodology to identify close to ground truth Pareto frontier solutions.

III. BRIDGING ARCHITECTURAL SOLUTION SPACE TO PHYSICAL SOLUTION SPACE

In most EDA problems, the metrics of the solution quality are typically conflicting. For instance, if we optimize the timing of the design, then the power/area may be compromised and vice versa. So one imperative job of EDA engineers is to find the Pareto-optimal points of the design enabling the designers to select among those. In this section, we first provide the preliminaries about Pareto optimality, and the error metrics of Pareto optimal solutions. Then we discuss the gap between the prefix architectural solution space and physical solution space in adders, which motivates the need of the machine learning-based approach for optimal adder exploration. Finally, a domain knowledge-based feature selection details are presented along with training data sampling for the learning models.

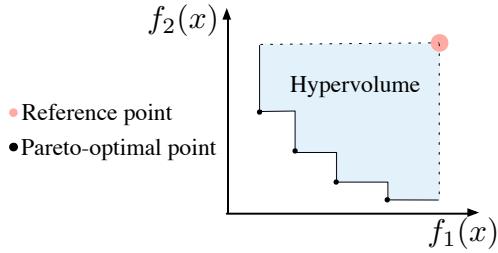


Fig. 5: Hypervolume with two objectives in objective space.

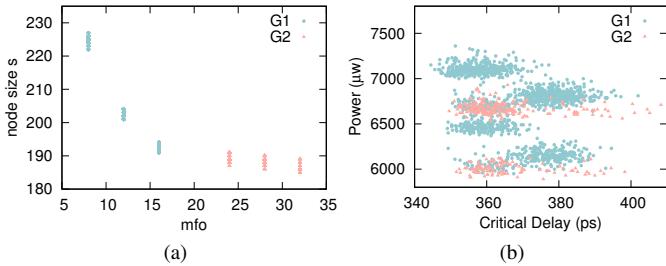


Fig. 6: Gap between prefix structure and physical design of adders: (a) Architectural solution space; (b) Physical solution space.

A. Preliminaries

Definition 1 (Pareto Optimality). *An objective vector $\mathbf{f}(\mathbf{x})$ is said to dominate $\mathbf{f}(\mathbf{x}')$ if:*

$$\begin{aligned} \forall i \in [1, n], f_i(\mathbf{x}) &\leq f_i(\mathbf{x}') \\ \text{and } \exists j \in [1, n], f_j(\mathbf{x}) &< f_j(\mathbf{x}'). \end{aligned} \quad (7)$$

A point \mathbf{x} is *Pareto-optimal* if there is no other \mathbf{x}' in design space such that $\mathbf{f}(\mathbf{x}')$ dominates $\mathbf{f}(\mathbf{x})$.

As in this paper for adder design, a Pareto-optimal design is where none of the objective metrics, such as area, power or delay, can be improved without worsening at least one of the others. The *Pareto Frontier* is the set of all the Pareto-optimal designs in the *objective space*. Therefore, the goal is to identify the Pareto-optimal set P for all the Pareto-optimal designs.

Definition 2 (Hypervolume). *The hypervolume computes the volume enclosed by the Pareto frontier and the reference point in the objective space [27].*

In Fig. 5, the shaded area is an example of the hypervolume of a Pareto set with two objectives. Then the *hypervolume error* for a predicted Pareto set \hat{P} is defined as

$$\eta = \frac{V(P) - V(\hat{P})}{V(P)}, \quad (8)$$

where P is the true Pareto-optimal set, and $V(P)$ is the hypervolume of the Pareto set P . Note that a prediction \hat{P} which contains the whole design space has an error of 0. Thus the predicted set \hat{P} with less points is desired.

B. Gap Between Logic and Physical Design

Since we focus on high performance adders and explore the prefix adders of logic level $L = \log_2 n$, the metrics at this

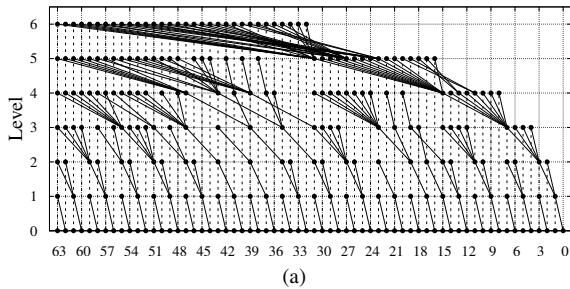
architecture stage are prefix node size s and max fan-out mfo . These two metrics are conflicting, *i.e.*, if we reduce mfo , s increases and vice-versa. Similar competing relationship exists between delay and power/area after physical design. It should be stressed that power and s are correlated, and mfo indirectly controls the timing as more restricted fan-out can mitigate congestion and load-distribution, thereby improving the delay of the adder. However, this relationship between architectural synthesis and physical design is approximate, and not a very high-fidelity one.

To demonstrate this, we plot node size s vs. mfo and power vs. delay in Fig. 6 for several 64 bit adder solutions. In this experiment, we have generated the prefix architecture solutions by PGG, and the final power/delay numbers are obtained by running those solutions through EDA tools as explained later in Section VI. An example of the prefix architecture and the corresponding physical solution is presented in Fig. 7. In Fig. 6(a), we broadly categorize the solutions into 2 groups, (i) G_1 with higher node size and lower mfo , and (ii) G_2 with lower node size and higher mfo . In Fig. 6(b), the same designs as Fig. 6(a) are projected into the physical solution space, restoring the group information. Design Compiler [28] (version F-2011.09-SP3) is used for logical synthesis, and IC Compiler [29] (version J-2014.09-SP5-3) is used for the placement and routing. Non linear delay model (NLDM) in 32nm SAED cell-library [30] is used for technology mapping. The key observations here are firstly, *there is a correlation between architectural solution space and physical design solution space*. For instance, the solutions from G_1 are mostly on the upper side, and those of G_2 are mostly on the lower side in Fig. 6(b), thereby indicating a correspondence between s and power. Nevertheless, *it is not completely reliable*. For example, (i) the delay numbers for G_1 and G_2 are very much spread, (ii) a cluster can be observed where the solutions from G_1 and G_2 are mixed up in Fig. 6(b), and (iii) several solutions of G_1 are better than several solutions of G_2 in power, which is not in accordance with the metrics at the prefix adder architecture stage. So we can not utterly rely on architectural solution space to achieve the optimal output in physical solution space.

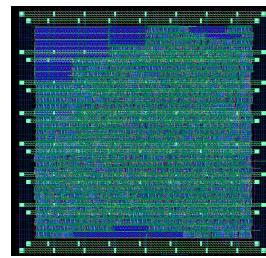
However, since our algorithm generates **hundreds of thousands** of prefix graph structures, it is intractable to run synthesis and physical design flows for even a small percentage of all available prefix adder architectures. To address this **fidelity gap** between the two design stages and the high computational cost together, we come up with a novel machine learning guided design space exploration as replacement of exhaustive search.

C. Feature Selection

The feature is a representation which is extracted from the original input representation, and it plays an important role in machine learning tasks. We now discuss the features to be used for the learning model. Features are considered from both prefix adder structure and tool settings, with a focus on the former. We select node size and maximum-fan-out (mfo) of a prefix adder as two main features for our learning model.



(a)



(b)

Fig. 7: (a) An example of architectural solution: Bit-width = 64, size = 201, Max. level = 6, Max. fanout = 12; (b) Corresponding physical solution.

However, for any given mfo and node size, there will be hundreds or even thousands of different prefix architectures. Therefore, additional features are required to better distinguish individual prefix adder attributes. We define a parameter sum-path-fan-out ($spfo$) for this. Let a and b are the fan-in nodes of a node n , then $spfo(n)$ is defined recursively as:

$$spfo(n) = \begin{cases} 0, & \text{if } n \in \text{input}, \\ \text{sum}(fo(a) + spfo(a), fo(b) + spfo(b)), & \text{otherwise.} \end{cases} \quad (9)$$

Here $fo(n)$ denotes the fan-out of any node n . Consider the prefix adder structure in Fig. 8, and according to the definition we have:

$$\begin{aligned} spfo(o_1) &= \text{sum}(fo(i_0) + spfo(i_0), fo(i_1) + spfo(i_1)) \\ &= \text{sum}(1, 1) = 2, \\ spfo(b_1) &= \text{sum}(fo(i_2) + spfo(i_2), fo(i_3) + spfo(i_3)) \\ &= \text{sum}(2, 1) = 3, \\ spfo(b_2) &= \text{sum}(fo(i_4) + spfo(i_4), fo(i_5) + spfo(i_5)) \\ &= \text{sum}(2, 1) = 3. \end{aligned}$$

Therefore, we can use the recursive definition to calculate

$$\begin{aligned} spfo(o_3) &= \text{sum}(fo(o_1) + spfo(o_1), fo(b_1) + spfo(b_1)) \\ &= \text{sum}(3 + 2, 2 + 3) = 10, \\ spfo(o_5) &= \text{sum}(fo(o_3) + spfo(o_3), fo(b_2) + spfo(b_2)) \\ &= \text{sum}(3 + 10, 3 + 3) = 19. \end{aligned}$$

In our methodology, we use the $spfo$ of the output nodes which are at $\log_2 n$ level (there are 32 nodes at level 6 for 64 bit adder) as the features to characterize the prefix structures, in addition to mfo , size and target delay. The basic intuition for selecting $spfo$ of the output nodes as the features is that the critical path delay of the adder is the longest path delay from input to output. So it depends on the (i) path-lengths, which can be represented at the prefix graph stage by the logic level of the node, and (ii) the number of fan-outs driven at every node on the path. Note that we have skipped the $spfo$ of the output nodes which are not at $\log_2 n$ level as for those nodes, the path length is smaller, and those would not potentially dictate the critical path delay.

Apart from these prefix graph structural features, we also consider tool settings from synthesis stage and physical design stage as other features. We have synthesized the adder structures using industry-standard EDA synthesis tool [28], where

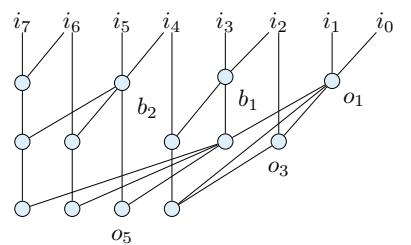


Fig. 8: Defining $spfo$ of a node.

we can specify the target-delay for the adder. The tool then adopts different strategies internally to meet that target-delay which we can hardly take into account during prefix graph synthesis. Consequently, changing the target-delay can lead to different power/timing/area metrics. So we have considered target-delay as a feature in our learning approach.

In physical design, utilization is an important parameter, which defines the area occupied by standard cell, macros and blockages. Different utilization values can lead to different layouts after physical design. Therefore, we take utilization as another feature in the learning model.

In addition to the target delay and utilization, other tool settings have also been explored. The optimization level setting in logical synthesis has a potential impact on the performance of adders, which can be adjusted by `compile` and `compile_ultra` commands with different options. After synthesizing, it is observed that the solutions generated with `compile_ultra` can significantly dominate the solutions generated by `compile`. Therefore, this setting is fixed to `compile_ultra` level as we are aiming at superior designs.

In this work, the technology node is not used as a feature. From the machine learning perspective, there is a common assumption for conventional machine learning applications that the training and test data are drawn from the same feature space and the same distribution [31]. The values of area/power/delay may vary a lot under different technology nodes, which results in different underlying data distributions. Therefore, the technology node for synthesis should be consistent. The proposed approach for feature extraction can also be applied to other technology nodes as long as the technology node is consistent during the design flow. If the technology node of the testing data switches to another one, the machine learning model should be re-trained using the data from that technology node to ensure the accuracy of the model.

D. Data Sampling

Since we can not afford to run the physical design flow for too many architectures, and too few training data may degrade the model accuracy significantly, a set of adders need to be selected to represent the entire design solution space. However, finding a succinct set of representative training data for the traditional supervised learning is difficult. In order to tackle this difficulty, we come up with two learning approaches in the next two sections. The first one is the passive

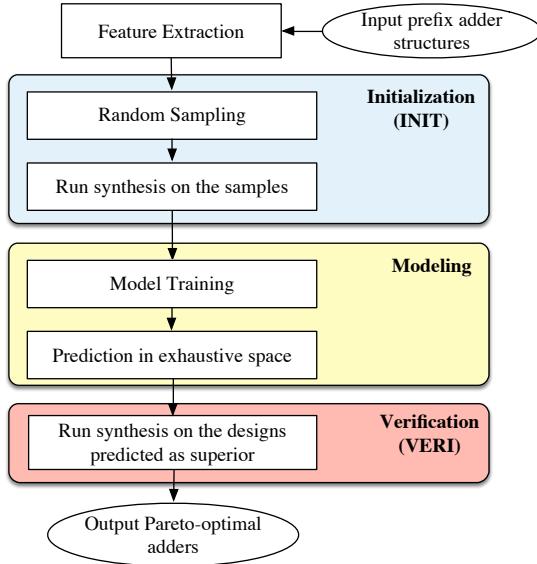


Fig. 9: Overall flow of α -sweep learning.

supervised learning where a quasi-random data sampling is performed to obtain the training data, followed by multi-objective scalarization to achieve the Pareto optimal solutions. The second one is the active learning approach where model training is integrated to finding Pareto-optimal frontiers of the design space.

IV. α -SWEEP LEARNING

In this section we propose a pareto-frontier exploration flow which is based on support vector machine. The overall flow of our α -sweep supervised learning-based Pareto-frontier exploration is presented in Fig. 9.

A. Scalarization to the Single-Objective

In this work, supervised learning is preferred over unsupervised learning since supervised learning has a substantial advantage over unsupervised learning for our problem. In particular, supervised learning allows to take advantage of the golden result, i.e., the true area/power/delay, generated by the synthesis tools for each design, instead of just letting the algorithm work out for itself what the classes should be. In general, supervised learning usually outperforms the unsupervised learning for this kind of regression and classification tasks.

Before applying machine learning for exploring Pareto frontier, we first validate the effectiveness of the features we extract by building regression models for single metric prediction. For learning models, we explored (i) several supervised learning techniques, such as linear regression, Lasso/Ridge, Bayesian ridge model and support vector regression (SVR) with linear, polynomial and radial-basis-function (RBF) kernel, and (ii) 36 features, including 4 primary features, size, mfo , target delay and utilization (tool settings), and 32 secondary features for $spfo$. We observed that we could get an R^2 score above 0.95 for area and power even with primary features and linear models. However, we don't get good scores

for delay with only primary features. Best model fitting for delay is achieved with SVR (RBF kernel) with these 4 primary and 32 secondary features. Since SVR with RBF kernel give good MSE (mean-squared-error) scores for all metrics, delay, area and power, we have used this model throughout for design space exploration.

The model experiments give us the following key insights: (i) **tool setting** can play an important role in building the learning models in EDA. For instance, MSE scores for area and power improve from 0.021 to 0.003, and 0.228 to 0.027 respectively when we add the ‘target delay’ feature in our model building, (ii) secondary features play an important role in improving the model accuracy. For instance, when we include $spfo$ features in model building, MSE score for delay improves from 0.200 to 0.170. (iii) linear models are not sufficient for modeling delay. For instance, MSE scores of delay improve from 0.214 to 0.170 when we go from linear models to SVR with RBF kernel, with the same set of features.

The problem of exploring the Pareto frontier of rich prefix adder space can be approached by first sampling a subset of prefix adder architectures, and generating the power, area, delay numbers of each prefix adder by running through the logic synthesis and physical design flow. Those known data set will be used as the training and testing data for supervised machine learning guided model fitting. Once the model is fitted, we can apply the exhaustive prefix adder architectures to this model and get the predicted Pareto frontier solution set. This is due to the merit of much faster runtime for a machine learning model in prediction stage than running the entire VLSI CAD flow.

However, conventional machine learning problem aims at maximizing the prediction accuracy rather than exploring a Pareto frontier out of a solution set. Improving the model accuracy does not necessarily improve the Pareto frontier and the direct use of the fitted model for Pareto frontier exploration can even miss up to 60% Pareto frontier points [3]. We therefore need a machine learning integrated Pareto frontier exploration methodology, where the Pareto frontier selection does not rely only on the model accuracy. So we develop a fast yet effective algorithmic methodology, enabled by regression model to explore the Pareto frontier of prefix adder solutions.

First we consider two spaces for Pareto frontier exploration: the delay vs. area as well as the delay vs. power. For either space, there exists a strong trade-off between the two metrics. For delay vs. power space, we propose to use a joint output Power-Delay function (PD) as the regression output rather than using any single output.

$$PD = \alpha \cdot Power + Delay. \quad (10)$$

The rationale of using scalarization [32] or the linear summation of the power and delay metrics is that such a linear relation provides a weighted bonding between the power and the delay so that by changing the α value, the regression model will try to minimize the prediction error on the more weighted axis hence leads to more accuracy on that direction. In contrast, the other metric direction will be predicted with less accuracy hence introducing some level of relaxations. It can be foreseen that changing the α value can lead to different

fitting accuracies of the regression model. By sweeping α over a wide range from 0 to large positive values, each time the regression model will be fitted to predict different best solutions which altogether form the Pareto frontier. We call this approach **α -sweep**. Note that, the *Power* and *Delay* values in Equation (10) are normalized and scaled to the range between 0 and 1 by Equation (11).

$$x = \frac{x - \min(X)}{\max(X) - \min(X)}, x \in X. \quad (11)$$

Similarly, we have a joint output Area-Delay (AD) function for Pareto frontier exploration on Area and Delay space.

$$AD = \alpha \cdot Area + Delay. \quad (12)$$

This α -sweep technique can be extended to simultaneously consider power, performance or delay, and area (PPA), using two scalars (α_1 and α_2) instead of one scalar factor α . The joint output function for Pareto frontier exploration on area – power – delay space can be formulated as:

$$PPA = \alpha_1 \cdot Area + \alpha_2 \cdot Power + Delay. \quad (13)$$

The results of α -sweep for both two-dimensional space and three-dimensional space are shown in the Section VI.

V. PARETO ACTIVE LEARNING

In our adder design problem, obtaining the true area/power/delay values or the labeled data for each adder requires running logic synthesis and physical design flow, which is often time-consuming if the amount of data is huge. Active learning is an iterative supervised learning which is able to interactively query the data pool to obtain the desired outputs at new data points. Since the samples are selected by the learning algorithm, the number of samples to fit a model can often be much lower than the number required in traditional supervised learning. Since an active sampling strategy is required in active learning, an “uncertainty estimation” of the prediction is needed. Gaussian Process (GP) can make predictions and, more importantly, provide the uncertainty estimation of its predictions by nature. Therefore, in this paper we further propose a Pareto active learning algorithm based on Gaussian Process regression.

A. Overall Flow

The overall flow of the Pareto active learning (PAL) is shown in Fig. 10. Given all the prefix adder structures, first we extract the feature vector for each adder as introduced in Section III-C. The active learning starts with Gaussian Process regression which will be illustrated later. Unlike the passive supervised learning in which all the features and the corresponding labels are prepared in advance, the active learning derives the labels of each training data during the learning process on-demand. To be specific, the algorithm incrementally identifies the most representative instances along with their features which are later fed into EDA synthesis flow (synthesis, placement and routing) for true area/power/delay numbers. Namely, the EDA synthesis flow and the learning process are interleaving. As more and more designs being selected, the model gets more and more accurate till convergence.

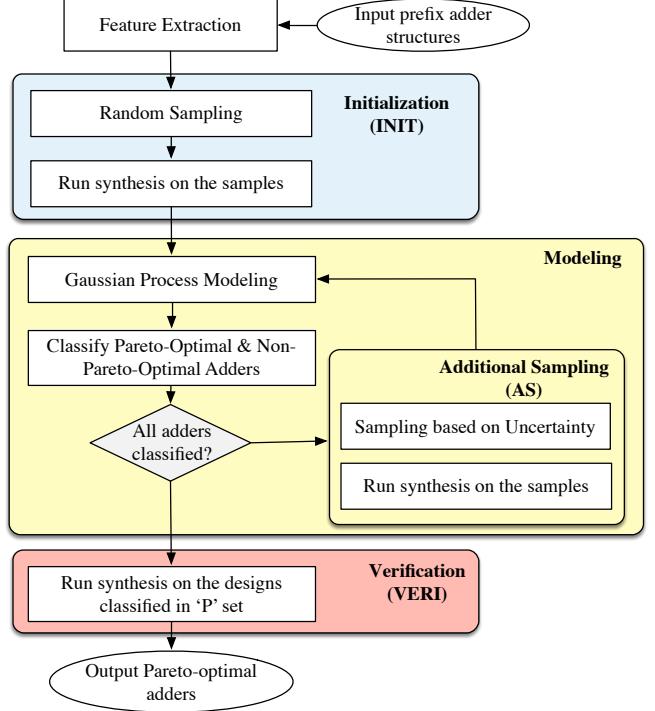


Fig. 10: Overall flow of Pareto active learning.

B. Gaussian Process Prediction

A Gaussian process is specified by its *mean function* and *covariance function*. A Pareto active learning scheme based on Gaussian process regression is proposed in [33]. The prior information is important to train the Gaussian Process model, which is a parameterized mean and covariance functions. Conventionally, the training process selects the parameters in the light of training data such that the marginal likelihood is maximized. Then the Gaussian Process model can be obtained and the regression can be proceeded with supervised input [34]. The ability of GP indicating prediction uncertainty reflects in GP learner providing a Gaussian distribution $\mathcal{N}(m(\mathbf{x}), \sigma(\mathbf{x}))$ of the values predicted for any test input \mathbf{x} by computing

$$\begin{aligned} m(\mathbf{x}) &= k(\mathbf{x}, \mathbf{X})^\top (k(\mathbf{X}, \mathbf{X}) + \sigma^2 \mathbf{I})^{-1} \mathbf{Y}, \\ \sigma^2(\mathbf{x}) &= k(\mathbf{x}, \mathbf{x}) - k(\mathbf{x}, \mathbf{X})^\top (k(\mathbf{X}, \mathbf{X}) + \sigma^2 \mathbf{I})^{-1} k(\mathbf{x}, \mathbf{X}), \end{aligned} \quad (14)$$

where \mathbf{X} is the training set, \mathbf{Y} is the supervised information of trained set \mathbf{X} . For Gaussian Process regression, a prediction of a design objective consists of a mean and a variance. The mean value $m(\mathbf{x})$ represents the predicted value and the variance $\sigma(\mathbf{x})$ represents the uncertainty of the prediction.

C. Active Learning Algorithm

The ability of GP learners in quantifying prediction uncertainty enables a suitable application for active learning. Basically, three sets are maintained during the PAL process, including a set of Pareto-optimal designs (P), non-Pareto-optimal designs (N) and ‘unclassified’ designs (U).

The GP models with discrepant prior are applied to learn the objective functions $f_{area}(\mathbf{x})$, $f_{power}(\mathbf{x})$, $f_{delay}(\mathbf{x})$. PAL

calls GP inference to predict the mean vector $\mathbf{m}(\mathbf{x})$ and the standard deviation vector $\sigma(\mathbf{x})$ of all unsampled \mathbf{x} in the design space based on Equation (14). Unlike other regression models such as linear regression and support vector regression, whose outputs are in form of numerical or categorical result, the output of GP is a distribution where uncertainties are involved. To capture the prediction uncertainty for a design \mathbf{x} , a hyper-rectangle is defined as

$$HR(\mathbf{x}) = \{\mathbf{y} : m_i(\mathbf{x}) - \beta^{\frac{1}{2}}\sigma_i(\mathbf{x}) \leq y_i \leq m_i(\mathbf{x}) + \beta^{\frac{1}{2}}\sigma_i(\mathbf{x})\},$$

where $i \in \{1, 2, 3\}$, corresponding to area, power and delay metrics in physical space. β is a user-defined parameter which determines the impact of $\sigma_i(\mathbf{x})$ on the region. In our implementation, β is set to 16 based on the analysis in [33], [35].

As shown in Fig. 10, the PAL algorithm is an iterative process. A few new points are selected in each iteration, and the GP model is retrained with new training set. Note that the model is supposed to be more and more accurate as more data being sampled. Therefore, the uncertainty region should be smaller and smaller. In order to ensure the non-increasing monotonicity of the uncertainty region while sampling and incorporating the previous evaluations, the uncertainty region of \mathbf{x} in the $(t+1)$ -th iteration is defined as

$$R_{t+1}(\mathbf{x}) = R_t(\mathbf{x}) \cap HR(\mathbf{x}), \quad (15)$$

where the initial $R_0 = \mathbb{R}^n$ which is the entire objective space.

The numbers of designs in Pareto-optimal set P and non-Pareto-optimal set N are non decreasing as iteration t increments. Thus, at iteration t , the points in P and N keep their classification. Intuitively, if one wants to compare the predicted performance of two designs, two extreme cases, i.e., optimistic prediction $\min(R_t(\mathbf{x}))$ and the pessimistic prediction $\max(R_t(\mathbf{x}))$ of each design, can be applied. If the optimistic prediction of design \mathbf{x} is dominated by the pessimistic prediction of other design \mathbf{x}' , then \mathbf{x} is classified as non-Pareto-optimal; And if the pessimistic prediction of design \mathbf{x} is not dominated by optimistic prediction of any other design \mathbf{x}' , then \mathbf{x} is classified as Pareto-optimal; A design will remain unclassified if neither condition holds. Fig. 11 is presented here as an example.

In the implementation, an error tolerance δ with value 0.001 is applied during classification. The rules for classification can be represented as follows.

$$\mathbf{x} \in \begin{cases} P, & \text{if } \max(R_t(\mathbf{x})) \leq \min(R_t(\mathbf{x}')) + \delta, \\ N, & \text{if } \max(R_t(\mathbf{x}')) \leq \min(R_t(\mathbf{x})) + \delta, \\ U, & \text{otherwise.} \end{cases} \quad (16)$$

After classification in each iteration, a new adder design with the largest length of the diagonal of its uncertainty region $R(\mathbf{x})$ is selected for sampling. The value is attached to \mathbf{x} as

$$w_t(\mathbf{x}) = \max_{\mathbf{y}, \mathbf{y}' \in R_t(\mathbf{x})} \|\mathbf{y} - \mathbf{y}'\|_2. \quad (17)$$

Intuitively, Equation (17) picks the points which are most worthy exploring. Afterwards, these designs are going through EDA flow to get the real area, power and delay numbers, and

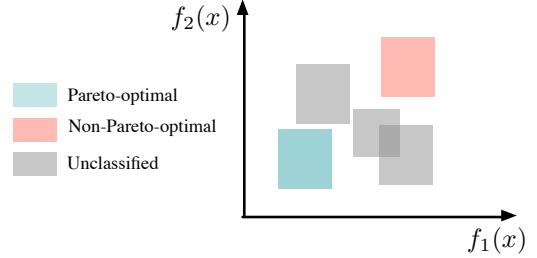


Fig. 11: An example of classification.

the GP model will hence be improved with those feedback results.

Algorithm 1 Active Learning for Pareto-frontier Exploration

Require: Adder architectural design space E , GP prior, maximum iteration number T_{\max} ;
Ensure: predicted Pareto-optimal set \hat{P} ;

- 1: $P \leftarrow \emptyset, N \leftarrow \emptyset, U \leftarrow E$;
- 2: Randomly select a small subset $X = \{\mathbf{x}_i\}$ of E ;
- 3: Get true values $Y = \{\mathbf{y}_i | \mathbf{y}_i = \text{EDAFlow}(\mathbf{x}_i)\}$;
- 4: $S \leftarrow X$;
- 5: $R_0(\mathbf{x}) \leftarrow \mathbb{R}^n, \forall \mathbf{x} \in E$;
- 6: $t \leftarrow 0$;
- 7: **while** $U \neq \emptyset$ and $t < T_{\max}$ **do**
- 8: Building GP model with $\{(x_i, y_i) : \forall x_i \in S\}$;
- 9: Obtain $R_t(\mathbf{x}), \forall \mathbf{x} \in E$;
- 10: **for all** $\mathbf{x} \in U$ **do**
- 11: **if** \mathbf{x} is Pareto-optimal based on Equation (16) **then**
- 12: $P.\text{add}(\mathbf{x}), U.\text{delete}(\mathbf{x})$;
- 13: **else if** \mathbf{x} is non-Pareto-optimal based on Equation (16) **then**
- 14: $N.\text{add}(\mathbf{x}), U.\text{delete}(\mathbf{x})$;
- 15: **end if**
- 16: **end for**
- 17: Obtain $w_t(\mathbf{x}), \forall \mathbf{x} \in (U \cup P) \setminus S$;
- 18: Choose $\mathbf{x}' \leftarrow \text{argmax}\{w_t(\mathbf{x})\}$;
- 19: $S \leftarrow S \cup \mathbf{x}'$;
- 20: $t \leftarrow t + 1$;
- 21: Obtain new data $(\mathbf{x}', \mathbf{y}')$ by running EDA flow;
- 22: **end while**
- 23: $\hat{P} \leftarrow P$;

The entire process is presented in Algorithm 1. It starts with the initialization (lines 1–6). In each iteration, the GP model is trained with the current training set S , and the uncertainty region for each design is obtained (lines 8–9). Then the designs in the U set are classified based on uncertainty regions and classification rules (lines 10–16). After that, the design with the largest uncertainty is sampled and the sampling set S is updated (lines 17–19). The newly sampled design is fed into synthesis tools to get the label which is used for training GP model in the next iteration (line 21). The learning process stops after all adder designs in architectural design space are classified. The prediction is $\hat{P} = P$ (line 23). Suppose T_{\max} is the maximum number of iterations, and $|E|$ is the size of

solution set, then the complexity of Algorithm 1 is at most $\mathcal{O}(T_{\max}|E|)$, as maximum size of U can be $|E|$. However, it should be stressed that although there are $T_{\max}|E|$ operations for PAL algorithm, the cost of each operation (which is a simple inference based on the Gaussian Process Regression model) is negligible in comparison to EDA synthesis flow run-time, and we will demonstrate later in TABLE IV that the total run-time of different approaches are dictated by the number of EDA synthesis flow runs needed in the respective approaches.

VI. EXPERIMENTAL RESULTS

In this section we show the effectiveness of the proposed algorithms and methodologies. First we compare the physical solution space before/after applying PGG algorithm. Then the Pareto frontier obtained by α -sweep is presented. Next, we demonstrate the Pareto frontier obtained by active learning, and compare the quality of Pareto frontiers generated by two approaches. Finally, we compare our explored optimal adders against legacy adders.

Since high performance adders are commonly used in CPU architectures which are typically 64 bit, we have mainly presented the results for 64 bit adders to demonstrate the methodology. However, the approach is very general to be used for adders of arbitrary bit-width. The flow is implemented in C++ and Python on Linux machine with 72GB RAM and 2.8GHz CPU. We use Design Compiler [28] (version F-2011.09-SP3) for logical synthesis, and IC Compiler [29] (version J-2014.09-SP5-3) for the placement and routing. "tt1p05v125c" corner and Non Linear Delay Model (NLDM) in 32nm SAED cell-library for LVT class [30] (available by University Program) is used for technology mapping. Primary input activity of 0.1 is used along with 1GHz operating frequency for power estimation. Regarding the tool settings, target delays of 0.1ns, 0.2ns, 0.3ns and 0.4ns are used. Utilization values are set to 0.5, 0.6, 0.7 and 0.8. We used Python based machine learning package scikit-learn [36] for the predictions. Throughout our all experiments, the run time for machine learning predictions is less than a minute.

We relied more on the fidelity of the SAED library rather than accuracy considering that SAED library may not be very realistic as that used in industry. For instance, the FO4 delay for a unit sized inverter for this library in the operating corner is 36ps [20], [37]. So 11 FO4 delay, typically being presented to be the delay for 64-bit adders in literatures [38], is approximately 400ps which is close to the reported delays for 64-bit adders in our work. To further demonstrate the fidelity of this library, we run the Kogge-Stone adders with bit-widths of 8, 16, 32, 64, 128 and 256 through the synthesis flow using this library. Then we normalize the measured delay in terms of FO4 delay, and plot it with bit-width (n) as shown in Fig. 12. It can be seen that the delay is linear with $\log_2 n$, which is expected for a logarithmic tree adder such as Kogge-Stone adder. So we believe if this algorithmic methodology is applied to more realistic industrial libraries, it can show similar benefit as demonstrated with SAED 32nm library.

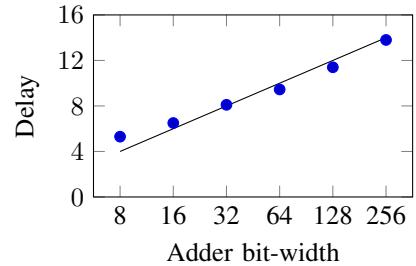


Fig. 12: Delay values (\times FO4 delay) of Kogge-Stone adders with various bit-width.

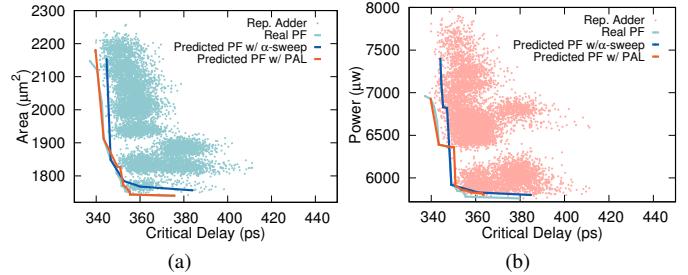


Fig. 13: (a) Pareto Frontier: area vs. delay; (b) Pareto Frontier: power vs. delay.

To validate the optimality and the hypervolume error of the two learning approaches against the real world solution space, we need to run the logical/physical EDA flow on a large set of adder solutions. Our machine and tool set takes about 5.5 minutes to complete this full flow of a single prefix adder. Therefore, we select a reasonable number (3000) of prefix adder solutions, which eventually took about 300 hours to complete, but still a comparatively larger data set in comparison to our training data set. Crucially, those 3000 adders are also sampled in a **Quasi-random** manner in order to represent the entire solution space.

A. Pareto Frontier Predicted by α -sweep Learning

In this experiment, we show the effectiveness of our α -sweep learning approach. We apply the α -sweep method with 15 different α values of $(1000, 0, 100, \frac{1}{100}, 50, \frac{1}{50}, 20, \frac{1}{20}, 10, \frac{1}{10}, 8, \frac{1}{8}, 2, \frac{1}{2}, 1)$, and collect the best 150 solutions for delay-area and delay-power spaces where for each α value, the best 10 architectures with lowest PD or AD values are fed into the logical/physical EDA flow to generate similar Pareto points. Note that $15 + 15 = 30$ learning models have been derived for this for all, but it is very fast as the same training data have been used, and the models are regression based.

Fig. 13(a) and Fig. 13(b) respectively show the corresponding Pareto frontiers of the α -sweep approach and the ground truth Pareto frontiers for the 3000 representative adders. Each dot in the delay-area or delay-power space indicates one adder solution after going through the logical/physical EDA flow. We can see that generally the predicted Pareto frontier solutions are fairly close to the real Pareto frontier, with some exceptions. Overall, the proposed approach can effectively

TABLE II: Comparison of different model accuracies

Model	MSE			Hypervolume error		
	Area	Power	Delay	Area-Delay	Power-Delay	Area-Power-Delay
Original	0.003	0.027	0.170	0.139	0.122	0.154
Noisy	0.024	0.951	0.711	0.168	0.148	0.162

 TABLE III: Pareto frontiers for PAL vs. α -sweep [39]

Objective Hypervolume error		PAL	α -sweep [39]
Area-Delay	average	0.100	0.139
	best	0.044	0.093
Power-Delay	average	0.109	0.122
	best	0.075	0.076
Area-Power-Delay	average	0.056	0.154
	best	0.039	0.125

Notes: All hypervolume error above are collected from 1000 repeated experiments.

achieve near optimal Pareto frontier without affording to spend expensive runtime on every adder. So this learning based methodology can be readily adopted to achieve Pareto frontiers for much larger solution space which is intractable for exhaustive exploration by conventional design flow.

We have conducted additional experiments to show the impacts of the low accuracy of the machine learning model. The basic idea is to inject random noise in the prediction stage, i.e., additional Gaussian noise is added into the predicted value. The accuracy will be lower than original results. Then we explore the Pareto frontier based on the noisy prediction. Generally, the quality of the final Pareto frontier is worse than original model. The comparison of Pareto frontier quality is presented in TABLE II.

B. Comparison of the Quality of Pareto-Frontier between PAL and α -sweep

We implement PAL to predict Pareto-optimal designs in both two-dimensional design spaces which are area-delay space and power-delay space, as well as three-dimensional space which is area-power-delay space. The results are compared with those of [39]. The initial input set for both area-delay and power-delay is of size 250, which are randomly selected from the exhaustive design space. The curves of Pareto frontiers for two-dimensional spaces are shown in Fig. 13. The hypervolume of area-delay Pareto frontiers are calculated with reference point (max(delay), max(area)). Similarly, The hypervolume of power-delay Pareto frontiers are calculated with reference point (max(delay), max(power)). Note that the unit for delay is nanosecond (ns) when calculating the hypervolume. It should be stressed that there is a sort of randomness in both α -sweep and PAL algorithm. For α -sweep, the training set is selected randomly. On the contrary, the initial set in PAL is randomly selected (line 2), thereby may result in different outputs. So the experiments are conducted for 1000 times such that the general performance is reflected. The comparison between two approaches are shown in TABLE III. Comparing the hypervolume error of Pareto frontier obtained by PAL and α -sweep, it can be seen that PAL achieves better performance

in predicting Pareto frontier in all design spaces, including area-delay, power-delay, and area-power-delay spaces.

C. Runtime Comparisons among Exhaustive Approach, α -sweep and PAL

There are three factors that will affect the runtime: (i) the total number of EDA synthesis runs required; (ii) Among all these required EDA synthesis runs how many of them can be parallelized; (3) The runtime of the training process in machine learning model. All these details are recorded in TABLE IV. The ‘INIT’ represents the set of training data in the α -sweep and the initial set in PAL, which can be parallelized because all the points are obtained in advance. The ‘AS’ represents the set of designs which are actively sampled during the learning process, which cannot be parallelized. The α -sweep approach does not involve active sampling, so the ‘AS’ set is none here. The ‘VERI’ represents the set of designs which are predicted to be Pareto-optimal. We should run EDA synthesis flow to get the real PPA values of these designs to extract the Pareto-frontier. This set of designs are obtained after the learning process stops, so the EDA synthesis runs on these designs are also conducted offline, which can be parallelized. Each EDA synthesis run takes about 5.5 minutes.

Then we can compare the total runtime of different exploration methodologies. For exhaustive exploration, all the prefix adders should be fed into EDA tools for synthesizing to obtain the value of each metric, which is extremely time-consuming. There is no training, additional sampling, verification. The total runtime cost involves EDA flows of all the designs in the design space. The Pareto frontier can be extracted from the results, whose runtime is much less than synthesizing and can be neglected. The total runtime is

$$T_{exh} = \frac{5.5 \times \#INIT}{\#Machines}. \quad (18)$$

Since the entire solution space is so huge that one can hardly run all of them, in our experiment, we sample representative 10K designs by random sampling. The total runtime of synthesizing is about 55000 minutes with single machine. It should be noted that the entire solution space is much more than 10K.

In the exploration by α -sweep, not all adders in the design space are needed for synthesizing. The total runtime is

$$T_\alpha = \frac{5.5 \times (\#INIT + \#VERI)}{\#Machines} + \text{Modeling time}. \quad (19)$$

In our experiment, we select 2500 of the designs out of those 10K designs by random sampling to build the model, including training and testing phases. It takes about 1.5 minutes to build the model and make predictions. When exploring in area-power-delay design space, 150 designs on average in the

TABLE IV: Comparison of runtime with single machine among different approaches

Method	#INIT	#AS	#VERI	#Total	Runtime (mins)		
					EDA	Modeling	Total
Exhaustive	10000	-	-	10000	55000.0	-	55000.0
α -sweep	2500	-	150	2650	14575.0	20.0	14595.0
PAL	700	10	290	1000	5500.0	2.0	5502.0

Notes: The designs in “#INIT” and “#VERI” can be synthesized in parallel. The number of designs in each category is collected from 1000 repeated experiments.

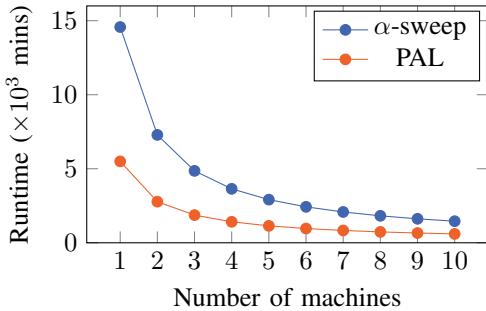


Fig. 14: Comparison of runtime with different number of machines.

design space are predicted to be Pareto-optimal. So on average 2650 designs are needed. The runtime for synthesizing is 14575 minutes. Note that in terms of learning models, the α -sweep method needs to build $15 \times 15 = 225$ models since α_1 and α_2 both have 15 values to choose from.

Similarly, the runtime of PAL can be calculated by

$$T_{PAL} = \frac{5.5 \times (\#INIT + \#VERI)}{\#Machines} + 5.5 \times \#AS + \text{Modeling time.} \quad (20)$$

The size of initial set is fixed, which is 700. It takes about 4 minutes to build the model and make predictions during the PAL process. When exploring the Pareto-optimal designs in area-power-delay space, 10 designs on average are sampled during PAL. 290 designs on average in the design space are predicted to be Pareto-optimal. In total, 1000 designs are needed on average. The runtime is 5500 minutes with single machine. PAL algorithm needs to build N models where N is the number of iterations in PAL. In our implementation, the maximum iteration is set to 20. It can be observed that the active learning approach outperforms the α -sweep learning in terms of both the quality of Pareto frontier and the number of EDA flow runs.

Note that all the runtime calculations are based on single machine. However, the EDA synthesis runs in all three flows can be distributed to multiple machines if available, except the adders sampled during active learning, which (10 on average in our experiments) is very less in comparison to the total number of the synthesis runs. So PAL can get a significant speedup over α -sweep and exhaustive approach with single machine and multiple machines.

D. Comparison on Different Sampling Strategies in PAL

In the sampling stage of PAL, the number of instances to be sampled has impact on the runtime since the EDA flow is required to obtain the real value for area, power and delay. The less instances we sample in each iteration, the more iterations are needed to ensure the PAL process converge, which is more likely to result in less samples in total. The more instances we sample in each iteration, the less iterations are needed. However, the total number of sampled instances would be large. In practice, the runtime cost of running EDA flow can be reduced by parallel execution if there are multiple licenses available. In this section, we explore the effect of different sampling strategies in terms of the total runtime and the quality of Pareto frontier in practical scenarios.

The results for different sampling strategies are listed in Fig. 15. Since the EDA flow for synthesis, placement and routing takes up the most significant part of the total runtime cost, the key factor is the number of adders which needs to be through EDA tool flow. If we have multiple machines available for the EDA tool flow, the runtime is determined by the total number of iterations as long as the number of samples does not exceed the number of machines. From the result, it can be seen that we can obtain the Pareto-frontier with comparable quality, using less runtime.

Note that when the sample size increases from 1 to 5, the average hypervolume error increases from 0.056 to about 0.070, which is still less than 0.154 (average hypervolume error achieved in α -sweep approach). Therefore, batch sampling can not only take care of parallel synthesizing but also achieve better quality for Pareto frontier than α -sweep, which can also show the advantages of the PAL.

E. Adder Performance Comparison

Finally, we compare our explored adders against DesignWare adders, legacy adders, such as Kogge-Stone, Sklansky, as well as a state-of-the-art adder synthesis algorithm in TABLE V. Since our approach generates numerous solutions, it is not feasible to perform a one-to-one comparison. Instead for each of the solution points in regular adders and [19], we have picked the Pareto points from our solution set which are able to excel them in all metrics. For instance, P_1 could provide around 8ps better delay with respectively 14% and 12% lesser area and power over Kogge-Stone adder. The DesignWare adders are synthesized from behavioral description of adder ($Y = A + B$) with the 16 configurations of tool settings (Combination of 4 target delay and 4 utilization values) that are used in generating the Fig. 4. We pick the one with best

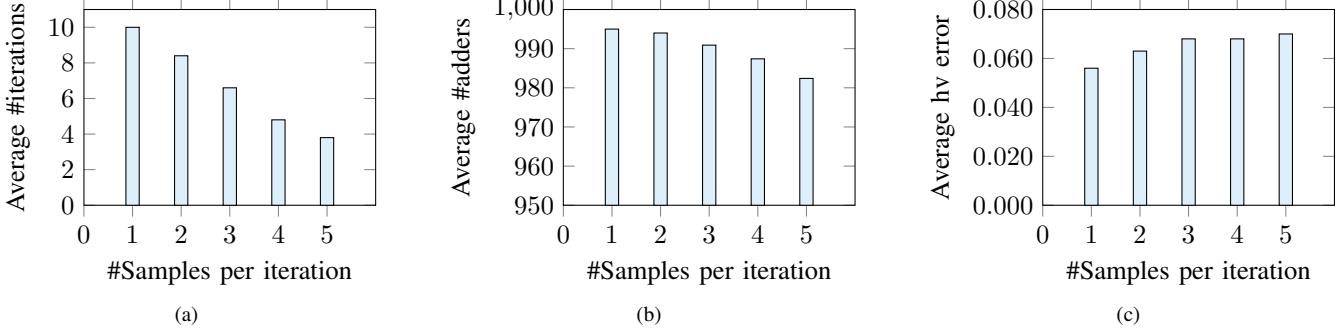


Fig. 15: Comparison among different number of samples per iteration.

TABLE V: Comparison with other approaches for 64 bit adders

Method	Delay (ps)	Area (μm^2)	Energy (fJ/op)
DesignWare	346.5	2531.3	8160
Ours (P_1)	339.0	2180.8	6930
Kogge-Stone	347.9	2563.7	8780
Ours (P_1)	339.0	2180.8	6930
Sklansky	356.1	1792.5	6100
Ours (P_2)	353.0	1753.0	5900
[19]	348.7	1971.4	6980
Ours (P_3)	343.0	1912.6	6390

delay, denoted by ‘‘DesignWare’’ in TABLE V. The same pareto point P_1 dominates that solution by providing around 7.5ps better delay, 14% lesser area, and 15% lesser energy. For [19] we pick the best delay solution. Note for a fixed mfo , [19] can give prefix network with smaller size, but this approach only provides a limited set of prefix structures. As a result, it is hard for [19] to explore the full physical design space of adders by machine learning. It should be stressed that [19] beats the custom adders implemented in an industrial design, and our methodology is able to excel the adders generated by the algorithm presented in [19].

VII. CONCLUSION

This paper presents a novel methodology of machine learning guided design space exploration for power efficient high-performance prefix adders. We have successfully demonstrated the effectiveness of our learning models, developed by training with quasi-random sampled data and features encapsulating architectural and tool attributes. In addition, an active learning approach is applied to ease the demand of labeled data and achieves even better Pareto frontier. Our adder synthesis algorithm is able to generate a wider solution space in comparison to a state-of-the-art algorithm, and when integrated with the learning model, could provide a remarkable performance vs. power vs. area Pareto frontier over a large representative solution space. To the best of our knowledge, this is the first work to bridge the gap between architectural and physical solution space for parallel prefix adders.

REFERENCES

- [1] B. Yu, D. Z. Pan, T. Matsunawa, and X. Zeng, ‘‘Machine learning and pattern matching in physical design,’’ in *Proc. ASPDAC*, 2015, pp. 286–293.
- [2] W.-T. J. Chan, K. Y. Chung, A. B. Kahng, N. D. MacDonald, and S. Nath, ‘‘Learning-based prediction of embedded memory timing failures during initial floorplan design,’’ in *Proc. ASPDAC*, 2016, pp. 178–185.
- [3] 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. DATE*, 2016, pp. 918–923.
- [4] W.-H. Chang, L.-D. Chen, C.-H. Lin, S.-P. Mu, M. C.-T. Chao, C.-H. Tsai, and Y.-C. Chiu, ‘‘Generating routing-driven power distribution networks with machine-learning technique,’’ in *Proc. ISPD*, 2016, pp. 145–152.
- [5] R. Samanta, J. Hu, and P. Li, ‘‘Discrete buffer and wire sizing for link-based non-tree clock networks,’’ in *Proc. ISPD*, 2008, pp. 175–181.
- [6] R. P. Brent and H. T. Kung, ‘‘A regular layout for parallel adders,’’ *IEEE Transactions on Computers*, vol. C-31, no. 3, pp. 260–264, 1982.
- [7] P. M. Kogge and H. S. Stone, ‘‘A parallel algorithm for the efficient solution of a general class of recurrence equations,’’ *IEEE Transactions on Computers*, vol. 100, no. 8, pp. 786–793, 1973.
- [8] T. Han and D. Carlson, ‘‘Fast area-efficient VLSI adders,’’ *Proc. ARITH*, pp. 49–56, 1987.
- [9] J. Sklansky, ‘‘Conditional sum addition logic,’’ *IRE Trans. on Electronic Computers*, vol. EC-9, no. 2, pp. 226–231, 1960.
- [10] C. Zhou, B. M. Fleischer, M. Gschwind, and R. Puri, ‘‘64-bit prefix adders: Power-efficient topologies and design solutions,’’ *Proc. CICC*, pp. 179–182, 2009.
- [11] J. Liu, Y. Zhu, H. Zhu, C.-K. Cheng, and J. Lillis, ‘‘Optimum prefix adders in a comprehensive area, timing and power design space,’’ in *Proc. ASPDAC*, 2007, pp. 609–615.
- [12] N. H. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison Wesley, 2004.
- [13] T. Matsunaga and Y. Matsunaga, ‘‘Area minimization algorithm for parallel prefix adders under bitwise delay constraints,’’ in *Proc. GLSVLSI*, 2007, pp. 435–440.
- [14] J. Liu, S. Zhou, H. Zhu, and C.-K. Cheng, ‘‘An algorithmic approach for generic parallel adders,’’ in *Proc. ICCAD*, 2003, pp. 734–740.
- [15] J. P. Fishburn, ‘‘A depth-decreasing heuristic for combinational logic; or how to convert a ripple-carry adder into a carry-lookahead adder or anything in-between,’’ in *Proc. DAC*, 1990, pp. 361–364.
- [16] R. Zimmermann, ‘‘Non-heuristic optimization and synthesis of parallel prefix adders,’’ *Proc. IWLAS*, pp. 123–132, 1996.
- [17] M. Snir, ‘‘Depth-size trade-offs for parallel prefix computation,’’ *Journal of Algorithms*, vol. 7, no. 2, pp. 185–201, 1986.
- [18] H. Zhu, C.-K. Cheng, and R. Graham, ‘‘Constructing zero-deficiency parallel prefix adder of minimum depth,’’ in *Proc. ASPDAC*, 2005, pp. 883–888.
- [19] S. Roy, M. Choudhury, R. Puri, and D. Z. Pan, ‘‘Polynomial time algorithm for area and power efficient adder synthesis in high-performance designs,’’ in *Proc. ASPDAC*, 2015, pp. 249–254.
- [20] ———, ‘‘Towards optimal performance-area trade-off in adders by synthesis of parallel prefix structures,’’ *IEEE TCAD*, vol. 33, no. 10, pp. 1517–1530, 2014.

- [21] M. Choudhury, R. Puri, S. Roy, and S. C. Sundararajan, "Automated synthesis of high performance two operand binary parallel prefix adder," Patent US 8,683,398, Mar, 2014.
- [22] G. Palermo, C. Silvano, and V. Zaccaria, "ReSPIR: a response surface-based pareto iterative refinement for application-specific design space exploration," *IEEE TCAD*, vol. 28, no. 12, pp. 1816–1829, 2009.
- [23] H.-Y. Liu and L. P. Carloni, "On learning-based methods for design-space exploration with high-level synthesis," in *Proc. DAC*, 2013, pp. 50:1–50:7.
- [24] 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. DATE*, 2016, pp. 918–923.
- [25] B. R. Zeydel, T. T. J. H. Kluter, and V. G. Oklobdzija, "Efficient mapping of addition recurrence algorithms in CMOS," *Proc. ARITH*, pp. 107–113, 2005.
- [26] M. Ketter, D. M. Harris, A. Macrae, R. Glick, M. Ong, and J. Schauer, "Implementation of 32-bit Ling and Jackson adders," *Proc. Asilomar*, pp. 170–175, 2011.
- [27] E. Zitzler, D. Brockhoff, and L. Thiele, "The hypervolume indicator revisited: On the design of pareto-compliant indicators via weighted integration," in *Evolutionary multi-criterion optimization*, 2007, pp. 862–876.
- [28] "Synopsys Design Compiler," <http://www.synopsys.com>.
- [29] "Synopsys IC Compiler," <http://www.synopsys.com>.
- [30] "Synopsys SAED Library," <http://www.synopsys.com/Community/UniversityProgram/Pages/32-28nm-generic-library.aspx>, accessed 23-April-2016.
- [31] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Transactions on knowledge and data engineering*, vol. 22, no. 10, pp. 1345–1359, 2010.
- [32] K. Turner and J. Ghosh, "Estimating the bayes error rate through classifier combining," in *Proc. ICPR*, vol. 2, 1996, pp. 695–699.
- [33] M. Zuluaga, A. Krause, G. Sergent, and M. Püschel, "Active learning for multi-objective optimization," in *Proc. ICML*, 2013, pp. 462–470.
- [34] C. E. Rasmussen and C. K. I. Williams, *Gaussian Process for Machine Learning*. The MIT Press, 2006.
- [35] N. Srinivas, A. Krause, S. Kakade, and M. Seeger, "Gaussian process optimization in the bandit setting: no regret and experimental design," in *Proc. ICML*, 2010, pp. 1015–1022.
- [36] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, no. Oct., pp. 2825–2830, 2011.
- [37] S. Roy, M. Choudhury, R. Puri, and D. Z. Pan, "Polynomial time algorithm for area and power efficient adder synthesis in high-performance designs," *IEEE TCAD*, vol. 35, no. 5, pp. 820–831, 2016.
- [38] T. McAuley, W. Koven, A. Carter, P. Ning, and D. M. Harris, "Implementation of a 64-bit Jackson adders," *Proc. Asilomar*, pp. 1149–1154, 2013.
- [39] S. Roy, Y. Ma, J. Miao, and B. Yu, "A learning bridge from architectural synthesis to physical design for exploring power efficient high-performance adders," in *Proc. ISLPED*, 2017, pp. 1–6.