HL-Pow: Learning-Assisted Pre-RTL Power Modeling and Optimization for FPGA HLS

Zhe Lin, *Member, IEEE*, Tingyuan Liang, *Student Member, IEEE*, Jieru Zhao, *Member, IEEE*, Sharad Sinha, *Senior Member, IEEE*, and Wei Zhang, *Member, IEEE*

***Abstract*—High-level synthesis (HLS) enables designers to customize hardware designs without the need for delving into low-level hardware details. However, it is still challenging to establish the correlation between the power consumption and hardware designs at an early design stage such as HLS. To overcome this problem, we introduce HL-Pow, a preregister-transfer-level (pre-RTL) power modeling, and optimization framework for FPGA HLS with the aid of up-to-date artificial intelligence techniques, which features high accuracy, speed, and generalization ability. HL-Pow is comprised of a power modeling framework and a design space exploration (DSE) engine. The power modeling framework encompasses: 1) a fully customized and light-weight feature construction flow to effectively identify and capture features that exert a major influence on power consumption and 2) a modeling flow that can build an accurate, fast, and transferable pre-RTL power estimator. With HL-Pow, the power evaluation process for hardware designs with FPGA HLS can be significantly expedited by circumventing the invocation of the time-consuming logic synthesis, physical design, and gate-level simulation steps. Furthermore, we describe a novel a priori knowledge-guided DSE algorithm which can combined with our power modeling approach to jointly achieve the design optimization for latency and power consumption with high efficiency and high quality. Experimental results demonstrate that HL-Pow produces accurate power prediction that is only 4.82% away from onboard power measurement, while offering a speedup of 24–190× (84× on avg.). In addition, HL-Pow shows high generalization ability across applications with different characteristics and from various domains. Finally, the proposed DSE algorithm can reach a close approximation of the real Pareto frontier while only requiring traversing a small subset of design points in a broad design space.**

***Index Terms*—Design space exploration (DSE), field-programmable gate array (FPGA), high-level synthesis (HLS), machine learning, power modeling.**

Manuscript received 18 July 2022; revised 24 October 2022 and 16 January 2023; accepted 27 January 2023. Date of publication 17 February 2023; date of current version 20 October 2023. This article was recommended by Associate Editor Z. Zhang. *(Corresponding author: Zhe Lin.)*

Zhe Lin is with the School of Integrated Circuits, Sun Yat-sen University, Shenzhen 518107, China (e-mail: linzh235@mail.sysu.edu.cn).

Tingyuan Liang and Wei Zhang are with the Department of Electronic and Computer Engineering, Hong Kong University of Science and Technology, Hong Kong, China (e-mail: tliang@ust.hk; wei.zhang@ust.hk).

Jieru Zhao is with the Department of Computer Science and Engineering, Shanghai Jiaotong University, Shanghai 200240, China (e-mail: zhao-jieru@sjtu.edu.cn).

Sharad Sinha is with the Department of Computer Science and Engineering, Indian Institute of Technology Goa, Goa 403401, India (e-mail: sharad_sinha@ieee.org).

Digital Object Identifier 10.1109/TCAD.2023.3246387

I. Introduction

HIGH-LEVEL synthesis (HLS) [1] automates the process of translating applications described by high-level languages (e.g., C++ and Python) into cycle-accurate register-transfer level (RTL) designs. With the aid of HLS tools, designers targeting hardware implementation on field-programmable gate arrays (FPGAs) or application-specific integrated circuits (ASICs) are no longer required to make a great effort developing hardware microarchitectures, e.g., carefully crafting the intracomponent and interstage pipelines. In addition, modern HLS tools are able to deliver relatively good estimation of quality of results (QoRs) regarding performance and resource estimation, and also offer a series of design knobs, or so-called directives or pragmas, to help designers tune the two aforementioned design metrics. Overall, the productivity and flexibility brought by HLS notably speedup the development process of hardware designs, opening up a number of opportunities for efficient design space exploration (DSE) [2], [3], [4], [5], [6], [7], [8], [9], [10]. However, even the off-the-shelf HLS tools [11], [12], [13] are still lacking in mature power analysis techniques, making it difficult to clearly pinpoint the influence of different HLS optimization strategies on power consumption.

Power consumption is a primal concern for a broad spectrum of FPGA designs, especially, for portable electronic devices and embedded systems. The common practice to obtain FPGA power consumption is through real measurement or gate-level power estimation, both of which require designers to spend substantial efforts. The traditional FPGA power evaluation flow starting from HLS is shown in Fig. 1(a). Given a C/C++ program, the HLS front-end and back-end flows are invoked to convert software programs into RTL designs. Afterward, the RTL and physical implementation flow, including logic synthesis, placement and routing, etc, is applied to compile the RTL designs into gate-level netlists. Subsequently, to perform power measurement, the bitstreams are converted from gate-level netlists and transferred to hardware systems for execution, and power consumption can be measured on FPGA by specialized devices. As for power estimation, gate-level simulation with real data as input is additionally performed to capture switching activities of the IO and internal signals. Thereafter, prebuilt power analyzers [14], [15], [16] are deployed to compute power consumption, leveraging the gate-level netlists and signal activities as the input. Finally, in pursuit of higher power efficiency, designers can refine the hardware architectures according to the feedback of power consumption, and reimplement the above design flow again for verification.

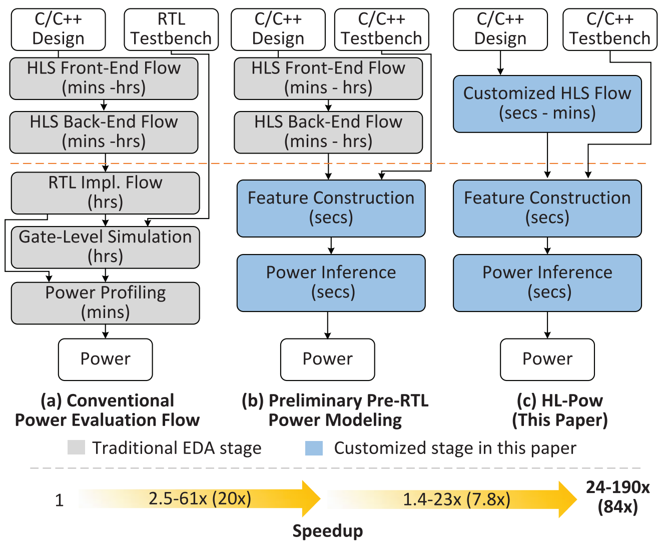

C/C++ Design
RTL Testbench
HLS Front-End Flow (mins -hrs)
HLS Back-End Flow (mins -hrs)
RTL Impl. Flow (hrs)
Gate-Level Simulation (hrs)
Power Profiling (mins)
Power
C/C++ Design
C/C++ Testbench
HLS Front-End Flow (mins - hrs)
HLS Back-End Flow (mins - hrs)
Feature Construction (secs)
Power Inference (secs)
Power
C/C++ Design
C/C++ Testbench
Customized HLS Flow (secs - mins)
Feature Construction (secs)
Power Inference (secs)
Power
(a) Conventional Power Evaluation Flow
(b) Preliminary Pre-RTL Power Modeling
(c) HL-Pow (This Paper)
Traditional EDA stage
Customized stage in this paper
1
2.5-61x (20x)
1.4-23x (7.8x)
24-190x (84x)
Speedup

Fig. 1. Comparison of FPGA power estimation methods with HLS: the (b) preliminary method shows a speedup of 2.5–61× (20× on avg.) over the (a) conventional method; (c) our HL-Pow methodology demonstrates a speedup of 1.4–23× (7.8× on avg.) over method (b), finally leading to a compound speedup of 24–190× (84× on avg.) over method (a).

Nevertheless, the development of power-efficient designs usually necessitates multiple rounds of power evaluation and design modification, which results in a long design time and high-labor intensity.

The state-of-the-art works [17], [18], [19], [20], [21] have presented power modeling techniques to accelerate the power analysis process of FPGA designs. These methods fall into two categories: 1) post-RTL power modeling and 2) preregister-transfer-level (pre-RTL) power modeling. Post-RTL power modeling methods [17], [18], [19] first collect architectural information from the gate-level netlists of post-synthesis or post-placement&routing and signal activities from vector-based simulation or onboard detection, and then develop a learning-based model specialized for each hardware design, and finally perform power inference at predefined time intervals. These methods are time consuming in acquiring features for power prediction and show a lack of generalization ability to accommodate the prebuilt power models to new hardware designs, which are particularly inefficient for the HLS design paradigm that typically demands fast QoR evaluation and optimization across a rich set of candidate design instances. In contrast, pre-RTL power modeling methods [20], [21] are faster in evaluation speed by bypassing most of the post-RTL design stages during power prediction and, thus, they are more appropriate for the HLS paradigm. However, even the up-to-date research works are restricted to the OpenCL-to-FPGA programming model [20] or C/C++ programs with affine functions [21], whose methodologies are difficult to directly transfer to general HLS design cases.

In light of the above considerations, in this work, we aim to investigate a generic pre-RTL power prediction methodology for FPGA designs starting from the HLS design flow, and also strive to speedup power-oriented design space optimization for FPGA HLS. Specifically, we propose the HL-Pow methodology. First, HL-Pow offers a modeling strategy with high-generalization ability so that unseen applications[1] can use a one-size-fits-all predictive model for power inference without model regeneration as long as the same FPGA substrate is targeted. Second, the power prediction of HL-Pow for new HLS applications is fast in runtime, as it dispenses with the need to go through the tedious post-RTL power estimation or measurement flow. To the best of our knowledge, HL-Pow is the first work that jointly achieves high accuracy, superior speed, and strong generalization ability for pre-RTL hardware power modeling with FPGA HLS. Finally, for the first time, HL-Pow sheds light on the joint effect of loop unrolling and pipelining on both latency and power of the derived hardware designs, and further distills this effect into a priori domain knowledge to effectively guide the latency-power DSE.

A preliminary version of this work appears in [22]. In this article, we augment our preliminary work [22] with a lightweight and fully customized HLS flow to further expedite the feature construction process, improve the power modeling with an ensemble learning strategy, generalize key observations to strengthen our DSE algorithm, investigate more complicated datasets, and study the importance of different features on power prediction results. As shown in Fig. 1, our enhanced power modeling flow [see Fig. 1(c)] reaps an average speedup of 7.8× over the preliminary work [22] [see Fig. 1(b)], and finally leads to an overall speedup of 84× over the conventional hardware power evaluation method [see Fig. 1(a)], while maintaining high-prediction accuracy. In particular, we highlight the contributions of this work as follows.

1) We introduce a fully customized feature construction flow for rapid identification and extraction of features closely related to FPGA power consumption, completely based upon an optimized and tailored HLS design flow.
2) We propose a learning-assisted hardware power modeling methodology with the ability to deliver accurate, fast, and transferable pre-RTL power estimation for FPGA HLS.
3) We describe a novel a priori-guided HLS DSE algorithm that can be coupled with our power modeling flow and demonstrate how the tradeoff between the latency and power consumption can be effectively and efficiently examined.

II. Related Work

A. FPGA Power Modeling

Post-RTL Power Modeling: Post-RTL FPGA power modeling methods [17], [18], [19], [23], [24], [25] aim at expediting the vector-based power prediction process of FPGA accelerators. Early studies [23], [24], [25] focus on developing a detailed power library for atomic components (e.g., lookup tables (LUTs) or memory units). In an offline stage, the power of each component of interest is characterized by running microbenchmarks that fully exercise this component, and then a power macro model (i.e., a power LUT or regression model) is built up to correlate the component's microarchitecture and input data pattern to its real-time power. In an online stage, the power estimation of the holistic hardware design can be produced by deriving the power of each component from the library and then simply aggregating all components'

[1]For the sake of conciseness, we denote an application as a C/C++ program associated with a set of directive configurations.

power. These library-based methods require an extensive effort for power characterization, and are time consuming when performing online power computation for large-scale hardware designs that usually contain more than millions of components. In contrast, recent studies [17], [18], [19] seek to generate highly abstracted power models that enable direct power prediction of a holistic hardware design. These works identify and capture as features the RTL or gate-level signals whose activities are strongly related to power and generate a learning-based model, such as linear regression [17], [18] and decision tree-based ensemble models [19], to predict power of a target design at different time intervals. For each hardware design, a dedicated model should be built up, which cannot be used by other hardware designs. Even though these methods are applicable to the HLS design paradigm, their feature collection and model generation flow incurs large runtime overhead for the HLS design optimization process, in which a large amount of design instances should be evaluated.

Pre-RTL Power Modeling: Pre-RTL FPGA power modeling methods [20], [21], [26] aim to establish the direct relationship between the post-RTL power consumption and the pre-RTL hardware behaviors revealed by OpenCL or C/C++ programs, which speeds up the power prediction process by avoiding invoking the post-RTL electronic design automation (EDA) flow. The work [26] introduces a library-based power characterization method similar to the works [23], [24], [25] for the post-RTL stage, which induces considerable runtime overhead for offline power characterization and online power aggregation. The recent high-abstraction-level pre-RTL power modeling works [20], [21] are the closest to our work, which aim at providing power estimation for the holistic hardware designs at the OpenCL or C/C++ level. The work [20] targets power estimation of the OpenCL-to-FPGA programming model. Based on the fact that the OpenCL paradigm tends to show predictable behaviors in the form of phases, it decomposes the execution timeline of a kernel into work groups, and then further divides work groups into work items. The dynamic power model is generated according to these two phase levels. However, this work is restricted to the OpenCL programming model, which is not compatible with C-based HLS design flow. The work [21] can be applied to the HLS flow, but its scope is limited to a specialized type of C/C++ programs that can be restructured as affine functions (i.e., linear combination of variables plus a constant). Specifically, the work [21] identifies the basic code segment of an affine function as a tile, performs tile-based power characterization, and deduces overall power consumption of each hardware design by summing up power of all tiles instantiated by the hardware design.

Overall, the state-of-the-art FPGA power estimation methods either target post-RTL power estimation [17], [18], [19] that are not efficient enough for HLS, or focus on specific pre-RTL programming paradigms [20], [21] which are not applicable to general HLS design cases. It is worth noting that HL-Pow is the first work that overcomes the limits of accuracy, speed, and generalization ability for pre-RTL power estimation with FPGA HLS.

B. Design Space Exploration

Offline Modeling Paradigm: A rich body of research investigates automatic DSE for HLS. One direction of automatic DSE is to embrace an offline modeling paradigm. These methods establish predictive models offline and use brute-force search to retrieve an approximate Pareto frontier between two or more target metrics online. The works [20], [21] elaborated in Section II-A are instances that provide exhaustive DSE after the power models are developed for specific applications. Apart from power estimation, some works [2], [3], [4] evaluate the tradeoff between performance and area by producing a predictive model for fast and early estimation of these HLS metrics and then walking through a large number of design points to find optimal solutions. These methods are relatively fast in metric evaluation, but the traversal of a rich set of design points is still time consuming.

Online Modeling Paradigm: Another branch of DSE algorithms focuses on the online modeling paradigm. These methods first select a small portion of design points to feed into HLS in real time, distill domain knowledge from the HLS results, and build up predictive models to find out promising but unexplored design points for further evaluation. This procedure is repeated until optimal solutions converge finally. The commonly used DSE techniques of this type include: 1) meta-heuristics, such as genetic algorithm [5] and simulated annealing [6]; 2) domain-specific heuristics [7] that are tailored for the target problems; and 3) learning-based approaches [8], [9], [10] that generate surrogate models for QoR assessment in real time. These methods do not rely on a well-developed model for metric assessment in advance. However, these methods can only select a limited number of samples as promising candidates to put into ground-truth QoR evaluation and, thus, are not able to sufficiently understand all the characteristics of the dataset, which may give rise to suboptimal DSE results.

In this work, we propose a DSE framework that fuses the above offline and online modeling methods into an integration to get the best of both worlds. In an offline stage, we develop a transferable model for rapid power inference of HLS designs. In an online stage, we present a novel a priori-knowledge guided DSE algorithm to expedite the tradeoff between the latency and power with the aid of online design space sampling. These two stages are complementary to each other, and our method merges the advantages of the above two types of methods in an effort to collaboratively enhance the speed of performance-power co-optimization for FPGA HLS.

III. Preliminaries

A. Power Consumption

In general, the power consumption of hardware designs can be decomposed into static power [27] and dynamic power [28], which can be expressed by the following equation:

$$P_{\text{total}} = P_{\text{static}} + P_{\text{dyn}} = V_{dd} I_{\text{leakage}} + \sum_{i \in I} \alpha_i C_i V_{dd}^2 f. \quad (1)$$

From (1), we can see that the power is the sum of products of signal switching activity α_i, capacitance C_i on the net $i \in I$, supply voltage V_{dd}, and operating frequency f. The static power consumption is caused by reverse-bias leakage current I_{leakage} between the diffused regions and the substrates of transistors, irrespective of the workloads. For FPGA designs, the static power is dependent upon the utilization of different types of resources. In addition, the changes in process, voltage, and temperature (PVT) also affect the static power consumption.

In contrast, dynamic power consumption is introduced by signal transitions which dissipate power by repeatedly charging and discharging the load capacitors.

B. High-Level Synthesis

The HLS converts C/C++ programs in behavioral description into cycle-accurate RTL designs. The complete HLS design flow contains two main stages: 1) the front-end and 2) the back-end. In the HLS front-end stage, the C/C++ source code is first translated into intermediate representation (IR) using the LLVM [29] compiler. This provides an operation-level abstraction of the behavior-level programs. Some compiler-level optimizations are performed along with the IR generation process, such as bitwidth reduction, dead code elimination, common subexpression elimination, and loop unrolling.

After the front-end execution, the HLS back-end process first conducts control and data flow graph (CDFG) generation. The control flow graph (CFG) of the CDFG captures sequencing, conditional branching, and loop structures, while the data flow graph (DFG) of the CDFG describes behaviors of operations, i.e., data processing elements. Following this, three core phases, namely, *scheduling* [30], *allocation* [31], and *binding* [32], work collaboratively to complete the transformation from CDFGs to hardware architectures. Scheduling assigns each operation to a control step, i.e., an execution cycle. Allocation maps each operation to a concrete hardware component from a set of design variants. Binding associates each hardware component with a physical instance and establishes the interconnects between instances. Note that allocation is sometimes merged into scheduling or binding for unified optimization. Finally, synthesizable RTL code is generated using the output from the prior steps.

The off-the-shelf HLS tools usually offer various optimization strategies (i.e., so called directives or pragmas) to help designers tune the programs to improve the performance of loops and memory structures. Among the rich set of optimization strategies, the most crucial ones are *loop unrolling* [33], *loop pipelining* [34], and *array partitioning* [35]. Loop unrolling replicates the code inside a loop body k times with k being the unrolling factor, increasing the volume of data that can be processed in parallel. Loop pipelining allows loop iterations to overlap and execute concurrently. Array partitioning splits a holistic array into multiple smaller elements, each of which can be accessed independently, so as to boost the data bandwidth. These three important optimization strategies are all investigated in this article.

IV. Power Modeling Methodology

The execution of the complete HLS flow is time consuming when the design scale is large. We observe that indeed not every HLS stage, such as the RTL code generation, is able to produce power-related information. In this work, we aim to deliver power estimation *at the earliest design stage of FPGA HLS*, which avoids performing irrelevant and high-cost steps that postpone power estimation. To this end, instead of relying on the commercial HLS tools that do not allow interstage manipulation, we improve upon an open-source HLS toolkit, Light-HLS [36], trimming the HLS stages and preserving those that facilitate the power-related feature construction process.

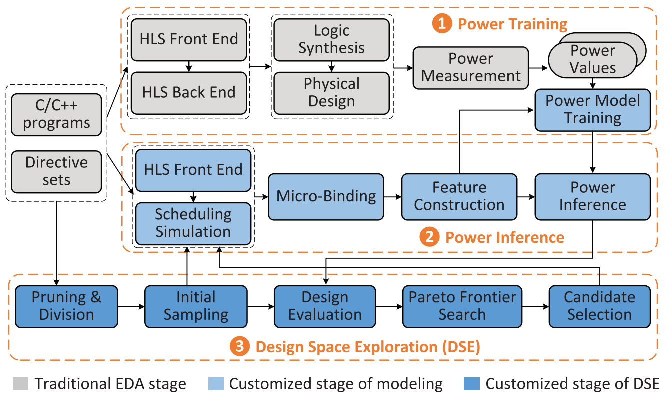


Fig. 2. Overall flow of HL-Pow. HL-Pow consists of three stages: power model training, power inference, and DSE.

Overall, the HL-Pow framework can be decomposed into three stages: 1) *power training stage:* train power model given a collection of applications, i.e., C/C++ programs along with a set of directive configurations; 2) *power inference stage:* predict power for a new application; and 3) *DSE stage:* search for the latency-power Pareto-optimal solutions of a given application. The complete design flow is depicted in Fig. 2.

In the first stage, i.e., training stage, a number of representative applications are used to generate training samples for power modeling, producing a number of design points that show significant variances in performance and resource utilization. These design points first pass through commercial HLS, logic synthesis, and physical design flows to generate gate-level netlists and bitstreams for onboard implementation, after which power measurement is performed to collect ground-truth power values. To develop a set of meaningful features for power learning in a pre-RTL stage such as HLS, we introduce an end-to-end feature construction flow consisting of portable HLS front end and scheduling [36], a simplified binding algorithm, and a feature construction stage. Putting it all together, the feature set and the corresponding ground-truth power consumption of each design point constitute a training sample. A collection of training samples derived from different applications is then used to build learning models capable of mapping from feature sets to power consumption.

In the second stage, i.e., power inference stage, HL-Pow can achieve fast and accurate power prediction for new applications of interest, based upon the pretrained power model. First, the new design points of the target application should pass through our proposed power inference flow to generate features in the same format as the training stage. Next, the constructed feature set is fed into the prebuilt model for power inference. In this inference stage, HL-Pow integrates a lightweight design flow as an alternative to the traditional EDA flow from HLS to RTL synthesis, physical design, gate-level power estimation/onboard measurement, which notably boosts the speed of power estimation.

The third stage, i.e., exploration stage, aims to selectively sample the design space and provide Pareto-optimal solutions between latency and power consumption in an efficient way, which is elaborated in the next section, i.e., Section V.

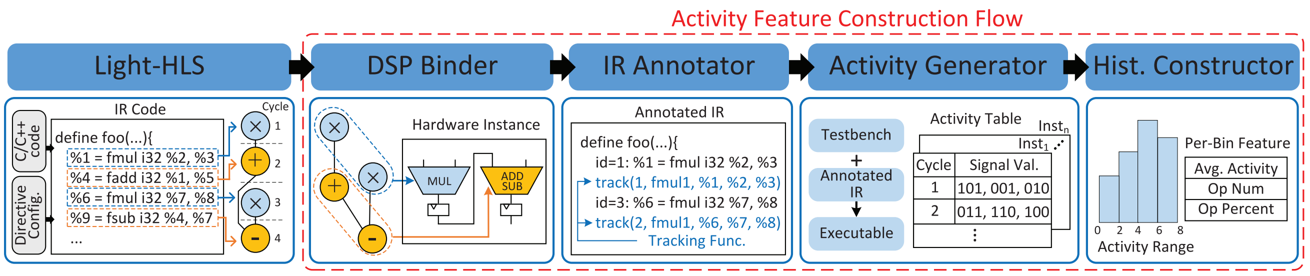


Fig. 3. Overview of activity feature construction flow. This flow constructs activity features for power inference from a high-level perspective using a fully customized design flow, avoiding the invocation of time-consuming back-end EDA process and gate-level simulation.

Our power modeling methodology is platform-independent and, thus, it can be used for various FPGA platforms. To set up a power model on a new FPGA platform, one needs to collect applications of interest, and follow the design flow in Fig. 2 to build up power models, after which the trained model can be used to directly predict power of unseen applications, provided that the power range of new applications is seen in the training datasets. Otherwise, new training samples that fall within the target power range should be used to incrementally update the power model.

In the remainder of this section, we first present our feature construction flow by illustrating how we identify and construct two important types of features, namely, *architectural features* and *activity features*. Thereafter, we introduce the model architectures and training strategies adopted in our power modeling flow.

A. Architectural Feature Construction

According to the constituents of power consumption, namely, static and dynamic power, as discussed in Section III-A, we propose to extract features responsible for static and dynamic power, respectively. Static power largely depends on the architectural information of the designs implemented on FPGA. Hence, we attempt to capture the utilization of all types of FPGA resources as architectural features, including LUT, block random access memory (BRAM), digital signal processing (DSP), and flip-flop (FF). Besides this, we also employ the *latency* as an overall design evaluation metric.

We obtain the aforementioned metrics using the open-source Light-HLS [36] tool instead of commercial HLS tools. Light-HLS is a light-weight HLS simulator that integrates a relatively complete HLS front-end stage and a portable scheduling algorithm as the back-end process. It is an appropriate substrate for our modeling task, because it provides users with fast QoR estimation of resource and latency, and allows us to fully manipulate the HLS process and remove the stages unnecessary in our flow. Herein, we propose scaling factor (SF) computation to augment the architectural features.

SF Computation: We compute the SFs of both the resources and latency as evaluation metrics for comparison of designs optimized with different directives. The SF is defined as follows:

$$SF = \frac{M_T}{M_B} \tag{2}$$

in which M represents one of the metrics (i.e., different types of resources or latency), T denotes the target design point derived from a C/C++ program and a directive configuration, and B denotes the baseline design point without applying any optimization. The SFs serve as a method to quantify the impact of different directive configurations on the same C/C++ program and offer a way to normalize the resource utilization and performance across different applications, which are useful features to reveal power, as stated in Section VI-C.

B. Activity Feature Construction

As shown in (1), dynamic power consumption is mainly introduced by the signal transitions that give rise to capacitance charging and discharging. As a result, signal activity is a first-order indicator of dynamic power. In the conventional power evaluation flow, the signal activities are extracted via gate-level simulation, which is cycle-accurate but the runtime overhead is considerably large, making it unfriendly toward early-stage power estimation. In this work, we propose to conduct fast signal activity extraction in the IR level, providing feedback as early as possible in the HLS stage and bypassing the expensive low-abstraction-level EDA flow and gate-level simulation. The full activity extraction flow of HL-Pow is depicted in Fig. 3. It comprises four fully customized components: 1) DSP binder; 2) IR annotator; 3) activity generator; and 4) histogram constructor.

DSP Binder: The activities of different components in the hardware design are critical contributors to dynamic power consumption. Based on this rationale, the DSP binder seeks to extract activities at a granularity of hardware instance instead of IR operation, so as to accurately account for dynamic power consumption from the hardware perspective. We note that Light-HLS only presents application-level resource estimation instead of operation-level binding. Nevertheless, operation-level binding information is necessary to aid in inferring accurate switching activities from IR operations to hardware instances by carefully taking the effect of hardware reuse into consideration. To this end, we propose a DSP binder, named microbinding, to perform a light-weight binding algorithm, aiming at mapping each DSP-related IR operation (e.g., floating-point arithmetic) to an exact hardware unit instance, i.e., RTL operation. The overall resource utilization of DSPs in Light-HLS is derived from an analytical model [2] that is close to optimal. On this basis, our binding task is reduced to associating each floating-point IR operation with a hardware unit instance, while ensuring that the total number of hardware unit instances of the same type of operations does not exceed the limit given by Light-HLS.

Algorithm 1: Microbinding Algorithm

```
input : Schedule information of the operations in basic blocks
        of the program
output: $\mathcal{B}$, instance-to-operation mapping set
for each basic block $b$ do
    for each DSP opcode $c$ do
        for each operation $op$ in $b_c$ do
            $t_{op} \leftarrow$ the execution cycle of $op$ in $b$
            success $\leftarrow$ false
            while not success do
                success $\leftarrow$ BindOperation($b$, $op$, $t_{op}$)
                $t_{op} \leftarrow t_{op} + 1$
Function BindOperation($b$, $op$, $t_{op}$):
    $inst \leftarrow$ FindReuseInstance($op$) // <case-1>
    if not HaveTimeConflict($b$, $op$, $t_{op}$, $inst$) then
        $\mathcal{B}_{c,inst} \leftarrow \mathcal{B}_{c,inst} \cup \{op\}$
        return true
    $inst \leftarrow$ FindIdleInstance($op$, $t_{op}$) // <case-2>
    if $inst$ is not None then
        $\mathcal{B}_{c,inst} \leftarrow \mathcal{B}_{c,inst} \cup \{op\}$
        return true
    return false // <case-3>
Function HaveTimeConflict($b$, $op$, $t_{op}$, $inst$):
    $II \leftarrow$ the initiation interval of $b$
    $\mathcal{T}_{inst} \leftarrow$ the execution cycles of $inst$
    if $II > 0$ then // pipelined
        for each time slot $t \in \mathcal{T}_{inst}$ do
            if $|t - t_{op}|$ % $II == 0$ then
                return true
    else // non-pipelined
        for each time slot $t \in \mathcal{T}_{inst}$ do
            if $t == t_{op}$ then
                return true
    return false
```

Microbinding is depicted in Algorithm 1. Herein, we mimic the principles of the binding stage in Vivado HLS and encompass three design principles. First, we greedily assign the same hardware unit instance to operations of the same type that share the same set of input but have different life time of execution cycles, which is denoted as *<case-1>* (lines 10–13) in the algorithm. This can help with the reduction of multiplexers and wiring to drive the input of instances in the created RTL designs, which is considered in off-the-shelf HLS tools. If this condition is not met, we assign the target operation an instance that has no conflict with it in all time slots, which is denoted as *<case-2>* (lines 14–17) in the algorithm. Second, if the resource limit cannot be met under the current binding scenario, i.e., *<case-3>* (line 18), we reschedule the execution time slot of the target operation to the next time slot, and re-examine the outcome again. This process is repeated until a valid binding result is achieved. The reason of implementing <case-3> is that the scheduling algorithm in Light-HLS aims at overall resource estimation and may aggressively schedule a large number of operations into a single cycle, leading to underoptimized solutions. Therefore, we introduce a refinement step to the scheduled results during operation-level binding. Third, when loop pipelining is employed, a new loop iteration starts at the IIth (i.e., initiation interval) stage of the prior iteration, instead of pending for the prior iteration to complete its execution. As a result, the ith stage of the kth iteration overlaps with the $(i + N \times II)$th stage of the $(k + N)$th iteration, with N being any positive integer. This effect is taken into account by the time conflict checking function (lines 19–30) in microbinding.

TABLE I
OPERATION TYPES AND IR OPCODES FOR ACTIVITY TRACKING

Operation Type	IR Opcode
arithmetic	add, sub, mul, div, sqrt, fadd, fsub, fmul, fdiv, fsqrt
logic	and, or, xor, icmp, fcmp
memory	store, load
arbitration	mux, select

IR Annotator: The IR annotator instruments the IR code with functions to keep track of the signal switching activities. First, we correlate signal switching activities to the input/output of the corresponding IR operations in HL-Pow, as signal transitions are primarily driven by the relevant operations. Second, we note that the power consumption is triggered by the hardware units that are physically present in the final design. Therefore, we identify four types of IR operations that can be cast to real hardware instances that significantly contribute to power consumption in datapaths and controlpaths (e.g., finite state machine) in the generated hardware, and mainly monitor their activities. These four operation types are arithmetic, logic, memory, and arbitration. The operation codes associated with different operation types are listed in Table I.

In addition, different IR operations can possibly contribute to the activities of an identical hardware unit instance in different execution cycles due to the effect of resource sharing. Based on the DSP binder, we instrument the IR code with an activity tracking function for each IR operation of interest, recording the values of its input/output signals and identify its assigned hardware unit instance. This IR annotator is developed within the LLVM tool chain [29]. Finally, an annotated IR code is generated after integrating the activity tracking functions.

Activity Generator: The activity generator makes use of the testbench of the target application and the annotated IR to produce signal activities for each hardware design instance. The testbench is provided by the users to explore power behaviors of typical workload they are interested in. It first compiles the given testbench, a library of activity tracking functions, and the annotated IR into object files separately, and then linked them together into a single executable file. Through running the executable file with the input vectors provided by the testbench, we are able to invoke the target kernel function in the IR, and capture the cycle-accurate input and output values for each hardware unit instance into a table.

We adopt *Hamming distance* [37] as the metric to quantify the switching activities between two consecutive execution cycles. Overall, Hamming distance counts the differences between two objects. In our context, the objects for comparison are activity vectors of the same size in a binary format and, thus, the Hamming distance computation can be expressed as

$$\mathrm{HD}(\boldsymbol{a}, \boldsymbol{b}) = \sum_{i=1}^{|\boldsymbol{a}|} |a_i - b_i|. \tag{3}$$

On this basis, we compute the average switching activity per hardware instance with

$$SA_{\mathrm{inst}} = \frac{\sum_{i=1}^{M_{\mathrm{inst}}} \sum_{j=1}^{N_{\mathrm{inst}}} HD(\boldsymbol{s}(i,j),\ \boldsymbol{s}(i,j-1))}{M_{\mathrm{inst}} \cdot N_{\mathrm{inst}}} \tag{4}$$

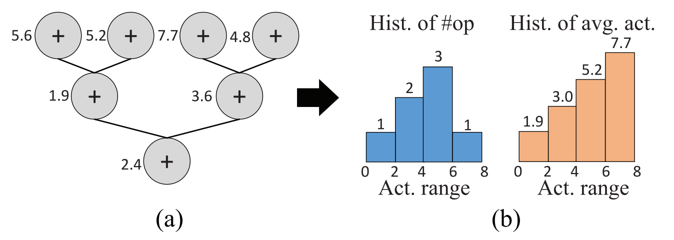


Fig. 4. Histogram representation of activity information, using opcode *add* of an adder tree as an example. (a) $\text{SA}_{\text{scaled}}$ of an adder tree. (b) Histograms of add: #ops & avg. act.

where $\boldsymbol{s}(i, j)$ is the bit vector for an operand or result i at time step j for the evaluated hardware instance inst, M_{inst} is the total number of operands and results, N_{inst} is the length of the list of activity vectors for inst, and $HD(\cdot)$ is the Hamming distance computation function in (3).

We further scale the average switching activity for each hardware instance as follows:

$$SA_{\text{scaled}} = \frac{N_{\text{inst}}}{L} \cdot SA_{\text{inst}} \tag{5}$$

where L is the latency of the target design point estimated by HLS. In this equation, (N_{inst}/L) can be regarded as an activation rate to amortize an instance's average switching activity over the total execution cycles.

Histogram Constructor: Histogram constructor establishes a *histogram representation* of switching activities as features, instead of simply using the raw activities per hardware design. As the combination of different C/C++ programs and directive configurations leads to different hardware realization that varies in the numbers of operations, the size of the activity set that can be obtained at this stage changes from sample to sample as a consequence. Nevertheless, this deviates from our goal of developing a one-size-fits-all power model that is applicable to diversified applications. Therefore, we apply an effective feature alignment technique in histogram constructor.

First, we construct a histogram per opcode listed in Table I, in which each histogram has a fixed number of bins covering a specific range of activities. Second, each hardware instance is mapped to the histogram corresponding to its opcode, and subsequently, each instance is distributed to a bin covering its switching activity, SA_{scaled}. Third, we derive two subhistograms from each histogram of opcode: a subhistogram to keep track of the number/percentage of instances per bin, and another subhistogram to compute average activities per bin. This technique is visualized in Fig. 4 with an adder tree as a toy example. In this way, we convert the operation-level activity information with varying size into histogram-level statistics that deliver a unified representation across different applications, which are then employed as activity features.

In addition to using the two subhistograms as operation-level activity features, we also take *the number*, *the sum*, and *average activities* of the hardware instances per opcode as design-level activity features.

We extensively investigate the per-bin activity coverage[2] ranging from 6.25% to 50% and we discover that, when the bin coverage is too small (i.e., 6.25%) or too large (i.e., 50%), the accuracy degrades. For a small activity coverage per bin, the model's generalization ability is harmed in that the model is more likely to be affected by noisy data in each bin. In contrast, using the bins with a large activity coverage means that the per-bin activity information is too coarse-grained to be learned effectively. A medium per-bin activity coverage (i.e., 12.5%–25%) benefits the model accuracy. Without loss of generality, we set the per-bin activity coverage to 25% in the remaining parts of this article.

[2]Per-bin activity coverage is the ratio of the activity range covered by a bin to the complete activity range.

C. Power Model Training

In the feature construction flow, HL-Pow produces a total number of 221 features, including 11 architectural features accounting for static power and 210 activity features indicative of dynamic power. To obtain ground-truth power values for each design point in the training stage, we conduct Vivado HLS and physical implementation flow, and collect real power measurement during onboard implementation. Besides onboard measurement, gate-level power estimation is another option to get ground-truth power values, whereas the accuracy may deviate from real measurement.

We build regression models for power prediction using a broad spectrum of supervised learning methods. These models are: 1) *linear regression*: classic linear regressors and Lasso regressors with a l_1-norm regularization term; 2) *support vector machine (SVM)*: support vector regressors with a radial basis function (RBF) kernel, a linear kernel or a polynomial kernel; 3) *tree-based model*: decision tree and ensemble models, including bagging trees, adaboost trees, random forests, and gradient boosting decision trees (GBDTs); and 4) *neural network*: multilayer perceptron (MLP) and convolutional neural network (CNN) models with 1-D kernels.

The feature set is a 1-D vector consisting of architectural features followed by activity histograms of opcodes in Table I, which allows the CNN models to mine the intrahistogram correlation of activity features, the major type of data correlation. We conduct *feature selection* to remove features with no variation across different samples, i.e., features that have the same value in all samples, reducing the number of features from 221 to 84. This can improve the feature quality and model performance. The power modeling task is formulated as a regression problem and evaluated by the mean average percentage error (MAPE)

$$\text{MAPE} = \frac{1}{N} \sum_{i}^{N} \frac{|\hat{p}_i - p_i|}{p_i} \times 100\% \tag{6}$$

where N, p, and $\hat{p}$ are the sample size, ground-truth power value, and predicted power, respectively.

The hyperparameters of models are listed as follows: 1) *Lasso regression*: regularization strength α in [0.1, 1]; 2) *SVM:* penalty-free tube ϵ in [0.1, 0.5]; 3) *tree-based model:* tree number in [200, 1600], tree depth in [4, 16], minimum number of samples to split a node or be at a leaf node in {2, 4, 8}; and 4) *neural network:* layer number in [2, 4], learning rate in [0.0001, 0.001], batch size in [32, 256], kernel size in [3, 5] for CNNs. We apply *tenfold cross-validation* to generate ten different validation sets for model evaluation, and the set of model hyperparameters that leads to the best average validation results is selected for power model construction. Finally, we average the inference results of the set of models

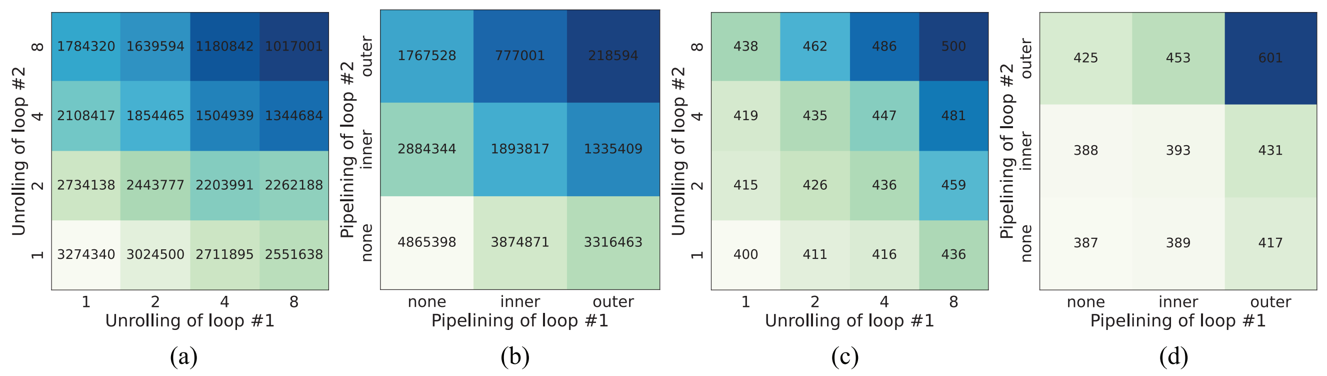


Fig. 5. Heatmaps of latency and power consumption for the application *k3mm* with two nested loops optimized with different directive configurations: (a) latency (#cycle) of different loop unrolling factors (1/2/4/8); (b) latency (#cycle) of different loop pipelining strategies (none/inner/outer loop pipelining); (c) power (mW) of different loop unrolling factors; and (d) power (mW) of different loop pipelining strategies.

presenting the highest accuracy in the cross validation, which serves as an effective ensemble strategy.

V. Design Space Exploration

With our power modeling framework, power prediction for a design point can be greatly expedited without running through the full HLS, physical implementation, and gate-level simulation/onboard measurement flow. However, when the goal is to find the Pareto-optimal points from a large design space, there is still a large runtime overhead to exhaustively assess power for a large amount of design points through HL-Pow power inference. To alleviate this issue, we propose a novel DSE algorithm to approximate the Pareto frontier between latency and power consumption by selectively sampling a small subset of the design points. The efficiency and efficacy of our DSE algorithm is guaranteed by: 1) the fast and accurate QoR estimation of HL-Pow and 2) the a priori domain knowledge that discloses the effect of pipelining and unrolling on both latency and power from real data derived from FPGA HLS and guides the DSE.

A. Observation

As for HLS, the joint effect of loop pipelining and unrolling on latency/power has not been clearly investigated. Herein, we derive two key observations from statistical analysis of a large spectrum of datasets to help elucidate this effect, which is then turned into a priori knowledge to navigate the sampling decision of Pareto-optimal points in our DSE framework. These two observations are elaborated as follows.

Observation 1: The latency/power consumption of HLS designs tends to be decreasing/increasing monotonically as the unrolling factor increases from one to maximum, or the pipelining option is changed from none, inner-loop to outer-loop pipelining.

Observation 2: The loop pipelining exerts a greater impact on latency and power consumption in comparison to loop unrolling.

Observation 1 is deduced from the fact that, by using a larger loop unrolling factor or a more aggressive loop pipelining scheme, the HLS designs are optimized with a higher extent of spatial or temporal parallelism, respectively. This makes it possible for more operations to be scheduled in a single execution cycle, finally leading to a reduction of latency along with a lift of power consumption. We verify the effect of observation 1 and demonstrate the results using a series of heat maps derived from real data, as shown in Fig. 5, in which the darker regions indicate higher performance (lower latency) and higher power consumption. Fig. 5 confirms that the trend of latency and power consumption regarding different directive configurations comply with observation 1.

As for observation 2, we propose to use the metric standard deviation reduction (SDR) [38] for illustration, which can be formulated as

$$\mathrm{SDR} = sd(T) - \sum_i \frac{|T_i|}{|T|} \times sd(T_i) \tag{7}$$

in which $sd(\cdot)$ is the standard deviation computation function, T is the complete dataset, and T_i denotes the ith subset of T split by the target attribute. In all, the SDR measures the ability that an attribute splits a dataset into subsets: the higher the SDR, the more homogeneous the data is in each split subset. This discloses that the changes of an attribute with higher SDR generally lead to larger variances in the evaluation metric of interest. Therefore, an attribute with higher SDR on a metric is of greater importance for that metric.

TABLE II
SDR of Latency and Power Over Loop Unrolling and Pipelining

Dataset	Design Size	SDR of Latency (#cycle)		SDR of Power (mW)	
		Unrolling	Pipelining	Unrolling	Pipelining
atax	4096	123094	446494	287	323
	16384	996343	3554805	355	406
bicg	4096	952	3371	58	72
	16384	3892	13442	73	95
gemm	4096	112710	546110	323	365
	16384	905284	4357477	460	528
gesummv	4096	927	4075	66	89
	16384	3776	16159	78	118
gemver	4096	1200	5446	93	108
	16384	4936	21741	97	119
k2mm	4096	198147	982741	454	513
	16384	1596195	7825513	97	188
k3mm	4096	147876	642265	289	308
	16384	1200884	5107215	83	107
mvt	4096	1101	3702	52	73
	16384	4490	14650	70	103
syrk	4096	85336	517751	219	340
	16384	681146	4135308	252	411
syr2k	4096	167731	1036941	356	494
	16384	1341633	8274238	369	552

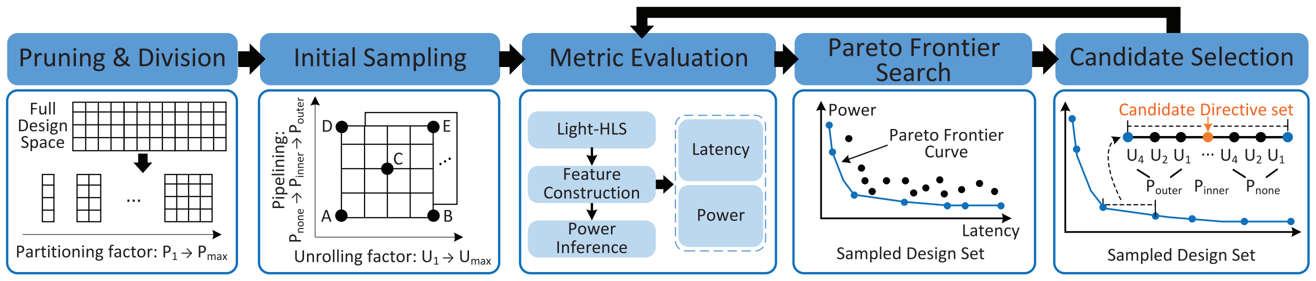


Fig. 6. Overall flow of the a priori-guided DSE algorithm.

In our circumstance, the objective is to compare the impact of loop unrolling/pipelining on latency/power by quantitatively evaluating how much these directives impact the datasets, which can be reduced to SDR computation. Correspondingly, the standard deviation is computed for the metrics latency and power, respectively. The attributes used for data splitting are either loop unrolling or pipelining with different configurations. Table II shows the SDR computed for different datasets. Overall, pipelining consistently outperforms unrolling in SDR, and eventually yields relative SDR improvement of 3.26–6.18× and 1.07–1.94× for latency and power, respectively. Based on this discovery, we can come to the conclusion that pipelining is a more effective and powerful strategy in tuning latency and power, compared with unrolling, which manifests our observation 2.

B. DSE Algorithm

We devise a five-stage DSE algorithm by taking advantage of the aforementioned two key observations distilled from real data, as depicted in Fig. 6.

Pruning and Division: We first perform a pruning and division stage to trim down the design space, and divide the design space into several regions to explore locally. The pruning is based on the phenomenon that when an outer loop is pipelined, all the inner loops are automatically unrolled [2]. In such a situation, regardless of what unrolling factors are set for the inner loops, the resulting architectures are the same as the one without unrolling the inner loops. Therefore, we reserve one valid design point and prune away the duplicate ones when this situation happens. Afterward, we split the design space into multiple regions by the *array-based directive*, namely, array partitioning, and use *loop-based directives*, including loop pipelining and loop unrolling for the search of promising points within each region.

Initial Sampling: Given the pruned and divided design space, an initial sampling stage is conducted to collect the first set of design points from the target application to assess. The heuristics is to select representative points in each region that are spreading out over the range of latency and power. We transform each region into a grid-like representation such that unrolling and pipelining constitute the x- and y-axis, respectively, and let the extent of parallelism increases along with the axes. According to *observation 1*, the trend of latency/power can be regarding as monotonically increasing along the x- or y-axis. On this basis, the design points in the corner and in the center of each grid are selected as the initial sampling set, in that they are most likely to demonstrate distinct values for latency and power. Without loss of generality, we show an

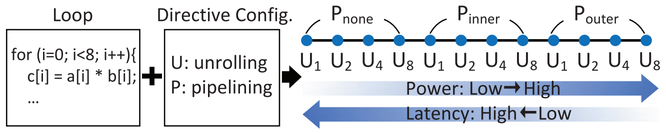


Fig. 7. Presort of design points regarding latency and power in a pipelining-then-unrolling priority order.

example with a two-level nested loop in stage 2 of Fig. 6. The corner and center point set $\{A, B, C, D, E\}$ is selected initially, yielding the directive configuration set of $\{P_{\text{none}} + U_1, P_{\text{none}} + U_{\max}, P_{\text{inner}} + U_{\max/2}, P_{\text{outer}} + U_1, P_{\text{outer}} + U_{\max}\}$.

Metric Evaluation and Pareto Frontier Search: In metric evaluation, the initial sampling set is fed into power inference of HL-Pow to evaluate power consumption. Likewise, the latency is simultaneously estimated by Light-HLS as a by-product. After obtaining both the estimated latency and power values, Pareto frontier search stage is performed to compute an approximate Pareto-optimal point set from the currently sampled dataset.

Candidate Selection: The candidate selection stage leverages the resulting approximate Pareto-optimal points as references for identifying promising candidate points to examine in the next round. Our heuristics for the candidate point selection is inspired by the *binary search* algorithm. That is, we attempt to iteratively search for the points that lie in the *center* of every two consecutive Pareto-optimal points from the same region to evaluate, which enables efficient traversal across the Pareto set.

To determine whether a point locates in the center of two consecutive Pareto-optimal points, we introduce a simple yet effective method to presort design points by loop-based directives with respect to latency and power. According to *observation 2*, pipelining leads to more pronounced changes in both latency and power in contrast to unrolling. Following this, we transform the design space between every two consecutive approximate Pareto-optimal points in a region into an *ordered sequence* by viewing pipelining as the primal factor and then unrolling, i.e., in a pipelining-then-unrolling priority order, as depicted in Fig. 7. In this way, we can regard the latency/power as monotonically decreasing/increasing from left to right, respectively. This case is verified with real datasets as shown in Fig. 8, where we showcase the changes of latency and power under different priority orders of unrolling and pipelining. Fig. 8(a) and (c) imply clear trends of latency and power in a pipelining-then-unrolling order, respectively, and vice versa, as shown in Fig. 8(b) and (d).

For each ordered sequence of design points between approximate Pareto-optimal points, we locate the center point and add

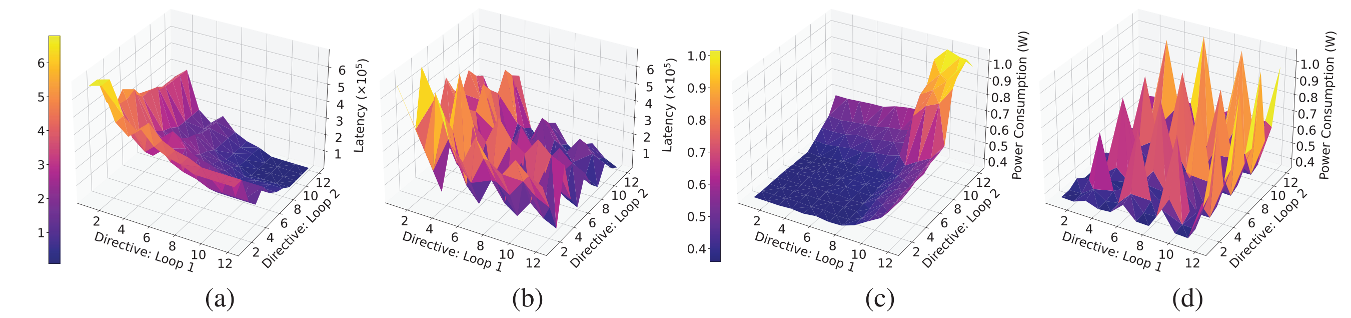


Fig. 8. Effect of different priority orders of unrolling/pipelining on latency/power for the application *bicg* with two nested loops: (a) latency (#cycle) in a pipelining-then-unrolling priority order; (b) latency (#cycle) in an unrolling-then-pipelining priority order; (c) power consumption (W) in a pipelining-then-unrolling priority order; and (d) power consumption (W) in an unrolling-then-pipelining priority order.

it to the sampling set. If this center point has already been added to the sampling set, we remove it from the sequence, and search for a new center point to add.

To summarize, the five stages shown in Fig. 6 are iterated to search for promising design points until a user-defined budget of HL-Pow runs is reached or no more candidates exist. Finally, to ensure that the real Pareto-optimal points are not pruned away due to the error induced by power modeling, we allow the design points within a certain deviation range from the nearest Pareto-optimal points to be incorporated into the candidate Pareto set. According to the experiments in Section VI-B, we set the deviation range to be within ±5% of the power of the approximate Pareto-optimal points, which is close to the power estimation error.

VI. Experimental Results

A. Experimental Setup

The HL-Pow design flow is fully automated and implemented with Python and C++ for feature construction, model establishment, and power inference. The conventional and deep learning machine learning models are implemented in Scikit-learn [39], XGBoost [40], and Pytorch [41], respectively. We apply our design flow to evaluate up to ten representative benchmarks from different design categories in Polybench [42], each of which produces multiple design points. The design points are synthesized using floating-point arithmetic and implemented under a timing constraint of 10 ns. To obtain ground truth power values, we pass the design points through the complete EDA flow from HLS to bitstream generation using AMD Xilinx Vivado 2018.2 toolkit. Then, we implement all the design points on an AMD Xilinx Ultrascale+ ZCU102 FPGA board and collect real power consumption through onboard measurement using the Power Advantage Tool [43].

For each benchmark, we generate design points with the design sizes of 4096 and 16 384, resulting in 20 unique datasets for model development and assessment. We uniformly optimize the datasets using cyclic array partitioning on {the last dimension, the last two dimensions} with factors in {1, 2, 4, 8}, loop pipelining with configurations in {none, inner, outer}, and loop unrolling with factors in {1, 2, 4, 8}. The loop-based directives are applied to the inner-most two levels of nested loops. This creates a design space of 32 768 design points. However, to keep the design space compact, we simply remove the design points with the loop unrolling factor larger

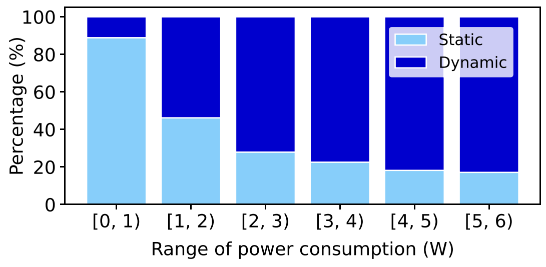


Fig. 9. Breakdown of total power consumption.

than the array partitioning factor, which causes memory conflicts and hampers optimization. Finally, this produces a design space of 4672 effective design points per dataset. To construct our dataset for modeling, we uniformly sample 531 design points from the design space for each dataset, producing a total number of up to 10 478 samples.

The power range of the datasets falls within 0.3–5.3 W and the breakdown of power in different power ranges is shown in Fig. 9. We can see that the static power dominates in the low power range (i.e., 0–2 W), while the dynamic power becomes the main source of total power in the high-power range (i.e., 3–5.3 W). It is conceivable that both the static and dynamic power are fundamental constituents that should be carefully taken into account in power modeling.

B. Accuracy and Runtime of Power Modeling

In this experiment, we examine the power estimation accuracy and the runtime speedup of HL-Pow over off-the-shelf methods. During model training, we adopt a *leave-one-out* scheme: a benchmark dataset with both the design sizes is left out of the complete benchmark set and served as the test set only, while the others are used for training. The training process is carried out in multiple rounds, with each round selecting a complete dataset as the test set. This allows us to make full usage of every dataset except the current test set to construct learning models, while ensuring that the pretrained model is not biased toward the given test set. To evaluate the efficacy of our feature construction method via our customized design flow, we compare our method to two variants with higher feature fidelity: 1) *w/ HLS* (i.e., our preliminary work [22]): replacing the light-weight scheduling and binding algorithms in Fig. 3 with the exact counterparts of a realistic HLS flow, i.e., Vivado HLS, so that accurate latency and activity histograms can be obtained in feature construction and

TABLE III
MAPE (%) of Different Categories of Models for HL-Pow and Variants With Different Feature Fidelity

Dataset	Design Size	Linear			SVM			CNN			GBDT		
		HL-Pow	w/ HLS	w/ Impl.	HL-Pow	w/ HLS	w/ Impl.	HL-Pow	w/ HLS	w/ Impl.	HL-Pow	w/ HLS	w/ Impl.
atax	4096	7.22	7.28	6.07	7.37	6.54	5.95	4.51	3.44	4.53	6.23	8.83	8.88
	16384	5.38	6.48	5.59	5.23	4.95	5.00	5.80	3.87	3.36	5.80	3.67	3.92
bicg	4096	5.44	4.80	3.84	4.14	3.91	3.44	3.16	3.17	2.76	2.48	2.30	2.29
	16384	3.70	3.27	3.30	3.56	3.07	3.25	5.07	4.65	4.25	3.51	3.46	3.07
gemm	4096	7.16	7.33	6.00	6.24	6.10	5.37	5.28	4.57	4.52	4.63	4.13	3.85
	16384	10.18	8.53	5.69	7.79	8.17	5.80	6.19	5.16	4.87	6.11	4.80	4.18
gesummv	4096	7.00	7.00	6.68	6.45	5.68	5.27	3.76	3.87	3.89	4.31	3.42	3.70
	16384	8.62	14.81	12.82	8.51	12.82	12.42	7.97	6.81	7.97	7.58	8.06	8.07
gemver	4096	11.30	7.51	6.73	6.56	7.55	8.04	6.06	7.03	6.05	5.89	5.44	4.68
	16384	9.02	9.01	8.87	7.25	6.03	6.11	9.28	9.88	10.38	6.40	4.81	4.08
k2mm	4096	6.46	4.20	4.31	6.54	4.68	4.16	5.00	4.17	3.50	4.88	4.04	3.41
	16384	7.56	10.03	8.03	7.73	9.61	8.31	8.89	8.83	8.74	6.96	8.22	6.85
k3mm	4096	7.11	5.81	5.16	5.41	5.06	4.85	5.47	4.48	5.32	4.75	4.55	4.02
	16384	8.78	9.84	6.59	6.49	8.67	7.83	9.60	6.10	8.37	4.31	3.28	3.22
mvt	4096	4.87	4.24	3.70	3.63	3.14	3.00	1.97	1.89	1.78	1.76	1.91	1.90
	16384	4.64	5.12	4.05	4.08	5.57	4.64	2.86	2.85	2.96	3.59	3.80	3.27
syrk	4096	4.44	5.97	4.58	4.53	6.41	4.88	3.88	3.00	2.52	3.67	3.65	3.42
	16384	4.54	6.90	4.94	4.70	6.08	3.95	4.29	3.53	2.83	3.48	3.38	2.84
syr2k	4096	5.13	7.30	4.77	5.20	5.33	4.18	5.07	4.27	3.62	4.09	3.25	3.22
	16384	7.80	9.88	7.53	6.52	7.27	6.01	7.80	9.06	8.47	6.20	7.15	5.34
average	–	6.80	7.23	5.94	5.88	6.30	5.60	5.55	4.99	4.99	**4.82**	**4.59**	**4.20**

TABLE IV
Breakdown of Average Runtime (s) Per Sample With HL-Pow and the Total Runtime of Baseline Methods

Dataset	Design Size	HL-Pow			w/ HLS	Onboard Meas.
		Cust. HLS	Feat. Gen.	Inference		
atax	4096	5.6	2.0	0.7	53.9	1279.1
	16384	31.3	6.1	0.7	203.7	1831.5
bicg	4096	5.6	1.2	0.6	25.0	1394.0
	16384	31.2	0.2	0.6	59.9	1688.8
gemm	4096	9.0	2.4	0.6	263.4	1599.0
	16384	46.3	7.2	0.6	1243.4	3087.5
gesummv	4096	6.0	1.4	0.7	26.7	1316.8
	16384	31.3	0.3	0.7	53.1	1719.6
gemver	4096	24.3	3.0	0.7	278.5	1905.1
	16384	102.4	0.3	0.6	1129.1	2982.3
k2mm	4096	14.1	5.2	0.6	106.9	1467.7
	16384	66.8	13.6	0.7	205.6	2403.0
k3mm	4096	13.8	5.8	0.6	109.4	1493.7
	16384	64.7	19.0	0.6	232.2	2708.9
mvt	4096	5.7	1.3	0.6	24.0	1463.7
	16384	31.4	0.2	0.6	44.4	1818.8
syrk	4096	9.0	3.2	0.7	243.3	1636.5
	16384	43.0	7.1	0.6	1074.5	3383.0
syr2k	4096	17.7	5.4	0.7	111.9	1518.3
	16384	97.1	14.7	0.8	375.1	2729.6

Runtime: (1) *HL-Pow*: customized HLS + feature generation + power inference. (2) *w/ HLS*: Vivado HLS + feature generation + power inference. (3) *Onboard Meas.*: Vivado HLS + logic synthesis + placement & routing + bitstream generation + onboard measurement.

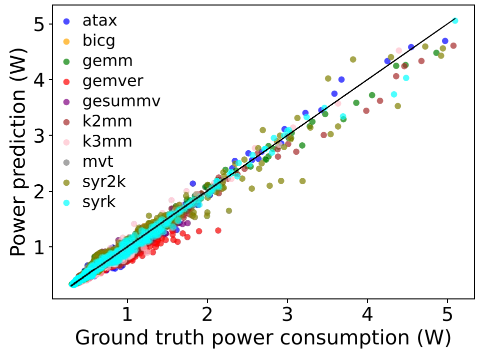


Fig. 10. HL-Pow power estimation versus onboard measurement.

2) *w/ Impl.*: based upon *w/ HLS*, further getting exact resource utilization from Vivado's physical implementation to calibrate the relevant approximate architectural features (i.e, resource utilization and the corresponding SFs). Table III reports the MAPE [see (6)] of the best models from the four model categories, as demonstrated in Section IV-C.

As shown in Table III, the best overall accuracy of HL-Pow is achieved by the GBDT models, with an average error of 4.82% from the actual measurement. The two variants *w/HLS* and *w/Impl.* demonstrate slightly higher accuracy than HL-Pow, reducing the error to 4.59% and 4.20%, respectively. This improvement stems from the adoption of more accurate features through real HLS and physical implementation, but at the cost of much larger execution time, as shown in Table IV. As for the linear and SVM models, it is interesting to note that the accuracy of HL-Pow exceeds that of *w/HLS*. We attribute this phenomenon to the joint effects of the underfitting issue caused by the simplicity of these models and the uncertainty of modeling due to noise in the datasets. Overall, from the perspective of the best models, the prediction accuracy of HL-Pow is on par with that of the two variants with higher fidelity of features, which confirms that HL-Pow is able to preserve high feature quality by effectively approximating the intrinsic mechanism of HLS. In addition, HL-Pow shows strong scalability for different design sizes.

Fig. 10 showcases the difference between the prediction results of HL-Pow for every design point. A more severe deviation from the line $y = x$ means a larger error induced. From Fig. 10, we can see that HL-Pow can accurately capture the power behaviors across the power ranges of 0.3–5.3 W. The average error of HL-Pow within the power range of 4–5.3 W is 7%, which is slightly higher than the average. We conjecture this situation can be improved by enlarging the sample size for this power range.

Furthermore, HL-Pow demonstrates a speedup of 1.4–23$\times$ (7.8$\times$ on avg.) over *w/HLS*, and *w/HLS* further reveals a speedup of 2.5–61$\times$ (20$\times$ on avg.) over onboard measurement. Finally, HL-Pow reaps a compound speedup of 24–190$\times$ (84$\times$ on avg.) over onboard measurement, which can be deduced from Table IV. This notable gain is attributed to: 1) the employment of the fully customized and tailored HLS flow and 2) the one-size-fits-all learning model for direct power inference without the need of conducting any post-RTL EDA flow.

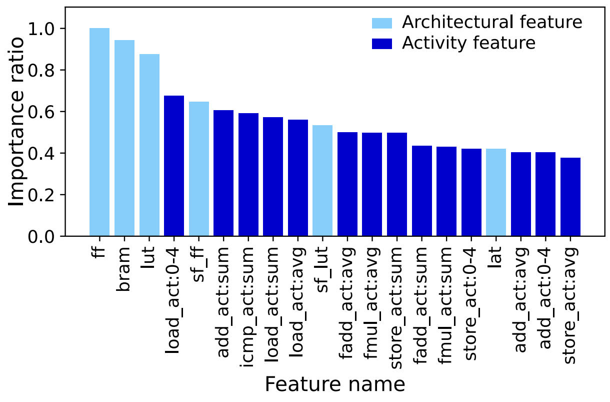


Fig. 11. Top 20 most important features in HL-Pow.

TABLE V
Estimation Error (%) of HL-Pow on Applications With Unseen Characteristics

Dataset	Linear	SVM	CNN	GBDT
md-knn	10.13	6.60	7.92	6.49
fft-strided	7.78	21.94	6.36	7.13
gemm-ncubed	16.25	13.45	4.10	5.79
spmv-ellpack	12.22	15.45	6.60	2.32
stencil-2d	7.02	23.02	10.43	7.44
average	10.68	16.09	7.08	**5.83**

C. Feature Importance

In this experiment, we investigate the validity of features that are captured by HL-Pow. We adopt the Gini feature importance as the evaluation metric, which is computed as the normalized total reduction of the average error brought by the target feature during tree growing. For each feature, we calculate the sum of its importance across the GBDT models developed with different test cases. Fig. 11 visualizes the top 20 important features in HL-Pow after normalization. We can observe that the architectural features are more prominent in the magnitude of importance, whereas the activity features are more dominant in the occupancy rate of the top 20 features, from which it is natural to infer that both of these two types of features are crucial in power modeling. In addition, the SF features (denoted with the prefix sf_ in Fig. 11) have high ranks in feature importance. Therefore, they are proven to be informative and crucial in the power modeling. In all, this experiment justifies the effectiveness of our proposed feature construction method.

D. Generalization Ability Toward Diversified Applications

In this experiment, we further study the generalization ability of HL-Pow. We use HL-Pow for direct power prediction of five representative MachSuite [44] datasets that have distinctly different characteristics from the training sets in terms of loop patterns, array manipulations, and design sizes. We directly feed the features of these new datasets into the pretrained models built in Section VI-B, and take the average as the final estimate. As shown in Table V, the best average error is 5.83%, which is derived from the GBDT model. This high-prediction accuracy (i.e., low error) proves the high-generalization ability of HL-Pow to adapt to applications from a wide spectrum of domains without the need to rebuild the power model.

E. QoR of DSE Algorithm

In this experiment, we assess the performance of our proposed a priori-guided DSE algorithm described in Section V. The predictive power models in Section VI-B are integrated into the metric evaluation stage of our DSE algorithm, each of which only provides the QoR estimation for the unseen datasets in their training stage. Experiments are conducted for the datasets with a design size of 4096. We evaluate the QoR of the DSE algorithm using the average distance from reference set (ADRS). ADRS is a metric to quantify the difference between the approximate and the exact Pareto sets, which can be formulated as

$$\mathrm{ADRS}\left(\hat{S}, S\right) = \left[\frac{1}{|S|}\sum_{s\in S}\min_{\hat{s}\in\hat{S}}\left(\delta\left(\hat{s}, s\right)\right)\right] \times 100\%$$

$$\delta\left(\hat{s}, s\right) = \max\left\{0, \frac{l\left(\hat{s}\right) - l(s)}{l(s)}, \frac{p\left(\hat{s}\right) - p(s)}{p(s)}\right\} \quad (8)$$

where $\hat{S}$ is the approximate Pareto set, S is the exact Pareto set, and $l(\cdot)$ and $p(\cdot)$ denote functions to retrieve the latency and power, respectively. The lower the ADRS, the smaller the gap between the approximate Pareto set and the exact one.

First, we study how different initial sampling rates impact the quality of the DSE algorithm, as shown in Fig. 12(a) and (b). On the one hand, Fig. 12(a) reveals that the ADRS starts to drop rapidly as the initial sampling rate is lifted, but the results come out to be stable from a certain initial sampling rate. On the other hand, Fig. 12(b) shows that when the initial sampling rate keeps increasing to a large portion of the total sampling budget, the ADRS in turn degrades due to a lack of budget for iterative searching of candidate points. Putting it all together, it is essential to keep a good balance of the sampling proportion between the initial sampling and iterative searching in an effort to enable high QoR of the DSE algorithm. Overall, an initial sampling rate between 0.75% and 3% yields desirable QoR for a wide range of total sampling budgets, which demonstrates the high adaptability of our DSE algorithm to sampling rates.

Second, we examine the ADRS and runtime of the DSE algorithm given different total sampling budgets. As shown in Fig. 12(c), the results demonstrate a good convergence trend, which validates the generalization ability of our DSE algorithm in handling diverse applications. In particular, the average ADRS for the total sampling budgets of 4%, 5%, 6%, and 7% are 1.79%, 1.33%, 1.04%, and 0.85%, respectively. From Fig. 12(d), we can see that the runtime of DSE is linearly related to the total sampling budget. This reveals that our DSE algorithm is able to traverse the design space uniformly under different sampling budgets.

Third, we compare our DSE algorithm with various methods, including using ground-truth power measurement (*GTM*) as a replacement of the power estimation in the DSE algorithm, so as to investigate how the accuracy of power estimation influences the effectiveness of the DSE algorithm and combining our predictive model with two widely used DSE algorithms for FPGA HLS, i.e., one with Bayesian optimization (BO) [10] and the other with genetic algorithm (GA) [5], which are developed with the tools OpenBox [45] and pymoo [46], respectively, in this article. We run these baseline methods with a total sampling budget of 4%. Results are shown in Table VI.

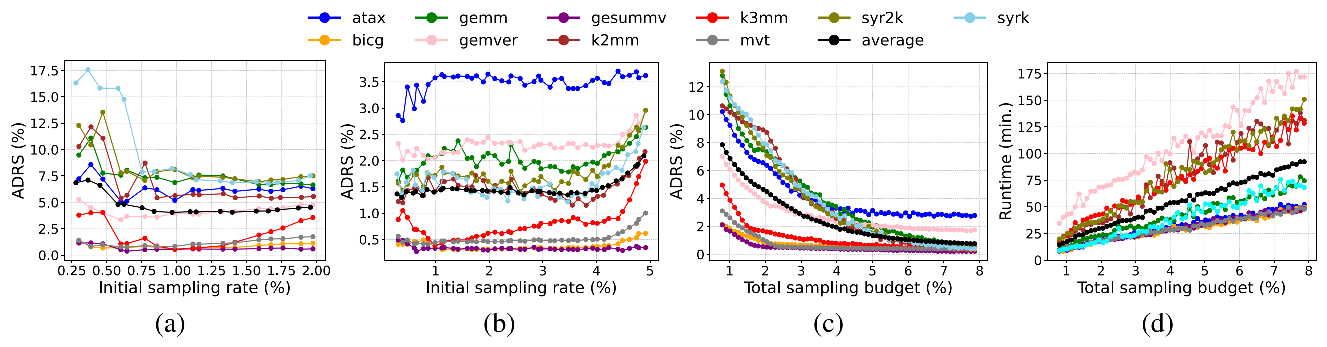


Fig. 12. QoR of DSE: (a) ADRS with different initial sampling rates given a total sampling budget of 2%; (b) ADRS with different initial sampling rates given a total sampling budget of 5%; (c) ADRS with different total sampling budgets given an initial sampling rate of 0.75%; and (d) runtime with different total sampling budgets given an initial sampling rate of 0.75%.

TABLE VI
ADRS AND RUNTIME OF VARIOUS DSE METHODS

Dataset	ADRS (%)				Runtime Speedup			
	GTM	BO	GA	HL-Pow	GTM	BO	GA	HL-Pow
atax	1.51	9.58	5.33	3.26	1	25	196	115
bicg	0.38	3.54	3.59	0.55	1	27	205	137
gemm	2.18	7.93	3.53	2.89	1	32	164	134
gesummv	0.29	3.98	3.11	0.37	1	28	205	141
gemver	1.39	4.73	5.71	2.34	1	39	86	55
k2mm	2.13	7.71	3.45	2.48	1	23	70	53
k3mm	0.33	2.24	2.64	0.75	1	20	68	66
mvt	0.54	4.92	3.66	0.45	1	36	207	156
syrk	2.19	6.78	4.72	2.56	1	28	171	126
syr2k	1.79	9.23	8.20	2.23	1	23	85	73
average	1.27	6.06	4.39	1.79	1	28	146	105

As for DSE quality, GTM produces the best ADRS, and HL-Pow exhibits comparable performance with only a small drop of 0.52%, significantly outperforming BO and GA. This fully justifies the value of our pre-RTL power modeling method in assisting early design optimization. In terms of runtime, both HL-Pow and GA yield remarkable speedup over GTM and BO. The HL-Pow is slightly slower than GA in that HL-Pow spends more time evaluating more complex design points that stand a larger chance of improving ADRS. Overall, the DSE of HL-Pow demonstrates superiority in both ADRS and speed over existing DSE methods.

VII. Conclusion

This article presented a learning-based power modeling and optimization framework for FPGA HLS, named HL-Pow. HL-Pow achieves an average power prediction error of 4.82% with a speedup of up to 24–190×, and an error of 5.83% for applications in domains different from the training sets. The DSE engine delivers an ADRS of 1.79% by only sampling 4% of design points in large-scale design spaces.

References

[1] P. Coussy and A. Morawiec, *High-Level Synthesis: From Algorithm to Digital Circuit*. Berlin, Germany: Springer, 2008.

[2] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He, “Performance modeling and directives optimization for high-level synthesis on FPGA,” *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 7, pp. 1428–1441, Jul. 2020.

[3] N. Wu, Y. Xie, and C. Hao, “IronMan-pro: Multi-objective design space exploration in HLS via reinforcement learning and graph neural network based modeling,” *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 42, no. 3, pp. 900–913, Mar. 2023.

[4] A. Sohrabizadeh, Y. Bai, Y. Sun, and J. Cong, “Automated accelerator optimization aided by graph neural networks,” in *Proc. ACM/IEEE Design Autom. Conf. (DAC)*, 2022, pp. 55–60.

[5] F. Ferrandi, P. L. Lanzi, D. Loiacono, C. Pilato, and D. Sciuto, “A multi-objective genetic algorithm for design space exploration in high-level synthesis,” in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI (ISVLSI)*, 2008, pp. 417–422.

[6] B. C. Schafer, T. Takenaka, and K. Wakabayashi, “Adaptive simulated annealer for high level synthesis design space exploration,” in *Proc. Int. Symp. VLSI Design, Autom. Test (VLSI-DAT)*, 2009, pp. 106–109.

[7] L. Ferretti, G. Ansaloni, and L. Pozzi, “Lattice-traversing design space exploration for high level synthesis,” in *Proc. IEEE Int. Conf. Comput. Design (ICCD)*, 2018, pp. 210–217.

[8] H.-Y. Liu and L. P. Carloni, “On learning-based methods for design-space exploration with high-level synthesis,” in *Proc. ACM/IEEE Design Autom. Conf. (DAC)*, 2013, pp. 1–7.

[9] Q. Sun et al., “Correlated multi-objective multi-fidelity optimization for HLS directives design,” *ACM Trans. Design Autom. Electron. Syst.*, vol. 27, no. 4, pp. 1–27, Mar. 2022.

[10] A. Mehrabi, A. Manocha, B. C. Lee, and D. J. Sorin, “Prospector: Synthesizing efficient accelerators via statistical learning,” in *Proc. Design, Autom. Test Europe Conf. Exhibit. (DATE)*, 2020, pp. 151–156.

[11] “Vivado design suite user guide: High level synthesis,” Xilinx Ltd, San Jose, CA, USA, White Paper, 2017.

[12] *Intel HLS Compiler Product Brief*, Intel, Santa Clara, CA, USA, 2017.

[13] A. Canis et al., “LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems,” *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 2, p. 24, Sep. 2013.

[14] D. Liu and C. Svensson, “Power consumption estimation in CMOS VLSI chips,” *IEEE J. Solid-State Circuits*, vol. 29, no. 6, pp. 663–670, Jun. 1994.

[15] J. H. Anderson and F. N. Najm, “Power estimation techniques for FPGAs,” *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 12, no. 10, pp. 1015–1027, Oct. 2004.

[16] H. A. Hassan, M. Anis, and M. Elmasry, “Total power modeling in FPGAs under spatial correlation,” *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 17, no. 4, pp. 578–582, Apr. 2009.

[17] M. Najem, P. Benoit, F. Bruguier, G. Sassatelli, and L. Torres, “Method for dynamic power monitoring on FPGAs,” in *Proc. Int. Conf. Field Programmable Logic Appl. (FPL)*, 2014, pp. 1–6.

[18] J. J. Davis, E. Hung, J. M. Levine, E. A. Stott, P. Y. K. Cheung, and G. A. Constantinides, “KAPow: High-accuracy, low-overhead online per-module power estimation for FPGA designs,” *ACM Trans. Reconfig. Technol. Syst.*, vol. 11, no. 1, p. 2, Jan. 2018.

[19] Z. Lin, S. Sinha, and W. Zhang, “An ensemble learning approach for in-situ monitoring of FPGA dynamic power,” *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 9, pp. 1661–1674, Sep. 2019.

[20] Y. Liang, S. Wang, and W. Zhang, “FlexCL: A model of performance and power for OpenCL workloads on FPGAs,” *IEEE Trans. Comput.*, vol. 67, no. 12, pp. 1750–1764, Dec. 2018.

[21] W. Zuo et al., “A polyhedral-based SystemC modeling and generation framework for effective low-power design space exploration,” in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, 2015, pp. 357–364.

[22] Z. Lin, J. Zhao, S. Sinha, and W. Zhang, "HL-Pow: A learning-based power modeling framework for high-level synthesis," in *Proc. Asia–South Pacific Design Autom. Conf. (ASP-DAC)*, 2020, pp. 574–580.

[23] F. Li, D. Chen, L. He, and J. Cong, "Architecture evaluation for power-efficient FPGAs," in *Proc. ACM/SIGDA 11th Int. Symp. Field Programmable Gate Arrays (FPGA)*, 2003, pp. 175–184.

[24] D. Chen, J. Cong, Y. Fan, and L. Wan, "LOPASS: A low-power architectural synthesis system for FPGAs with interconnect estimation and optimization," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 18, no. 4, pp. 564–577, Apr. 2010.

[25] H. Liang, Y.-C. Chen, T. Luo, W. Zhang, H. Li, and B. He, "Hierarchical library based power estimator for versatile FPGAs," in *Proc. IEEE Int. Symp. Embedded Multicore/Many-Core Syst.-Chip (MCSoC)*, 2015, pp. 25–32.

[26] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, "Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *Proc. ACM/IEEE 41st Int. Symp. Comput. Archit. (ISCA)*, 2014, pp. 97–108.

[27] H. J. M. Veendrick, "Short-circuit dissipation of static CMOS circuitry and its impact on the design of buffer circuits," *IEEE J. Solid-State Circuits*, vol. 19, no. 4, pp. 468–473, Aug. 1984.

[28] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen, "Low-power CMOS digital design," *IEEE J. Solid-State Circuits*, vol. 27, no. 4, pp. 473–484, Apr. 1992.

[29] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Gener. Optim. (CGO)*, 2004, pp. 75–86.

[30] J. Cong and Z. Zhang, "An efficient and versatile scheduling algorithm based on SDC formulation," in *Proc. ACM/IEEE Design Autom. Conf. (DAC)*, 2006, pp. 433–438.

[31] J. Cong, P. Zhang, and Y. Zou, "Optimizing memory hierarchy allocation with loop transformations for high-level synthesis," in *Proc. ACM/IEEE Design Autom. Conf. (DAC)*, 2012, pp. 1229–1234.

[32] C. Mandal, P. P. Chakrabarti, and S. Ghose, "GABIND: A GA approach to allocation and binding for the high-level synthesis of data paths," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 8, no. 6, pp. 747–750, Dec. 2000.

[33] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Amsterdam, The Netherlands: Elsevier, 2011.

[34] M. Lam, "Software pipelining: An effective scheduling technique for VLIW machines," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, vol. 23, Jun. 1988, pp. 318–328.

[35] Y. Wang, P. Li, P. Zhang, C. Zhang, and J. Cong, "Memory partitioning for multidimensional arrays in high-level synthesis," in *Proc. ACM/IEEE Design Autom. Conf. (DAC)*, 2013, pp. 1–8.

[36] T. Liang, J. Zhao, L. Feng, S. Sinha, and W. Zhang, "Hi-ClockFlow: Multi-clock dataflow automation and throughput optimization in high-level synthesis," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, 2019, pp. 1–6.

[37] R. W. Hamming, "Error detecting and error correcting codes," *Bell Syst. Tech. J.*, vol. 29, no. 2, pp. 147–160, 1950.

[38] J. R. Quinlan, "Learning with continuous classes," in *Proc. Aust. Joint Conf. Artif. Intell.*, vol. 92, 1992, pp. 343–348.

[39] F. Pedregosa et al., "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, no. 85, pp. 2825–2830, 2011.

[40] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proc. ACM SIGKDD Int. Conf. Knowl. Disc. Data Min. (KDD)*, 2016, pp. 785–794.

[41] A. Paszke et al., "PyTorch: An imperative style, high-performance deep learning library," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, vol. 32, 2019, p. 721.

[42] L.-N. Pouchet. "Polybench: The polyhedral benchmark suite." 2012. [Online]. Available: http://www.cs.ucla.ed%7Epouchet/software/polybench

[43] *Zynq UltraScale+ MPSoC Power Advantage Tool 2018.1*, Xilinx, San Jose, CA, USA, 2018.

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

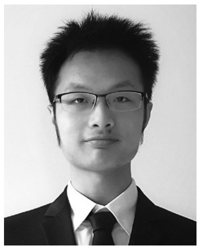
[45] Y. Li et al., "OpenBox: A generalized black-box optimization service," in *Proc. ACM SIGKDD Conf. Knowl. Disc. Data Min. (KDD)*, 2021, pp. 3209–3219.

[46] J. Blank and K. Deb, "Pymoo: Multi-objective optimization in Python," *IEEE Access*, vol. 8, pp. 89497–89509, 2020.

Zhe Lin (Member, IEEE) received the B.S. degree from the School of Electronic Science and Engineering, Southeast University, Nanjing, China, in 2014, and the Ph.D. degree from the Department of Electronic and Computer Engineering, The Hong Kong University of Science and Technology, Hong Kong, in 2019.

He is an Assistant Professor with the School of Integrated Circuits, Sun Yat-sen University, Shenzhen, China. Previously, he was an Associate Research Fellow with Peng Cheng Laboratory, Shenzhen.

Dr. Lin was a recipient of the Best Paper Award Nominations at DATE 2022 and FCCM 2019.



Tingyuan Liang (Student Member, IEEE) received the B.S. degree in electrical and information engineering from Zhejiang University, Hangzhou, China, in 2017. He is currently pursuing the Ph.D. degree with the Department of Electronic and Computer Engineering, The Hong Kong University of Science and Technology, Hong Kong.

His current research interests include high-level synthesis, placement, and computer architecture.

Jieru Zhao (Member, IEEE) received the Ph.D. degree in electronic and computer engineering from The Hong Kong University of Science and Technology, Hong Kong, in 2020.

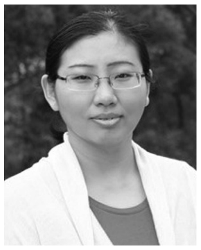
She is an Assistant Professor with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China.

Dr. Zhao is a recipient of the Best Paper Award at ICCAD 2017 and the Best Paper Nominations at DATE 2022, SC 2021, and CASES 2018.

Sharad Sinha (Senior Member, IEEE) received the Bachelor of Technology (B.Tech.) degree in electronics and communication engineering from Cochin University of Science and Technology, Kochi, India, in 2007.

He is an Associate Professor of Computer Science and Engineering with the Indian Institute of Technology Goa, Goa, India. Previously, he was a Research Scientist with Nanyang Technological University, Singapore. From 2007 to 2009, he was a Design Engineer with Processor Systems (India) Pvt. Ltd, Bengaluru, India.

Dr. Sinha has received the Best Paper Award at ICCAD 2017, ICCAD 2022, and IEEE INDICON 2022, the Best Paper Award Nomination at CASES 2018, and the Best Speaker Award from IEEE CASS Society, Singapore Chapter, in 2013 for his Ph.D. work on High-Level Synthesis. He serves as an Associate Editor for IEEE JOURNAL OF TRANSLATIONAL ENGINEERING IN HEALTH AND MEDICINE and as a Senior Editor for *ACM Ubiquity*.



Wei Zhang (Member, IEEE) received the Ph.D. degree from Princeton University, Princeton, NJ, USA, in 2009.

She is currently a Professor with the Department of Electronic and Computer Engineering, The Hong Kong University of Science and Technology, Hong Kong. She was an Assistant Professor with the School of Computer Engineering, Nanyang Technological University, Singapore, from 2010 to 2013. She has authored over 140 book chapters and papers in peer reviewed journals and international conferences. Her current research interests include reconfigurable system, computer architecture, EDA, and embedded system security.

Prof. Zhang's team has won the Best Paper Award in ISVLSI 2009, ICCAD 2017, and ICCAD 2022. She serves as an Associate Editor for IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, *ACM Transactions on Embedded Computing Systems*, IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, *ACM Transactions on Reconfigurable Technology and Systems*, and *ACM Journal on Emerging Technologies in Computing Systems*. She also serves on many organization committees and technical program committees.