Integrating Approximate Logic Synthesis into Approximate High-Level Synthesis

Jian Shi[1], Ruicheng Dai[1], Chang Meng[2], Yue Yang[1], and Weikang Qian[1]
[1]Global College, Shanghai Jiao Tong University, Shanghai, China
[2]Department of Mathematics and Computer Science, Eindhoven University of Technology, Eindhoven, the Netherlands
timeshi@sjtu.edu.cn,ruichengdai@sjtu.edu.cn,c.meng@tue.nl,yue.yang@sjtu.edu.cn,qianwk@sjtu.edu.cn

Abstract

Approximate high-level synthesis (HLS) and approximate logic synthesis (ALS) are two techniques for generating approximate circuits. They operate at different granularities. Approximate HLS typically modifies instructions in a control and data flow graph, whereas ALS modifies gates and interconnects in a gate-level netlist. The absence of a unified framework combining these techniques limits the potential for joint optimization. To bridge this gap, we propose to integrate ALS into the flow of approximate HLS. This integration expands the design space of approximate HLS by introducing fine-grained approximation induced by ALS, thereby generating approximate circuits with higher quality. Experimental results show that under the same error bound, our method reduces the hardware cost by 11% on average compared to the state-of-the-art methods.

1 Introduction

Many modern applications, such as image processing and machine learning, often require substantial computational resources [1]. Fortunately, many of these applications can tolerate certain levels of errors. *Approximate computing* takes advantage of this error tolerance to trade computational accuracy for improved performance or energy efficiency.

An important area in approximate computing is designing *approximate circuits*. One approach to this is *approximate logic synthesis* (*ALS*), which automatically generates gate-level approximate circuits with minimized circuit area or delay within a given error bound [2]. ALS focuses on optimizing gate-level netlists by modifying their internal signals [3–9]. It explores a large design space, enabling the generation of high-quality approximate circuits, but the vast design space also limits its application to large circuits [8].

To handle large-scale designs, *approximate high-level synthesis* (HLS) has been proposed [10]. It is built upon traditional HLS, a technique to automatically convert designs specified in high-level languages (*e.g.*, C/C++) into *register-transfer level* (*RTL*) implementations. A key step in an HLS flow is transforming high-level languages into a *control and data flow graph* (*CDFG*), which captures the control and data dependencies within the design [11]. Fig. 1(a) shows an example of a CDFG, where nodes represent different instructions or IO ports and edges represent their dependencies. Existing approximate HLS methods either apply coarse-grained approximation methods such as precision scaling and variable-to-variable substitution to the CDFG or rely on pre-designed approximate arithmetic units from a library [12–18]. This restricted set of approximation choices causes the generated approximate circuits frequently exhibiting suboptimal quality, characterized by limited area reduction and unacceptably high output errors.

To solve the above issues, in this work, we propose to integrate ALS into the flow of approximate HLS and develop a novel approximate HLS method, dubbed *HALS*. The rationale behind this is that ALS can introduce finer-grained approximation. Hence, this integration expands the design space of approximate HLS, thereby enabling the generation of approximate circuits with higher quality. Our main contributions are as follows:

- We propose a partitioning method tailored for ALS within HLS. This method partitions a given CDFG into ALS-manageable sub-graphs.
- We introduce a fast error estimation method to evaluate the error impact of ALS-generated circuits within the HLS flow.
- We develop an efficient *design space exploration* (*DSE*) method to select an optimized combination of ALS-generated candidates during the approximate HLS process.

The experimental results show that under the same error bound, our method reduces the hardware cost by 11% on average compared to the previous state-of-the-art works. The code of HALS is available at https://www.github.com/SJTU-ECTL/HALS.

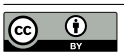
2 Background and Related Works

2.1 Approximate High-Level Synthesis

As introduced earlier, HLS transforms high-level specifications (*e.g.*, in C/C++) into RTL designs. In an HLS flow, it first compiles the source code into a compiler-level *intermediate representation* (*IR*). Then, it analyzes the control and data dependencies among the IR instructions to construct a CDFG [19]. As shown in Fig. 1(a), the IR instructions are represented as circles connected by directed edges, which denote their dependencies. The graph is bounded by input and output ports represented by green and purple squares, respectively. These ports are essential for modeling but do not consume actual hardware resources. Once optimized, the CDFG is scheduled and mapped to the final RTL design.

To further improve hardware performance, approximate HLS is proposed [10]. It directly generates approximate designs from the given high-level specifications. Existing approximate HLS methods primarily focus on coarse-grained approximations applied to individual nodes or edges within a CDFG. Major techniques include

This work is supported by the National Natural Science Foundation of China under Grant 62574132 and the Natural Science Foundation of Shanghai under Grant 25ZR1401189. Corresponding author: Weikang Qian.



ICCAD '26, San Jose, CA, USA

ACM ISBN 979-8-4007-2873-0/2026/11
https://doi.org/10.1145/3831252.3834065

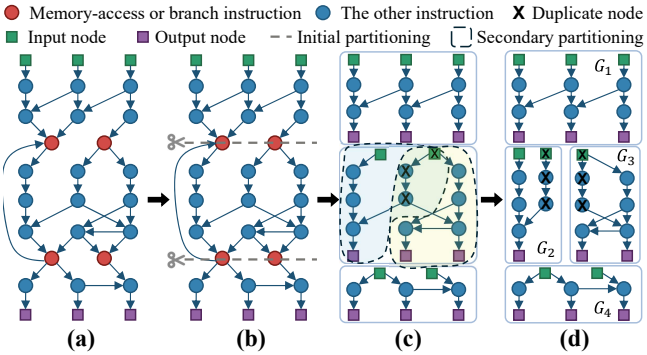


Figure 1: Overview of the graph partitioning in HALS. The process begins with (a) a CDFG containing nodes and edges representing IR instructions and dependencies, respectively. (b) The initial partitioning uses memory-access and branch instructions as boundaries. (c) The secondary partitioning applies the spectral clustering on outputs to further partition large sub-graphs. (d) The set of partitioned sub-graphs is prepared for the subsequent ALS flow.

precision scaling [12, 13], *mapping instructions to pre-designed approximate function units* [15, 17, 18], *variable-to-variable/constant substitution* [16, 18], and *loop perforation* [20–22]. However, these techniques are applied to nodes in the CDFG individually. They neglect the substantial design space of joint optimization across multiple consecutive nodes, resulting in suboptimal quality. To unlock joint optimization, fine-grained gate-level techniques must be exploited in the HLS flow. This necessitates the integration of ALS.

2.2 Approximate Logic Synthesis

ALS is an automated technique that optimizes gate-level netlists to minimize hardware costs (*i.e.*, area, delay, and power) while strictly adhering to a given error bound [2]. Unlike coarse-grained HLS approximations, ALS achieves this by exploring a massive design space through *local approximate changes* (*LACs*), such as replacing a signal with a constant value (0 or 1) or a similar existing signal [3–8]. These LACs modify local subcircuits, thereby enabling the fine-grained, joint optimization across multiple IR nodes that cannot be achieved by conventional approximate HLS.

To navigate the vast design space, most ALS methods operate in an iterative manner [3–8]. In each iteration, the ALS methods identify and evaluate a massive number of candidate LACs before applying the modification. Because this evaluation process is invoked frequently and must capture the complex impact of fine-grained structural changes, efficient error estimation is needed to accelerate the ALS flow [9, 23, 24]. However, existing estimation methods evaluate LACs strictly at the *primary outputs* (*POs*) of the given netlist. They do not consider the impact when this approximated netlist is integrated into a larger design generated by HLS. To unlock the full potential of ALS within approximate HLS, an error estimation method is required to rapidly evaluate the impact of local LACs on the entire HLS-generated design.

2.3 Error Metrics and Error Estimation Methods

Evaluating the accuracy of approximate circuits requires well-defined error metrics. In this work, we focus on average error metrics, including *mean absolute error* (*MAE*), *mean square error* (*MSE*), *mean absolute percentage error* (*MAPE*), *signal-to-noise ratio* (*SNR*), and *peak signal-to-noise ratio* (*PSNR*). Suppose a circuit has $\mathcal{T}$ input combinations and the probability of input combination t is $P(t)$. Let $F_{exact}(t)$ and $F_{approx}(t)$ denote the exact and approximate outputs, respectively, for input combination t and F_{MAX} be the maximum possible output value. These error metrics are defined as follows:

$$MAE = \sum_{t=1}^{\mathcal{T}} \left(P(t) \cdot \left| F_{exact}(t) - F_{approx}(t) \right| \right), \tag{1}$$

$$MSE = \sum_{t=1}^{\mathcal{T}} \left(P(t) \cdot \left(F_{exact}(t) - F_{approx}(t) \right)^2 \right), \tag{2}$$

$$MAPE = \sum_{t=1}^{\mathcal{T}} \left(P(t) \cdot \left| \frac{F_{exact}(t) - F_{approx}(t)}{F_{exact}(t)} \right| \right), \tag{3}$$

$$SNR = 10 \cdot \log_{10} \left(\frac{\sum_{t=1}^{\mathcal{T}} P(t) \cdot F_{exact}(t)^2}{MSE} \right), \tag{4}$$

$$PSNR = 10 \cdot \log_{10} \left(\frac{F_{MAX}^2}{MSE} \right). \tag{5}$$

In practice, the average error of an approximate circuit is typically evaluated using *Monte Carlo* (*MC*) *simulation* [25]. This method samples T input combinations and calculates the average error across these samples as the estimated average error for the circuit. However, MC simulation is slow for large designs. To accelerate the evaluation process, researchers have proposed various error models to predict errors during approximate HLS. They can be broadly categorized into two types: analytical models and *machine learning* (*ML*)-based models [26].

Analytical models predict errors by propagating them through consecutive arithmetic instructions, achieving high accuracy in arithmetic-intensive datapaths [13–15, 17, 27, 28]. However, practical HLS applications contain instructions that disrupt straightforward error propagation, such as memory accesses and branches. The presence of these instructions leads to extensive case-by-case analysis in analytical models, limiting their generalizability [14].

ML-based models, such as *multi-layer perceptrons* (*MLPs*) and *graph neural networks* (*GNNs*), have also been used to predict errors in approximate HLS [29, 30]. These models can effectively capture complex error relationships even when memory-access and branch instructions are present. However, ML-based models incur substantial inference latency per invocation, which limits their applicability in the iterative ALS step within the flow of approximate HLS.

Therefore, unlocking the power of ALS within HLS requires a fast error model that can handle complex situations when memory-access and branch instructions are present.

3 Methodology

We first formally define the optimization problem.

Problem Formulation: Given a high-level design description C_{HLS} representing the exact baseline hardware design $\mathcal{D}$, an input data distribution $\mathcal{P}_{\mathcal{T}}$, a user-specified error metric $\mathcal{E}$, and a corresponding error bound e_b, the objective of HALS is to generate an approximate hardware design $\mathcal{D}'$ that minimizes the total hardware cost while strictly satisfying $e_{\mathcal{D}'} \leq e_b$, where $e_{\mathcal{D}'}$ is the error of $\mathcal{D}'$ evaluated under the error metric $\mathcal{E}$ and the input distribution $\mathcal{P}_{\mathcal{T}}$.

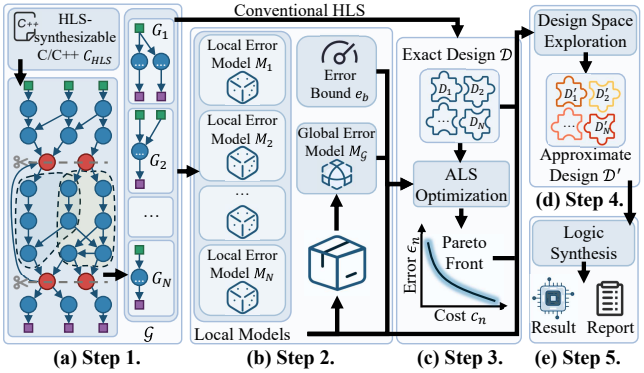


Figure 2: Overview of HALS: (a) Step 1: Convert the given C/C++ into a CDFG and partition it into ALS-manageable sub-graphs $\mathcal{G} = \{G_1, \ldots, G_N\}$; (b) Step 2: Construct a local error model M_n for each sub-graph G_n and a global error model $M_{\mathcal{G}}$ evaluating the combined impact of all sub-graphs; (c) Step 3: Apply ALS to the exact design of each sub-graph; (d) Step 4: Selectively replace sub-graphs with their approximate candidates through DSE; (e) Step 5:Optimize the final design by logic synthesis and generate the corresponding report.

To solve this problem, HALS employs a five-step flow, as shown in Fig. 2. It takes C_{HLS}, $\mathcal{P}_{\mathcal{T}}$, $\mathcal{E}$, and e_b as inputs and proceeds as follows. In Step 1 (Section 3.1), C_{HLS} is compiled into a CDFG, which is then partitioned into a set of ALS-manageable sub-graphs, $\mathcal{G} = \{G_1, \ldots, G_N\}$. Each $G_n \in \mathcal{G}$ corresponds to an exact hardware module D_n generated by a conventional HLS flow, which collectively form the exact input design $\mathcal{D}$. In Step 2 (Section 3.2), a local error model M_n is built for each sub-graph G_n, and a global error model $M_{\mathcal{G}}$ is constructed to characterize the combined impact of all sub-graphs on the final design error. In Step 3 (Section 3.3), guided by the local error model M_n, ALS generates a set of approximate candidates for each sub-graph G_n based on D_n. In Step 4 (Section 3.4), guided by the global error model $M_{\mathcal{G}}$, HALS selectively replaces sub-graphs in $\mathcal{G}$ with their approximate candidates. This process assembles the approximate design $\mathcal{D}'$ that minimizes hardware cost while strictly ensuring its overall error $e_{\mathcal{D}'}$ is below e_b. In Step 5 (Section 3.5), the assembled approximate design $\mathcal{D}'$ undergoes conventional logic synthesis and optimization to produce the final design. The following subsections detail these steps.

3.1 Step 1: Graph Partitioning

In this step, the input high-level design description C_{HLS} is compiled and partitioned into a set of N ALS-manageable sub-graphs $\mathcal{G} = \{G_1, G_2, \ldots, G_N\}$, as shown in Fig. 2(a). This step begins by compiling C_{HLS} into an *LLVM intermediate representation* (*LLVM IR*) using *Clang*, a high-performance and open-source compiler frontend [31]. During compilation, standard optimization techniques such as loop unrolling, function inlining, and abstract syntax tree flattening are applied to maximize the throughput of the design [32–34]. By analyzing the control and data dependencies among the IR instructions, a CDFG is constructed. HALS then applies a two-stage partitioning method to this CDFG to divide it into ALS-manageable sub-graphs, as detailed below.

3.1.1 Initial Partitioning. Most ALS methods are only applicable to circuits represented as *directed acyclic graphs* (*DAGs*) and lack support for memory operations. However, a CDFG generated by HLS may contain cycles and memory-access instructions. Thus, ALS cannot be directly applied to HLS-generated designs.

In a CDFG, cyclic dependencies are induced by the back edges of branch instructions. To guarantee that every resulting sub-graph is a DAG, HALS treats branch instructions as partition boundaries. Furthermore, to ensure the partitioned sub-graphs are fully compatible with ALS, memory-access instructions are also used as partition boundaries. As shown in Fig. 1(b), gray dashed lines denote the cuts made at these boundaries, separating the memory-access and branch instructions from the other instructions.

To ensure the structural completeness of each obtained sub-graph, input and output ports are inserted at the boundaries. Under this partitioning strategy, a single sub-graph may contain multiple input and output ports, as shown in Fig. 1(c).

3.1.2 Secondary Partitioning. Conventional ALS is applied to the gate-level netlist, where the computational complexity of some steps grows exponentially with the gate count [35]. For a large sub-graph from the initial partitioning, the gate count of the synthesized netlist can be excessive, leading to long ALS runtime. Thus, it is necessary to further partition a large sub-graph into smaller ones while minimizing logic duplication across these smaller sub-graphs. Although conventional partitioning methods (such as ratio cut) can reduce sub-graph size, they often introduce complex inter-partition dependencies that would complicate the subsequent error model [36]. To avoid the complex dependencies while minimizing logic duplication, HALS adopts an *output similarity-based clustering* method to perform the secondary partitioning.

Specifically, given a large sub-graph from the initial partitioning, HALS first groups its outputs into several clusters based on their logical similarities. For each cluster, a new sub-graph is generated by retaining its outputs and their corresponding ancestor instructions. This method avoids the complex inter-partition dependencies introduced by arbitrary graph cuts. Meanwhile, clustering outputs with high similarity maximizes logic reuse within each sub-graph and minimizes logic duplication across sub-graphs. For each obtained sub-graph, it is then simplified using standard LLVM passes. Figs. 1(c) and (d) show an example of this secondary partitioning process and the corresponding result, respectively.

Let G_p be a sub-graph from the initial partitioning. Assume it contains k_p outputs $\{o_{p,1}, \ldots, o_{p,k_p}\}$ and θ_p IR instructions. The middle sub-graph in Fig. 1(c) shows an example with $k_p = 3$ and $\theta_p = 9$. For each output $o_{p,i}$, a backward traversal collects all its ancestor instructions within G_p, forming an ancestor set $S_{p,i}$.

HALS uses the *spectral clustering* method [37] to realize the output similarity-based clustering. Specifically, it constructs a similarity matrix $\mathbf{A}_p \in [0,1]^{k_p \times k_p}$, where $\mathbf{A}_p[i,j]$ quantifies the similarity between outputs $o_{p,i}$ and $o_{p,j}$ using the Jaccard index of their ancestor sets [37]:

$$\mathbf{A}_p[i,j] = \text{Jaccard}\left(S_{p,i}, S_{p,j}\right) = \frac{\left|S_{p,i} \cap S_{p,j}\right|}{\left|S_{p,i} \cup S_{p,j}\right|}. \tag{6}$$

A higher $\mathbf{A}_p[i, j]$ value indicates a larger overlap in the ancestor instructions of $o_{p,i}$ and $o_{p,j}$. The i-th row of $\mathbf{A}_p$ serves as a k_p-dimensional feature vector for $o_{p,i}$, projecting $o_{p,i}$ to a space based on its similarity to all other outputs. HALS then applies k-means clustering to these row vectors to cluster the sub-graph outputs.

Although the similarity-based clustering method minimizes logic duplication across partitions, some logic duplication across different sub-graphs is inevitable because outputs in separate clusters may still share common ancestors. For example, the dashed regions in Fig. 1(c) outline two sub-graphs, G_2 and G_3, corresponding to two clusters. Between G_2 and G_3, two overlapping ancestor instructions and one input are duplicated, as marked by an "X" in Fig. 1(d). While this redundancy might temporarily counteract hardware cost reduction, much of the duplicated logic is eliminated during the circuit optimization phase (detailed in Section 3.5).

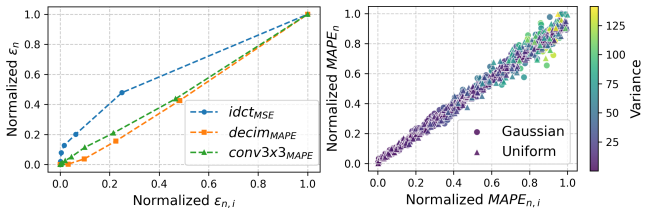
3.2 Step 2: Error Model Construction

In this step, we construct error models tailored for HALS, as shown in Fig. 2(b). As discussed in Sections 2.2 and 2.3, HALS requires a fast error estimation method that maps the error impact of sub-graph LACs to the entire HLS-generated design $\mathcal{D}$, while capable of handling memory-access and branch instructions. To satisfy these requirements, we propose a scalable *two-level regression model*, consisting of local error models and a global error model.

3.2.1 Local Error Model. For each sub-graph G_n, a local error model M_n is established to guide the ALS. Conventional ALS methods select candidate LACs based on their errors at the POs of the given netlist. In the context of HALS, this implies that the ALS methods can only evaluate the local errors at the outputs $\{o_{n,1}, o_{n,2}, \ldots, o_{n,k_n}\}$ of the sub-graph G_n. We denote the error of $o_{n,i}$ as $\epsilon_{n,i}$. When G_n is approximated while the rest of the design remains exact, these local errors propagate, causing the entire design $\mathcal{D}$ to exhibit a system-level error ϵ_n. To capture this relationship, M_n is designed to map the local errors $\{\epsilon_{n,1}, \epsilon_{n,2}, \ldots, \epsilon_{n,k_n}\}$ of G_n to its induced system-level error ϵ_n.

We first study the mapping properties by conducting experiments on some designs and error metrics. Specifically, we conducted MC simulations by injecting errors into one output $o_{n,i}$ of a sub-graph, while maintaining the other outputs exact, and observed the corresponding system-level error ϵ_n. When the error magnitude is not excessive (quantified by MAPE $\leq 30\%$ for each $o_{n,i}$), the average errors exhibit the following properties:

- *Monotonicity*: For a given sub-graph output $o_{n,i}$, there exists a monotonic relationship between $\epsilon_{n,i}$ and ϵ_n. As shown in Fig. 3(a), this monotonic behavior is consistently observed across different designs and error metrics.
- *Distribution invariance*: If two different approximation methods with different error distributions produce a similar average error $\epsilon_{n,i}$, they result in a comparable ϵ_n. Fig. 3(b) shows this using a 5-stage decimation filter. Despite differences in the injected error variance (represented by color gradients), the relationship between the normalized $\text{MAPE}_{n,i}$ and MAPE_n remains highly consistent.



(a) (b)

Figure 3: Experimental validation of error propagation properties. (a) Monotonic relationship between the normalized local error $\epsilon_{n,i}$ and the sub-graph-induced system-level error ϵ_n across different designs and error metrics. (b) Distribution invariance observed in a 5-stage decimation filter, where Gaussian and uniform error distributions with different variances are injected into the third stage. Despite different variances (indicated by color gradients), similar normalized $\text{MAPE}_{n,i}$ values result in comparable normalized MAPE_n.

Based on these observations, M_n is formulated as follows:

$$M_n := \epsilon_n = \sum_{i=1}^{k_n} \alpha_{n,i} \cdot (\epsilon_{n,i})^{\beta_{n,i}} + \gamma_n, \tag{7}$$

where k_n is the number of outputs for G_n, and $\alpha_{n,i}$, $\beta_{n,i}$, and γ_n are parameters to be fitted. The model is considered reliable if γ_n is relatively small and the coefficient of determination R_n^2 exceeds 0.85; otherwise, a neural network is recommended as a fallback.

3.2.2 Global Error Model. Because each local error model M_n evaluates an individual sub-graph, it ignores the error interactions with other sub-graphs. To capture these interactions for the subsequent DSE step (detailed in Section 3.4), a global model $M_{\mathcal{G}}$ is introduced. Let $\epsilon_{\mathcal{G}}$ denote the *concurrent global error* of design $\mathcal{D}$ when all sub-graphs are approximated simultaneously. $M_{\mathcal{G}}$ is designed to map the individual sub-graph-induced system-level errors ϵ_n to this concurrent global error $\epsilon_{\mathcal{G}}$. Similar to the local model, $M_{\mathcal{G}}$ is formulated as follows:

$$M_{\mathcal{G}} := \epsilon_{\mathcal{G}} = \sum_{n=1}^{N} \alpha_n \cdot (\epsilon_n)^{\beta_n} + \gamma, \tag{8}$$

where α_n, β_n, and γ are parameters to be fitted. The model is considered reliable if γ is relatively small and $R_{\mathcal{G}}^2$ exceeds 0.85.

3.2.3 Model Fitting via IR-Level Simulation. To fit M_n and $M_{\mathcal{G}}$, HALS collects paired observations of $(\{\epsilon_{n,1}, \epsilon_{n,2}, \ldots, \epsilon_{n,k_n}\}, \epsilon_n)$ and $(\{\epsilon_1, \epsilon_2, \ldots, \epsilon_N\}, \epsilon_{\mathcal{G}})$ using LLVM IR-level MC simulations, which execute much faster than RTL or gate-level simulations. By executing these simulations across the entire design, HALS takes all memory-access and branch instructions into account and captures how local approximations aggregate and propagate through these instructions to the design outputs.

In each simulation round, input stimuli are sampled from the user-provided distribution $\mathcal{P}_{\mathcal{T}}$. To simulate the effect of ALS optimizations, HALS injects an unbiased, uniformly distributed error into each output $o_{n,i}$ of G_n by rounding its least significant $\tau_{n,i}$ bits. The rounding bitwidth $\tau_{n,i}$ ranges from 1 to $(\psi_{n,i} - \Delta_\psi)$, where $\psi_{n,i}$ is the original bitwidth of $o_{n,i}$, and Δ_ψ is a user-specified parameter. Using these randomized rounding configurations, HALS executes two distinct simulation phases to collect the fitting data:

- *Local model fitting*: HALS approximates the outputs of one sub-graph G_n at a time, leaving the rest of the design exact. This step collects the local errors $\{\epsilon_{n,1}, \epsilon_{n,2}, \ldots, \epsilon_{n,k_n}\}$ (measured at the outputs of G_n) and ϵ_n (measured at the outputs of $\mathcal{D}$) to fit the $(2k_n + 1)$ parameters of M_n.
- *Global model fitting*: HALS samples several approximate configurations per sub-graph from the local fitting phase and combines them into a number of global configurations. Each combination corresponds to a vector of sub-graph-induced system-level errors $\{\epsilon_1, \epsilon_2, \ldots, \epsilon_N\}$ measured in the local fitting phase. Furthermore, the combination is applied to all sub-graphs simultaneously to yield the concurrent global error $\epsilon_{\mathcal{G}}$. The data pairs $(\{\epsilon_1, \ldots, \epsilon_N\}, \epsilon_{\mathcal{G}})$ over all combinations are used to fit the $(N + 1)$ parameters of $M_{\mathcal{G}}$.

Since M_n and $M_{\mathcal{G}}$ contain only $(2k_n+1)$ and $(2N+1)$ parameters, respectively, the required numbers of simulation rounds to fit M_n and $M_{\mathcal{G}}$ scale linearly with the number of sub-graph outputs k_n and the total number of sub-graphs N, respectively.

3.3 Step 3: ALS for Sub-Graph

As shown in Fig. 2(c), this step performs ALS on each sub-graph G_n identified in Step 1. ALS takes three inputs: an exact circuit, an error metric, and a corresponding error bound. The exact circuit is set as the exact hardware module D_n generated for G_n in Step 1. Instead of using a conventional error metric, the ALS step of HALS is guided by the sub-graph-induced system-level error ϵ_n, which is evaluated by applying the local error model M_n constructed in Step 2 on the local output errors $\{\epsilon_{n,1}, \ldots, \epsilon_{n,k_n}\}$ of the ALS-generated design. The error bound is set as the given bound e_b.

Most ALS methods operate as iterative and incremental optimization processes. They start from the exact circuit and progressively minimize the hardware cost while satisfying the given error bound. During this progress, many intermediate approximate circuits are generated. Although these intermediate circuits consume more hardware resources than the final optimized circuit, they exhibit lower errors. Prior work has proposed to use these intermediate approximate circuits as candidates for subsequent DSE [38]. Inspired by this, for each sub-graph G_n, HALS collects these intermediate circuits during the ALS process and extracts a Pareto front representing the optimized trade-offs between its induced system-level error ϵ_n and the corresponding hardware cost c_n. These circuits then serve as the candidates for the subsequent DSE in Step 4.

3.4 Step 4: Optimized ALS Design Selection

In this step, as shown in Fig. 2(d), HALS selectively replaces the exact sub-graphs with their ALS-generated approximate candidates. The objective is to minimize the total hardware cost (*e.g.*, area, delay, or power) without violating the global error bound e_b.

For each sub-graph G_n, Step 3 has already generated a Pareto front representing the optimized trade-offs between its induced system-level error ϵ_n and the corresponding hardware cost c_n. To determine an optimized combination of these candidates across the entire design, HALS employs a greedy iterative DSE algorithm inspired by the search method in PreDAC [17]. The search is guided by a *cost-error ratio*. Specifically, assume that the current candidate for G_n has a cost c_n and an induced system-level error ϵ_n. If a new candidate yields a lower cost $c_{n'}$ but a higher induced error $\epsilon_{n'}$, the cost-error ratio is calculated as:

$$\text{cost-error ratio} = \frac{\Delta\text{cost}}{\Delta\epsilon_{\mathcal{G}}} = \frac{c_n - c_{n'}}{\alpha_n \cdot \left((\epsilon_{n'})^{\beta_n} - (\epsilon_n)^{\beta_n}\right)}, \tag{9}$$

where the denominator is derived from the global error model $M_{\mathcal{G}}$. A higher cost-error ratio indicates a more favorable trade-off between the error and the hardware cost.

At each iteration, HALS greedily selects the candidate with the maximum cost-error ratio over all sub-graphs to replace its corresponding current candidate. The iterative replacement process continues until the global error predicted by $M_{\mathcal{G}}$ reaches the bound e_b, or all approximate candidates have been considered. The obtained approximate design is then validated by a gate-level MC simulation. If the actual simulated error exceeds e_b, the DSE process triggers a rollback mechanism: it undoes the most recent replacement, reverting to the preceding design, and re-validates it. This rollback continues until an approximate design that strictly satisfies e_b is found. Then, the obtained approximate design $\mathcal{D}'$ is passed to Step 5 for final logic synthesis and optimization.

3.5 Step 5: Circuit Optimization

In this step, as shown in Fig. 2(e), the assembled approximate design $\mathcal{D}'$ from Step 4 undergoes full-design logic synthesis and optimization to yield the final hardware implementation. Synthesizing $\mathcal{D}'$ as a whole is essential for two reasons. First, it allows conventional synthesis tools to globally identify and eliminate the temporary logic duplication introduced by the secondary partitioning in Step 1. Second, it unlocks more optimization opportunities. Techniques such as retiming [39] and register packing [40] can exploit these opportunities to further improve area, delay, and power. Finally, this step outputs the optimized approximate design alongside a report on circuit area, delay, power, and error.

4 Experimental Results

4.1 Experimental Setup

The HLS was implemented using an LLVM-based front-end [31] and an in-house developed backend. For ALS, we utilized a modified version of ResubALS [7] that integrates the local error models for the partitioned sub-graphs. Logic synthesis and power analysis were performed using Yosys [41] and Synopsys Design Compiler [42], respectively, while Verilator [43] was used for RTL simulation. Consistent with prior works [17, 18], the synthesis flow utilized the Nangate 45nm library [44]. All experiments were conducted on a server with an AMD Ryzen 9 5950X CPU at 3.4GHz using 16 threads and 32GB of DDR4 memory.

We evaluated HALS on 14 benchmarks. Specifically, 5 applications were sourced from S2CBench [45], 4 from PaderBench [46], and 5 from PolyBench [47]. Table 1 details the characteristics of these benchmarks, including the number of simulation patterns used for error modeling (denoted as *#Sim. Patterns*), the target hardware optimization metric (*Metric*), the baseline cost of the original exact circuit evaluated under the corresponding metric (*Orig. Cost*), and the final number of partitioned sub-graphs (*#Sub-graphs*).

To ensure realistic evaluation, the input distributions were tailored for each benchmark. For *fir9*, *fir13*, *interp*, and *decimation*, the

Table 1: Characteristics of the evaluated benchmarks. The baseline cost is measured in μm^2 for area and μW for power.

Benchmark	#Sim. Patterns	Metric	Orig. Cost	#Sub-graphs
fir9 [45]	10,000	Area	465.13	1
fft [45]	10,000	Area	76281	16
interp [45]	10,000	Area	49932	4
decimation [45]	10,000	Area	12489	5
Sobel [45]	512 × 512	Area	556.56	1
RGB2YCbCr [46]	10,000	Area	1673.7	1
conv3x3 [46]	10,000	Power	4950.0	1
fir13 [46]	10,000	Area	1176.9	1
Gauss.Blur [46]	512 × 512	Power	428.86	1
cholesky [47]	65,536	Area	37389	3
gesummv [47]	65,536	Area	9370.1	2
jacobi-1d [47]	65,536	Area	12669	2
jacobi-2d [47]	65,536	Area	14702	2
syr2k [47]	65,536	Area	7479.9	2

inputs comprise data points that are uniformly distributed and coefficients following specific mathematical distributions. For *Sobel* and *Gauss.Blur*, inputs are derived from real-world image pixel distributions. For all other benchmarks, including *conv3x3*, *fft*, *RGB2YCbCr*, and the PolyBench benchmarks, all inputs are uniformly distributed.

To ensure fair comparisons and demonstrate the generality of HALS, the target hardware cost metrics were selected to match those of prior works [17, 18]. As shown in Table 1, circuit area is evaluated for most benchmarks, whereas power consumption is only used for *conv3x3* and *Gauss.Blur*. The design quality is reported using the *hardware cost ratio*, defined as the ratio of the hardware cost of the approximate design over that of the original exact design. A lower ratio indicates a better result.

In some small benchmarks, boundaries for the initial partitioning (*e.g.*, branch and memory-access instructions) are either absent or localized near the inputs, resulting in a single sub-graph after the initial partitioning. Furthermore, this sub-graph contains only one output. Therefore, the secondary partitioning is bypassed, yielding a single indivisible sub-graph for these benchmarks, as reflected in Table 1. Consequently, no error models were constructed for them. Instead, the errors during the ALS and DSE steps were evaluated through direct MC simulations. For the other benchmarks, the number of sub-graphs is determined by the partitioning step, where the number of centroids for the k-means clustering process in the secondary partitioning is adjustable. More centroids yield more, but smaller, sub-graphs. Since the runtime of the ALS step grows exponentially with the sub-graph gate count, increasing the centroids reduces the overall HALS runtime. However, more sub-graphs also increase logic duplication, thereby degrading the design quality. On the other hand, fewer centroids produce larger sub-graphs that can exhaust server memory during the ALS step. Therefore, the number of centroids was empirically minimized to maximize design quality while strictly adhering to the server memory constraint.

4.2 Evaluation of the Error Model

To evaluate the proposed error model, we compare its performance against a neural network baseline. Specifically, we trained a unified MLP to replace both the local models (M_n) and the global model ($M_{\mathcal{G}}$). The MLP consists of two hidden layers with 64 and 32 neurons, respectively, and ReLU activations. It is trained on the same MC simulation datasets with the parameter Δ_ψ fixed at 3. Because the MLP serves as a unified predictor, its input layer is designed to accept the errors from all sub-graph outputs across the entire design. In the ALS step for a specific sub-graph G_n, we activate the input neurons corresponding to the errors of G_n's outputs ($\{\epsilon_{n,1}, \ldots, \epsilon_{n,k_n}\}$), and set all other inputs to zero. This allows the MLP to mimic M_n and predict the sub-graph-induced system-level error ϵ_n. In the DSE step, the MLP evaluates the combined errors from all candidate sub-graphs concurrently, substituting $M_{\mathcal{G}}$ to predict the global error $\epsilon_{\mathcal{G}}$.

Table 2: Performance comparison between the proposed model and the MLP baseline. For our method, the construction time is the total time it takes to construct all local models and the global model, while the inference time is reported separately for the local and global models.

Benchmark	Construction Time (s)		Inference Time (μs)			Accuracy (R^2)	
	Ours (Total)	MLP	Ours M_n	Ours $M_{\mathcal{G}}$	MLP	Ours ($M_{\mathcal{G}}$)	MLP
fft	18.2	23.0	0.44	0.36	1.25	**1.00**	0.98
interp	0.82	4.35	0.04	0.04	1.21	**1.00**	**1.00**
decimation	3.50	1.54	0.11	0.09	1.17	0.91	**0.94**
cholesky	1.89	4.68	0.10	0.06	1.20	**0.96**	**0.96**
gesummv	15.5	3.50	0.05	0.03	1.20	**1.00**	**1.00**
jacobi-1d	96.1	37.4	0.04	0.04	1.11	0.98	**0.99**
jacobi-2d	153	58.7	0.04	0.03	1.11	**1.00**	**1.00**
syr2k	0.42	1.11	0.03	0.03	1.34	**1.00**	**1.00**
Geomean	7.44	7.06	0.07	0.06	1.20	**0.98**	**0.98**

Table 2 details the construction time, inference time, and final accuracy for the MLP-based error model and our proposed model on benchmarks with more than one sub-graph in Table 1. The single-sub-graph benchmarks are not considered as their errors are evaluated via direct MC simulations without using the error models. The accuracy is quantified by R^2, with the highest value for each benchmark highlighted in bold. For our model, the construction time includes dataset generation and fitting for all local models and the global model. For the baseline MLP model, it includes global dataset generation and MLP training time. The results indicate that both methods demand comparable construction time and achieve nearly identical prediction accuracy with average $R^2 \approx 0.98$. However, our method shows an advantage in inference efficiency. Because our models avoid the matrix multiplications in MLPs, our local models (M_n) evaluate 17.1× faster on average than the MLP in the ALS step. Similarly, our global model ($M_{\mathcal{G}}$) achieves a 20× speedup over the MLP in the DSE step.

4.3 Comparison with State-of-the-Art Methods

To evaluate HALS against state-of-the-art approximate HLS methods, we compare it with ADVISOR [18] and PreDAC [17]. Table 3 shows the hardware cost ratios and runtimes (denoted as *RT*) across various error bounds e_b. The results for the compared methods are taken from their papers. For a fair comparison, our evaluation focuses on the 9 benchmarks from S2CBench and PaderBench used in their respective studies. Specifically, the S2CBench benchmarks (the first five in Table 3) are compared against ADVISOR, whereas the PaderBench benchmarks (the subsequent four) are compared

Table 3: Comparison of hardware cost ratio and runtime (*RT*) for ADVISOR, PreDAC, and HALS under various error bounds e_b.

Benchmark	Error Metric $\mathcal{E}$	Tight Error Bound e_b	Prior Works		HALS (Ours)		Loose Error Bound e_b	Prior Works		HALS (Ours)	
			Cost Ratio	RT (s)	Cost Ratio	RT (s)		Cost Ratio	RT (s)	Cost Ratio	RT (s)
fir9	MAPE	10%	57.59%	840	**50.03%**	**18.50**	20%	39.27%	840	**32.65%**	**19.81**
fft	MAPE	10%	41.71%	1980	93.06%	4944.3	20%	28.71%	1980	89.81%	4945.4
interp	MAPE	10%	54.25%	3540	**38.86%**	6439.8	20%	46.67%	3540	**29.83%**	6440.1
decimation	MAPE	10%	46.59%	5580	56.24%	**3206.1**	20%	34.26%	5580	49.11%	**3208.0**
Sobel	PSNR	20dB	39.35%	2580	**34.80%**	**77.97**	10dB	18.92%	2580	22.74%	**86.91**
RGB2YCbCr	MAE	1000	55.32%	1731	**29.28%**	**1455.6**	1400	54.18%	1992	**28.20%**	**1460.1**
conv3x3	MAE	1000	38.40%	97.8	**34.55%**	20112	1400	36.80%	147	**29.29%**	23185
fir13	SNR	25dB	77.00%	14	**17.44%**	193.96	19dB	56.00%	17	**25.93%**	170.47
Gauss.Blur	SNR	25dB	51.90%	218	**32.58%**	**32.28**	19dB	41.60%	274	**14.52%**	**36.59**
Geomean			50.21%	706.39	**38.89%**	**669.42**		37.82%	786.85	**31.48%**	**693.58**

against PreDAC. Both baselines are collectively denoted as "Prior Works" in Table 3. Note that due to differences in computational platforms, the reported runtimes are provided for reference only.

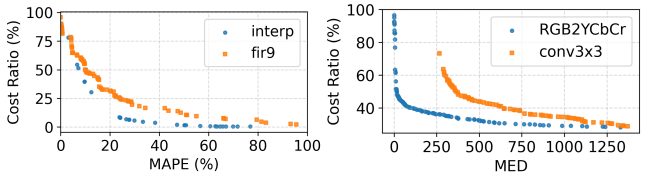
As shown in Table 3, HALS achieves lower hardware costs under the same error constraints for most benchmarks. Overall, HALS reduces the average hardware cost ratio by 11.32% compared to the baselines under tight error constraints, and by 6.34% under loose constraints. Two exceptions are *decimation* and *fft*, which we analyze in detail below.

Decimation is a pipelined design dominated by multiply-accumulate instructions across all stages. This architecture causes rapid error amplification as approximations in early stages propagate downstream. Therefore, HALS must enforce strict local error bounds on these early stages, meaning that most hardware cost reduction is derived from the final stages of the pipeline.

The highly connected butterfly structure of *fft* causes outputs to share a large number of ancestor instructions. Our secondary partitioning duplicates many shared ancestors across sub-graphs, which resulted in a 3.19× intermediate area increase before the ALS step. This massive duplication cannot be fully eliminated during the circuit optimization in Step 5. To address highly connected CDFGs like *fft*, we plan to introduce a duplication factor in the future to characterize sub-graph topology. A predefined threshold for this factor will allow HALS to dynamically bypass secondary partitioning, preventing excessive logic duplication.

Regarding the runtime, the ALS step dominates the overall runtime of HALS. Because the runtime complexity of certain steps in ALS grows exponentially with the gate count, the runtime is highly sensitive to the size of individual sub-graphs rather than the total circuit area. For example, although *conv3x3* does not possess the largest circuit area among the benchmarks, it cannot be partitioned, resulting in the longest runtime. In contrast, while *fft* exhibits the largest circuit area, it is partitioned into 16 independent sub-graphs. Because the ALS for these sub-graphs can be executed in parallel, its overall runtime is shorter than *conv3x3*.

To further evaluate HALS, we plot the Pareto fronts for 4 benchmarks across a broader range of error bounds in Fig. 4. The results show that HALS achieves substantial hardware savings under any given error constraint.



(a) (b)

Figure 4: Pareto fronts of hardware cost ratio vs. error for 4 benchmarks optimized by HALS.

4.4 Scalability on Complex Designs

While HALS shows superior performance on the benchmarks from S2CBench and PaderBench, most of them are relatively small and lack complex memory interactions. To evaluate the generality and scalability of HALS on large-scale and memory-intensive benchmarks, we extended our evaluation to five benchmarks from PolyBench [47]. Table 4 shows the results, including the runtime and area ratio, under a 10% MAPE constraint. Across these benchmarks, HALS achieved an average area ratio of 54.22%. The results show the robustness and scalability of the proposed HALS method.

Table 4: Optimization results of HALS for PolyBench benchmarks under a 10% MAPE error bound.

Benchmark	Orig. Area (μm^2)	Runtime (s)	Area Ratio
cholesky	37389	553.04	67.46%
gesummv	9370.1	250.21	72.34%
jacobi-1d	12669	157.48	44.52%
jacobi-2d	14702	147.68	33.77%
syr2k	7479.9	393.93	63.87%
Geomean	13731	263.39	54.22%

5 Conclusion

This paper introduces HALS, a framework that integrates ALS into the flow of approximate HLS. By incorporating fine-grained approximation induced by ALS, HALS expands the design space of approximate HLS and enables the generation of higher-quality approximate circuits. Experimental results show that under the same error bound, HALS reduces the average hardware cost by 11% compared to existing state-of-the-art methods. In the future, we plan to optimize the graph partitioning strategy to further reduce the hardware overhead introduced by partitioning.

References

[1] M. Ahmadinejad and M. H. Moaiyeri. Energy- and quality-efficient approximate multipliers for neural network and image processing applications. *TETC*, 10(2):1105–1116, 2022.

[2] I. Scarabottolo et al. Approximate logic synthesis: A survey. *Proc. IEEE*, 108(12):2195–2213, 2020.

[3] S. Venkataramani et al. Substitute-and-simplify: A unified design paradigm for approximate and quality configurable circuits. In *DATE*, pages 1367–1372, 2013.

[4] A. Chandrasekharan et al. Approximation-aware rewriting of AIGs for error tolerant applications. In *ICCAD*, pages 1–8, 2016.

[5] J. Schlachter et al. Design and applications of approximate circuits by gate-level pruning. *TVLSI*, 25(5):1694–1702, 2017.

[6] C. Meng et al. ALSRAC: Approximate logic synthesis by resubstitution with approximate care set. In *DAC*, pages 1–6, 2020.

[7] C. Meng et al. Efficient resubstitution-based approximate logic synthesis. *TCAD*, 44(6):2040–2053, 2025.

[8] X. Wang et al. AccALS 2.0: Accelerating approximate logic synthesis by simultaneous selection of multiple local approximate changes. *TCAD*, 44(5):1620–1633, 2025.

[9] R. Dai et al. Efficient approximate logic synthesis with dual-phase iterative framework. In *DATE*, pages 1–7, 2025.

[10] B. C. Schafer and B. Parchamdar. Approximate computing in high-level synthesis: From survey to practical implementation. *ACM Comput. Surv.*, 58(8), February 2026.

[11] R. Namballa et al. Control and data flow graph extraction for high-level synthesis. In *ISVLSI*, pages 187–192, 2004.

[12] D.-U. Lee et al. Accuracy-guaranteed bit-width optimization. *TCAD*, 25(10):1990–2000, 2006.

[13] C. Li et al. Joint precision optimization and high level synthesis for approximate computing. In *DAC*, pages 1–6, 2015.

[14] S. Lee et al. High-level synthesis of approximate hardware under joint precision and voltage scaling. In *DATE*, pages 187–192, 2017.

[15] J. C.-Godínez et al. AxHLS: Design space exploration and high-level synthesis of approximate accelerators using approximate functional units and analytical models. In *ICCAD*, pages 1–9, 2020.

[16] P. Chowdhury and B. C. Schafer. Unlocking approximations through selective source code transformations. In *GLSVLSI*, pages 359–364, 2021.

[17] Z. Cui et al. PreDAC: An efficient framework of pre-refining enhanced design space exploration for approximate computing. In *DAC*, pages 1–7, 2025.

[18] B. Parchamdar and B. C. Schaefer. ADVISOR: Approximate computing-friendly high-level synthesis design space explorer. In *DAC*, pages 1–7, 2025.

[19] B. C. Schafer and Z. Wang. High-level synthesis design space exploration: Past, present, and future. *TCAD*, 39(10):2628–2639, 2020.

[20] S. S.-Douskos et al. Managing performance vs. accuracy trade-offs with loop perforation. In *ESEC/FSE*, pages 124–134, 2011.

[21] S. Lee and A. Gerstlauer. Data-dependent loop approximations for performance-quality driven high-level synthesis. *ESL*, 10(1):18–21, 2018.

[22] S. Li et al. Sculptor: Flexible approximation with selective dynamic loop perforation. In *ICS*, pages 341–351, 2018.

[23] S. Su et al. VECBEE: A versatile efficiency–accuracy configurable batch error estimation method for greedy approximate logic synthesis. *TCAD*, 41(11):5085–5099, 2022.

[24] C. Meng et al. VACSEM: Verifying average errors in approximate circuits using simulation-enhanced model counting. In *DATE*, pages 1–6, 2024.

[25] M. Li et al. A Monte Carlo simulation flow for SEU analysis of sequential circuits. In *DAC*, pages 1–6, 2016.

[26] P. S.-Marbell et al. Exploiting errors for efficiency: A survey from circuits to applications. *ACM Comput. Surv.*, 53(3), June 2020.

[27] J. C.-Godínez et al. Compiler-driven error analysis for designing approximate accelerators. In *DATE*, pages 1027–1032, 2018.

[28] S. Chen et al. A high-accuracy time-efficient error metric model for approximate computing circuits. In *ISVLSI*, pages 94–99, 2024.

[29] G. Zervakis et al. Multi-level approximate accelerator synthesis under voltage island constraints. *TCS II*, 66(4):607–611, 2019.

[30] L. Sathidevi et al. PreAxC: Error distribution prediction for approximate computing quality control using graph neural networks. In *ISQED*, pages 1–7, 2023.

[31] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, page 75, 2004.

[32] J.C. Huang and T. Leng. Generalized loop-unrolling: a method for program speedup. In *ASSET*, pages 244–248, 1999.

[33] Y. Yu. fAST: Flattening abstract syntax trees for efficiency. In *ICSE-Companion*, pages 278–279, 2019.

[34] T. Theodoridis et al. Understanding and exploiting optimal function inlining. In *ASPLOS*, pages 977–989, 2022.

[35] S. Hashemi et al. BLASYS: Approximate logic synthesis using Boolean matrix factorization. In *DAC*, pages 1–6, 2018.

[36] Y.-C. Wei and C.-K. Cheng. Towards efficient hierarchical designs by ratio cut partitioning. In *ICCAD*, pages 298–301, 1989.

[37] H. Jia et al. The latest research progress on spectral clustering. *Neural Computing and Applications*, 24(7):1477–1486, 2014.

[38] J. Shi et al. QUADOL: A quality-driven approximate logic synthesis method exploiting dual-output LUTs for modern FPGAs. In *DATE*, pages 1–7, 2026.

[39] J.-H. R. Jiang and R. K. Brayton. Retiming and resynthesis: A complexity perspective. *TCAD*, 25(12):2674–2686, 2006.

[40] O. Ergin et al. Register packing: Exploiting narrow-width operands for reducing register file pressure. In *MICRO*, pages 304–315, 2004.

[41] C. Wolf. Yosys Open SYnthesis Suite. https://yosyshq.net/yosys/about.html. Accessed: April. 3rd, 2026.

[42] Synopsys, Inc. *Design Compiler*. Synopsys, Inc.

[43] W. Snyder. Verilator 4.0: Open simulation goes multithreaded, 2018.

[44] Nangate, Inc. Nangate 45nm open cell library. https://si2.org/open-cell-library, 2022.

[45] B. C. Schafer and A. Mahapatra. S2CBench: Synthesizable SystemC benchmark suite for high-level synthesis. *ESL*, 6(3):53–56, 2014.

[46] M. Awais. PaderBench. https://git.uni-paderborn.de/mawais/PaderBench, 2019. Accessed: April. 3rd, 2026.

[47] L.-N. Pouchet. PolyBench/C: the polyhedral benchmark suite. https://www.cs.colostate.edu/~pouchet/software/polybench/, 2012. Accessed: July. 23rd, 2026.