

AutoShrink 2.0: Automating Large-Scale Pareto Optimization in HLS via Adaptive and Cost-Aware Search Space Shrinkage

YINGXIN ZENG, Sun Yat-sen University - Shenzhen Campus, China
BINGHAO CHENG, Sun Yat-sen University - Shenzhen Campus, China
JIANWANG ZHAI, Beijing University of Posts and Telecommunications, China
KANG ZHAO, Beijing University of Posts and Telecommunications, China
ZHE LIN*, Sun Yat-sen University - Shenzhen Campus, China

High-level synthesis (HLS) expedites accelerator customization by delivering a high-level hardware programming paradigm and offering a rich set of optimization directives. However, the quality of HLS designs is highly dependent on the designers’ selection of directives in the trade-off between different design metrics, which is non-trivial and can lead to a large design turnaround time. Design space exploration (DSE) serves as a promising solution to this problem, but existing studies on DSE suffer from a lack of scalability for large-scale applications and often spend a substantial portion of the exploration budget on costly design points. To address this problem, this paper proposes *AutoShrink 2.0*, a DSE engine that adaptively shrinks the large search space of an HLS design to gradually retain only high-quality solutions. AutoShrink 2.0 incorporates (1) a comprehensive design space pruning strategy that integrates the joint effect of directives and domain knowledge, (2) an importance-guided Pareto optimization algorithm which dynamically traces the importance ranking of the applied directives and leverages this ranking to navigate the search direction of Pareto frontiers, and (3) a cost-aware post-filtering mechanism to screen out candidates that are computationally expensive yet of limited gain, reducing runtime overhead within a single evaluation. Experimental results on Vitis HLS demonstrate that AutoShrink 2.0 achieves an average distance to reference set (ADRS) of 8.40% across multiple applications with design spaces containing up to 10^{16} design points, outperforming three heuristic methods and three state-of-the-art approaches by 8.01× and 6.24×, respectively. On timeout-prone benchmarks, the cost-aware post-filtering strategy reduces the average timeout ratio by 66.1% compared with the variant without this strategy. Furthermore, design tool transfer experiments from Vitis HLS to Catapult HLS demonstrate the promising cross-tool applicability of AutoShrink 2.0.

CCS Concepts: • **Hardware** → **High-level and register-transfer level synthesis**.

Additional Key Words and Phrases: High-level synthesis, Design space optimization, Quality of results

This work is supported by the Shenzhen Science and Technology Program (No. RCBS20231211090600003, JCYJ20250604175310014), the Guangdong Basic and Applied Basic Research Foundation (No. 2023A1515110769, 2024A1515013155), the National Natural Science Foundation of China (No. 62404257, 62404021), the Beijing Natural Science Foundation (No. L2603021), and the Fundamental Research Funds for the Beijing University of Posts and Telecommunications (No. 2025AI4S20).
*Zhe Lin is the corresponding author.

Authors’ Contact Information: Yingxin Zeng, zengyx29@mail2.sysu.edu.cn, Sun Yat-sen University - Shenzhen Campus, School of Integrated Circuits, Shenzhen, China; Binghao Cheng, chengbh5@alumni.sysu.edu.cn, Sun Yat-sen University - Shenzhen Campus, School of Integrated Circuits, Shenzhen, China; Jianwang Zhai, zhajw@bupt.edu.cn, Beijing University of Posts and Telecommunications, Beijing, China; Kang Zhao, zhaokang@bupt.edu.cn, Beijing University of Posts and Telecommunications, Beijing, China; Zhe Lin, linzh235@mail.sysu.edu.cn, Sun Yat-sen University - Shenzhen Campus, School of Integrated Circuits, Shenzhen, China.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
Manuscript submitted to ACM

Manuscript submitted to ACM

ACM Reference Format:

Yingxin Zeng, Binghao Cheng, Jianwang Zhai, Kang Zhao, and Zhe Lin. 2026. AutoShrink 2.0: Automating Large-Scale Pareto Optimization in HLS via Adaptive and Cost-Aware Search Space Shrinkage. *ACM Trans. Des. Autom. Electron. Syst.* 1, 1 (October 2026), 25 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

1 Introduction

High-level synthesis (HLS) [5] enables designers to describe hardware behavior using high-level languages such as C or C++. Compared to the traditional register-transfer level (RTL) workflow, HLS design methodology delivers a salient speedup of more than $6\times$ [18]. Modern HLS tools [1, 3, 9] provide designers with a series of optimization directives to rapidly tune the hardware micro-architectures without modifying the C/C++ source code. The quantity and diversity of HLS directives constitute a vast configuration space, wherein the selection and configuration of each directive can yield a distinct hardware implementation, thereby influencing the quality of results (QoR). In most cases, HLS design optimization relies heavily on domain expertise to guide directive selection, typically involving a time-consuming iterative process of directive tuning and quality verification until a satisfactory QoR trade-off is reached.

Design space exploration (DSE) [29] has emerged as an effective and concise solution to the problem of automatically finding high-quality directive configurations that meet designers' QoR requirements from a vast number of directives. A major branch of DSE methods adopts online iterative search strategies [10–12, 24, 28, 32, 36, 40], where candidate configurations are repeatedly generated, evaluated through synthesis or estimation, and then used as feedback to guide subsequent exploration. Although these methods can adapt to the target application during exploration, obtaining reliable feedback usually requires a substantial number of evaluated design instances, leading to large runtime overhead under cold-start conditions. This problem is further exacerbated in large design spaces. In contrast, another branch of DSE methods [31, 35, 38] aims to establish a predictive model that is generally applicable to various designs and uses it as a well-defined surrogate model without the need for further refinement, thereby avoiding time-consuming online calibration and ensuring high efficiency. Nevertheless, these pre-trained methods require the construction of high-quality training datasets in advance, and their prediction models often exhibit low accuracy in new application domains, thus revealing poor generalization ability.

Overall, existing HLS DSE methods are constrained by a critical dilemma: when confronted with large and intricate design spaces, they either suffer from low exploration efficiency or demonstrate limited generalization capability. To address this challenge, this paper strives to put forward an efficient yet generalizable DSE framework, specifically targeting large-scale design spaces. The core idea of our method originates from three central observations. *Observation 1: The interaction between directives can sometimes result in a large number of redundant or under-optimized configurations, particularly for large design spaces.* These situations can be identified and pruned away in advance with the guidance of HLS domain knowledge and designers' experience. *Observation 2: Directives exert varying degrees of influence on the QoR of a given application.* Pinpointing the most influential directives and fully exploiting their potential helps guide the search toward faster convergence to optimal solutions. *Observation 3: Certain combinations of directives may lead to excessive runtime that remarkably slows down the optimization process while bringing marginal benefits.* The conversion of C/C++ to RTL involves several problems proven to be NP-complete or NP-hard [13], e.g., the scheduling [23] and binding [25]. Therefore, the execution time of HLS tools can become unaffordable under the situations that make the problem size too large to solve. Proper handling of this issue is critical to improving the efficiency of large-scale DSE. Building upon this foundation, we propose *AutoShrink 2.0*, a novel HLS

Manuscript submitted to ACM

DSE approach that automatically and adaptively shrinks the search space into compact yet high-quality subspaces across the *pre-exploration*, *intra-exploration*, and *post-exploration* stages.

AutoShrink 2.0 builds upon our preliminary work [37], which introduces comprehensive pruning and importance-guided sampling for large-scale HLS DSE. Specifically, comprehensive pruning reduces the design space by leveraging both directive interactions and domain knowledge. Importance-guided sampling ranks directives according to their influence on QoR and assigns each directive a specific change rate based on its ranking, where more important directives are assigned smaller change rates to enable finer-grained exploration in the vicinity of the current Pareto frontier. On top of these strategies, AutoShrink 2.0 further introduces three substantial enhancements. First, we develop a cost-aware filtering mechanism to screen out runtime-expensive design points, which significantly improves exploration quality under fixed runtime constraints. Second, we augment the original importance-guided sampling strategy with a warm-up mechanism to enhance robustness under cold-start conditions. Third, we significantly expand the experimental validation by adding resource-constrained single-objective evaluation, comprehensive ablation studies, parameter sensitivity analysis, evaluation-efficiency trade-off analysis, and cross-tool evaluation. Together, these innovations strengthen the effectiveness of our method and extend its applicability to large-scale HLS DSE scenarios.

In summary, the contributions of this paper are listed as follows:

- We present AutoShrink 2.0, a DSE framework that can be seamlessly integrated into existing DSE flows to guide large-scale Pareto search through progressive design space shrinkage, demonstrating high performance, efficiency and promising cross-tool applicability.
- In the pre-exploration stage, we introduce a comprehensive design space pruning strategy that eliminates redundant and suboptimal design points through the joint exploitation of directive interactions and domain knowledge.
- In the intra-exploration stage, we propose an importance-guided exploration algorithm, which adaptively deduces the importance ranking of directives and progressively narrows the scope of exploration to a reduced yet promising subset of solutions.
- In the post-exploration stage, we develop a cost-aware post-filtering approach to identify and remove highly time-consuming design instances from the potential candidates, enabling efficient and scalable DSE for extremely large-scale scenarios.
- Experimental results show that AutoShrink 2.0 achieves an arithmetic-mean ADRS of 8.40% across diverse applications with design spaces spanning 10^3 to 10^{16} , outperforming generic optimization methods and state-of-the-art (SOTA) DSE approaches by 8.01 $\times$ and 6.24 $\times$, respectively. Compared with our preliminary work [37], AutoShrink 2.0 reduces the arithmetic-mean ADRS from 14.71% to 8.40%, yielding a 1.75 $\times$ improvement. In addition to Vitis HLS, AutoShrink 2.0 remains effective in Catapult HLS.

2 The AutoShrink 2.0 Framework

2.1 Problem Formulation

Given a C/C++ application associated with a set of HLS directives, the objective of our HLS DSE is to approach the *Pareto frontier* P , i.e., the set of optimal design points for multiple QoR metrics, including latency and the resource utilization of look-up table (LUT), block random access memory (BRAM), and digital signal processing (DSP). The DSE iteratively samples promising design points, which collectively form the *approximate Pareto frontier* $\hat{P}$.

Manuscript submitted to ACM

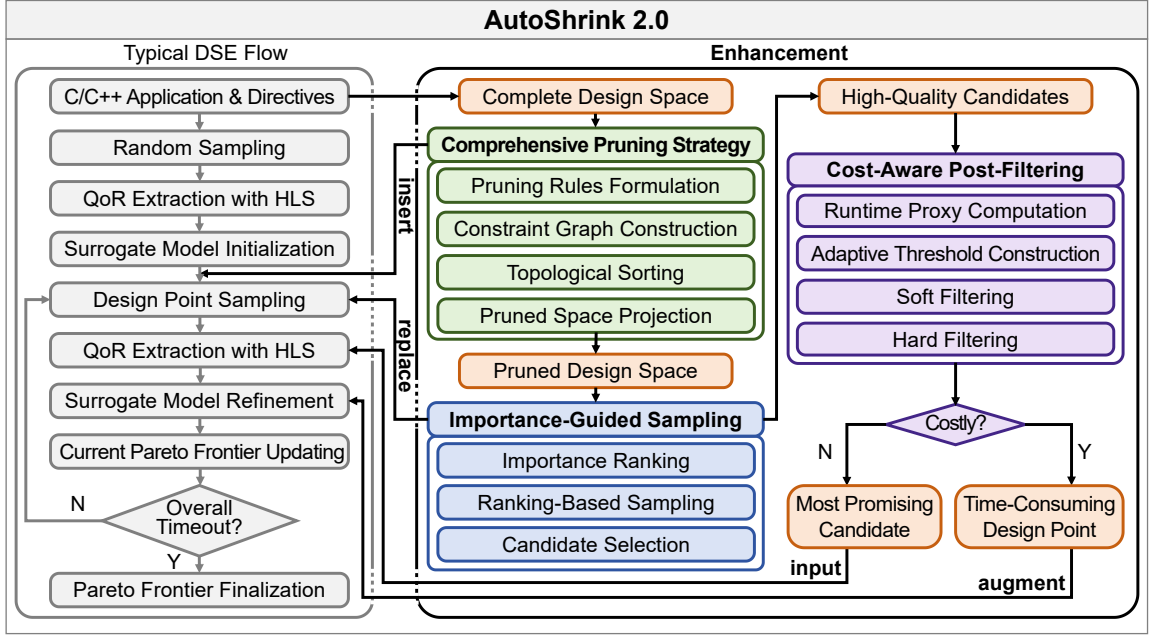


Fig. 1. AutoShrink 2.0 features three components: (1) comprehensive pruning, (2) importance-guided sampling, and (3) cost-aware post-filtering, which can be seamlessly integrated into typical DSE flows to support large-scale scenarios.

The *ADRS* metric is adopted to evaluate the quality of DSE results. Specifically, *ADRS* quantifies the deviation between P and $\hat{P}$, which is formally defined as:

$$ADRS(\hat{P}, P) = \frac{1}{|P|} \sum_{p \in P} \min_{\hat{p} \in \hat{P}} f(\hat{p}, p), \quad (1)$$

$$f(\hat{p}, p) = \max \left\{ 0, \frac{l(\hat{p}) - l(p)}{l(p)}, \frac{t(\hat{p}) - t(p)}{t(p)}, \frac{m(\hat{p}) - m(p)}{m(p)}, \frac{d(\hat{p}) - d(p)}{d(p)} \right\}, \quad (2)$$

where $l(\cdot)$, $t(\cdot)$, $m(\cdot)$, $d(\cdot)$ are functions to retrieve the latency and the resource usage of LUT, BRAM, and DSP, respectively.

2.2 Overall Framework

As shown in Fig. 1 (left), AutoShrink 2.0 adheres to a typical online iterative DSE flow without relying on pre-trained predictors. This DSE flow begins by randomly sampling a subset of design points, extracting their QoR metrics via HLS, and initializing an online surrogate model. Then, the surrogate model and a design point sampling engine work collectively to identify the most promising candidate to explore in the next round. This candidate passes through HLS to obtain QoR metrics, with which the surrogate model and the approximate Pareto frontier are updated. The above steps constitute an *exploration–evaluation–feedback* loop, which iterates until the predefined time budget is reached, ultimately producing the approximate Pareto frontier $\hat{P}$. Without loss of generality, we utilize the widely used and high-performance Bayesian optimization to provide both the surrogate model and the sampler in the flow.

Manuscript submitted to ACM

Table 1. Pruning rules and directives' relationships.

Rule	Constraining Directive	Configuration	Constrained Directive	Configuration
1.1	Function Dataflow	On	Inner Functions' Inline	Off
1.2	Loop Unroll	Partially Unroll	Inner Loops' Flatten	Off
1.3	Loop Pipeline	On	Inner Loops' Unroll	Fully Unroll
2.1	Function Inline	On	Same Function's Dataflow	Off
2.2	Loop Unroll	Fully Unroll	Same Loop's Flatten	Off
			Same Loop's Pipeline	Off
3.1	Loop Unroll (Access Same Array)	List of Unroll Factors	Accessed Array's Partition	Max of Unroll Factors
3.2	Loop Merge	On	Inner Functions' Inline	On

Building on this basic flow, AutoShrink 2.0 seamlessly integrates three customized optimization strategies into the DSE routine while ensuring generality and compatibility with existing methods, as depicted in Fig. 1 (right). Together, these three key components steer the design search throughout the complete exploration process: (1) a comprehensive pruning strategy is inserted during *pre-exploration* to significantly reduce the design space for evaluation; (2) an importance-guided sampling mechanism substitutes the primitive sampling engine, enhancing candidate selection during *intra-exploration*; and (3) a cost-aware post-filtering scheme identifies the most promising candidates and eliminates extremely high-cost design points from the set of potential candidates during *post-exploration*.

3 Comprehensive Pruning Strategy

Motivated by *Observation 1* in Section 1, AutoShrink 2.0 begins by constructing a compact but effective design space. We notice that prior studies [17, 33] have introduced design space pruning techniques for HLS DSE. However, each of these studies is predominantly guided by a simple rule, which proves effective for small design spaces but fails to scale to larger ones. In light of this, we derive a comprehensive set of pruning rules targeting key directives for functions, loops, and arrays. On this basis, we further develop an enhanced pruning strategy capable of efficiently narrowing large design spaces.

3.1 Pruning Rules Formulation

We generalize three types of rules from a variety of directives by carefully profiling multiple applications with Vitis HLS [1]. Each rule identifies a directive that constrains the configuration of one or more directives. In HLS, directives take effect in a top-down manner: from the top function to inner functions, and subsequently to loops and arrays. Accordingly, we establish the constraining relationships of directives in our rules following the same top-down order, as shown in Table 1. Without loss of generality, our pruning rules can be easily extended to other mainstream HLS tools such as Catapult HLS [7], LegUp [3], and Bambu [9], as our focus is on generic program constructs and HLS principles that are consistent across different tools.

Rule-1: The adoption of some directives precludes the specification of other directives. (1) *Outer functions' dataflow disables inner functions' inlining*: Function dataflow overlaps the execution of inner functions, which requires preserving their boundaries. This prohibits the inlining of inner functions, as it embeds their bodies into the parent and removes such boundaries. (2) *Partial unrolling of an outer loop invalidates inner loop flattening*: Partial unrolling of an outer loop duplicates inner loop structures, producing a non-perfectly nested loop. This conflicts with the flatten directive applied to the inner loop, which requires perfect nesting to fuse multiple nested loops into a single flat loop. (3) *Outer loop pipelining enforces full unrolling of inner loops*: Outer-loop pipelining aims to schedule all operations

Manuscript submitted to ACM

holistically across the loop body. Accordingly, inner loop structures are eliminated through complete unrolling, rendering any further unrolling configurations in vain.

Rule-2: C/C++ components can be removed by certain directives, causing other directives on the same component to be ineffective. We observe that certain directives will reorganize specific program constructs, thereby influencing the applicability of other directives targeting the original structures. (1) *Function inline invalidates directives on the same function*: When a function is inlined, its body is merged into the parent function, eliminating the original function instance and causing all directives that are applied to that function to become invalid. (2) *Full loop unrolling prohibits other directives from being applied at the same loop level*: When a loop is fully unrolled, it is transformed into discrete operations, and the loop itself no longer exists. Therefore, directives like loop pipelining and flattening that previously operated on this loop are rendered ineffective.

Rule-3: Specific directives should be coordinated to achieve performance gain. (1) *Loop unroll factor should match the partition factor of accessed arrays*: Loop unrolling increases parallelism along with memory bandwidth demand. Optimal performance is generally achieved when the partition factor applied to an array dimension matches the loop unroll factor used to access that dimension. Otherwise, parallelism from loop unrolling is limited by insufficient memory bandwidth.

(2) *Loop merging across functions requires function inlining*: Loop merging is possible only when the target loops reside at the same code hierarchy. When loops are located in different functions, function inlining is typically required to bring them into the same scope. For example, inlining $\{function1\{loop1\}, function2\{loop2\}\}$ allows *loop1* and *loop2* to be merged under a *loop_merge* directive.

3.2 Pruning Strategy

The aforementioned pruning rules define intricate constraints among directives. We propose a strategy to orchestrate these rules so as to transform an arbitrary design point into a valid one that satisfies all applicable rules, thereby yielding a reduced design space for each application. This pruning strategy consists of three stages.

Constraint graph construction. Given an HLS design of interest, we first extract all directives applied to this design, and construct a *constraint graph* $G = (V, E)$ that depicts the complete directive set, wherein each node $v \in V$ is a directive instance in the directive set and each edge $e \in E$ represents a directed relation from the constraining directive to the directive being constrained as defined in Table 1.

Topological sorting. Since the relationships among directives in the pruning rules are established in a top-down order, the edges of the constraint graph consistently follow the same order, pointing from functions to loops and then to arrays. This prevents any reverse edges from inner to outer hierarchies, ensuring that the constraint graph is a directed acyclic graph (DAG). On this foundation, we perform *topological sorting* [16] on the constraint graph, which is only applicable to DAGs and returns an ordered list of nodes, ensuring that the sink nodes are located after their source nodes. In other words, passing through the ordered list ensures strict adherence to the pruning rules defined by the constraint graph.

Pruned space projection. After obtaining the topologically sorted list of directives, we seek to project an arbitrary design point into the pruned space. Considering the applicable directives, their specified configurations and all possible configurations, we traverse the sorted list of directives sequentially and specify their configurations under the pruning rules gradually. Unconstrained directives can retain their original configurations. However, for constrained directives, we filter out invalid configurations according to pruning rules, keep only the feasible subset, and finally choose the configuration closest to the pre-set one based on cosine similarity. This process dynamically projects a design point

Manuscript submitted to ACM

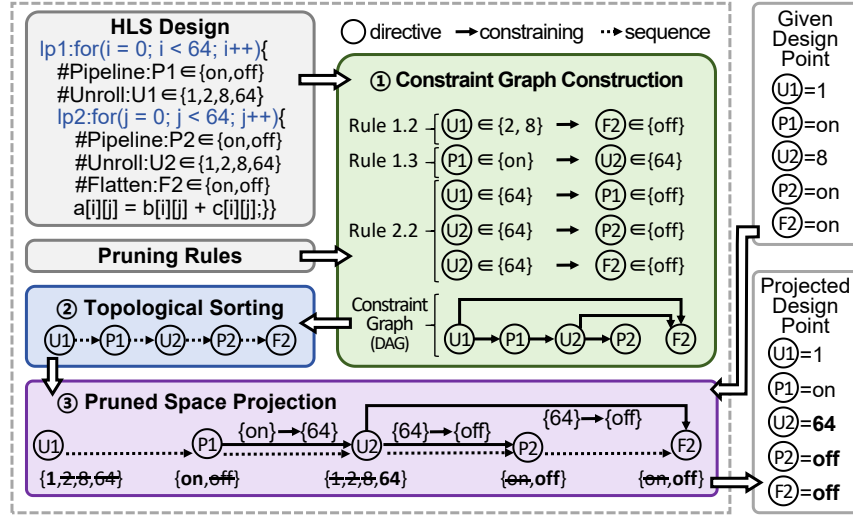


Fig. 2. A simplified example of the comprehensive pruning strategy.

into the pruned space rather than enumerating all valid design points, which significantly reduces computational cost and supports fast pruning for large-scale design spaces at runtime.

For ease of understanding, we provide a simplified example in Fig. 2. Given an arbitrary design point $\langle U1 = 1, P1 = on, U2 = 8, P2 = on, F2 = on \rangle$, we identify the applicable pruning rules and build the constraint graph with five directive nodes and five edges. As the graph is acyclic, we perform a topological sort on the directive nodes and then apply the pruned space projection. In this order, $U1$ and $P1$ remain unchanged; $P1 = on$ forces $U2$ to be fully unrolled with a factor of 64, which eliminates *loop2* and makes both $P2$ and $F2$ inapplicable (both set to *off*). Finally, the projected design point becomes $\langle U1 = 1, P1 = on, U2 = 64, P2 = off, F2 = off \rangle$.

4 Importance-Guided Sampling

In the exploration stage, existing DSE methods [10, 11, 24, 28] typically rely on simple sampling strategies such as random sampling to find a subset of design points as promising candidates. This practice renders large-scale DSE extremely slow, as high-quality design points are rarely identified due to the low probability of random selection. To address this limitation, we propose substituting the sampling engine with an enhanced exploration algorithm, which comprises three steps: *importance ranking*, *ranking-based sampling* and *candidate selection*, as depicted in Algorithm 1.

4.1 Importance ranking

After profiling a large number of HLS applications, we observe that directives exert varying degrees of influence on the final QoR of the synthesized hardware. This motivates us to develop a sampling engine capable of discerning and leveraging the unique impact of each directive on QoR. To start with, we employ the functional analysis of variance (fANOVA) [14]. Let $\mathcal{O} = \{\text{latency, LUT, DSP, BRAM}\}$ denote the set of QoR objectives provided by HLS tools. For each single objective $o \in \mathcal{O}$, a separate random forest model $f^{(o)}(\mathbf{x})$ is constructed to predict its behavior, where $\mathbf{x} = (x_1, x_2, \dots, x_n)$ represents a design point, n is the number of directives, and x_i denotes the configuration of the i -th HLS directive (categorical values such as *on/off* are encoded numerically, e.g., as 1 and 0). Specifically, fANOVA

Algorithm 1: Importance-Guided Sampling

Input: P, P_{last} : current/last-round Pareto frontier; S_p : flag for using pruned space (initially true); t, T_{total} : elapsed and total search time; R_{warmup} : fraction of the total search time allocated to warm-up; N : number of samples per round; $cr_{\text{min}}, cr_{\text{min}}^*$: current and final minimum change rate; w : warmup progress, defined as current time divided by total planned warmup time; C, C_{max} : counter of consecutive rounds without Pareto frontier improvement

Output: Most promising candidate $\mathbf{x}^*$

```

1 // Stage 1: Importance Ranking
2 Estimate importance scores  $[I_1, I_2, \dots, I_n]$  for directives  $[x_1, x_2, \dots, x_n]$  using fANOVA (Eqs. 4–7);
3  $R \leftarrow$  list of directives ranked by importance scores (descending);
4 // Stage 2: Ranking-Based Sampling
5  $w \leftarrow \min\left(\frac{t}{R_{\text{warmup}} T_{\text{total}}}, 1\right)$ ;
6  $cr_{\text{min}} \leftarrow cr_{\text{min}}^* + (1 - cr_{\text{min}}^*) \cdot \frac{1 + \cos(\pi w)}{2}$ ;
7 Initialize  $[\mathbf{x}] \leftarrow []$  (list of sampled design points);
8 for  $j = 1$  to  $N$  do
9   Pick  $\mathbf{x} \in P$  at random;
10  for  $k = 1$  to  $|R|$  do
11     $cr_k \leftarrow cr_{\text{min}} + (1 - cr_{\text{min}}) \cdot \frac{k-1}{|R|-1}$ ;
12    if  $cr_k \geq \text{Random}(0, 1)$  then
13      Assign  $R[k]$  in  $\mathbf{x}$  a random value from its value set;
14  if  $S_p == \text{true}$  then
15    Project  $\mathbf{x}$  to nearest point in pruned space (Sec. 3.2);
16  Append  $\mathbf{x}$  to  $[\mathbf{x}]$ ;
17 // Stage 3: Candidate Selection
18  $QoR_{\text{pred}} \leftarrow \text{predict}([\mathbf{x}])$ ;
19  $A \leftarrow \text{acquisition}(QoR_{\text{pred}})$ ;
20  $\mathbf{x}^* \leftarrow \arg \max_{\mathbf{x} \in [\mathbf{x}]} A$ ;
21  $C \leftarrow (P == P_{\text{last}}) ? (C + 1) : 0$ ;
22  $S_p \leftarrow S_p \wedge (C < C_{\text{max}})$ ;
23 return  $\mathbf{x}^*$ 

```

approach decomposes $f^{(o)}(\mathbf{x})$ into additive components:

$$f^{(o)}(\mathbf{x}) = f_0^{(o)} + \sum_{i=1}^n f_i^{(o)}(x_i) + \sum_{i < j} f_{ij}^{(o)}(x_i, x_j) + \dots + f_{1,2,\dots,n}^{(o)}(x_1, x_2, \dots, x_n). \quad (3)$$

Here, $f_0^{(o)}$ denotes the global mean of objective o , while $f_i^{(o)}(x_i)$ and $f_{ij}^{(o)}(x_i, x_j)$ represent the individual and pairwise effects of directives, respectively. The random forest models are iteratively updated during exploration [14]. Based on this, the importance of directive x_i for objective o is quantified as the ratio of the variance attributed to its individual effect to the total variance:

$$I_i^{(o)} = \frac{\text{Var}[f_i^{(o)}(x_i)]}{\text{Var}[f^{(o)}(\mathbf{x})]}, \quad (4)$$

where Θ_i denotes the value set of directive x_i , and the variance attributed to directive i is defined as

$$\text{Var}[f_i^{(o)}(x_i)] = \frac{1}{|\Theta_i|} \int (f_i^{(o)}(x_i))^2 dx_i. \quad (5)$$

AutoShrink 2.0: Automating Large-Scale Pareto Optimization in HLS via Adaptive and Cost-Aware Search Space Shrinkage

9

And the total variance of QoR, $\text{Var}[f^{(o)}(\mathbf{x})]$, is computed by

$$\text{Var}[f^{(o)}(\mathbf{x})] = \sum_{\substack{S \subseteq \{1, \dots, n\} \\ S \neq \emptyset}} \mathbb{E}_{\mathbf{x}_S} [(f_S^{(o)}(\mathbf{x}_S))^2]. \quad (6)$$

Here, S denotes any subset of input variables whose size ranges from 1 to n (the total number of directives), and $\mathbb{E}_{\mathbf{x}_S}[\cdot]$ denotes the expectation with respect to the distribution of the variable vector $\mathbf{x}_S$. Applying the above calculation independently to each objective $o \in \mathcal{O}$ yields an objective-specific importance $I_i^{(o)}$. We then average these scores with equal weights to obtain a unified importance I_i for directive x_i :

$$I_i = \frac{1}{|\mathcal{O}|} \sum_{o \in \mathcal{O}} I_i^{(o)}. \quad (7)$$

After obtaining the importance values of all n directives, we rank them in descending order of importance.

4.2 Ranking-based sampling

We then propose a customized sampling strategy leveraging the importance ranking of directives. Firstly, we keep updating the current Pareto frontier with the explored design points. Subsequently, to identify a promising design point for sampling, we fix the entire directive set to an instance in the current Pareto frontier. Finally, new design candidates are sampled by modifying the directive configurations of selected Pareto instances according to a set of *change rates*, enabling exploration in the vicinity of the current optimal region. AutoShrink 2.0 maintains two types of change rates to control the sampling behavior, as illustrated below.

Per-directive change rate. For each inserted directive i , a *change rate* cr_i is assigned based on its importance ranking: a directive with higher importance is assigned a lower probability of change, with the change rate increasing as importance decreases. Specifically, the change rates are linearly distributed across the ranked directive list, ranging from a minimum change rate ($cr_{\min}$) to fully random (100%). In this way, we extend the current Pareto frontier by fixing important directives to their configurations on the frontier with higher probabilities, while granting the remaining directives greater flexibility for exploration.

Minimum change rate. We allow the minimum change rate $cr_{\min}$ to decrease with the search progress. This design is motivated by the cold-start stage, where only limited design points have been explored and the resulting Pareto frontier may not yet cover high-quality regions of the design space. If the search overly relies on importance-guided sampling at this stage, it may become easily trapped in local optima. To circumvent this situation, we assign a relatively large $cr_{\min}$ in early iterations to encourage broader exploration during a warm-up stage and gradually reduce it as the Pareto frontier becomes more reliable. Specifically, $cr_{\min}$ decreases from 100% to a predefined minimum value $cr_{\min}^*$ following a cosine schedule inspired by the cosine annealing strategy in SGDR [22]. Let

$$w = \frac{t}{R_{\text{warmup}} T_{\text{total}}}, \quad (8)$$

where w denotes the normalized warm-up progress, t is the elapsed search time, R_{warmup} is the fraction of the total search time allocated to the warm-up stage, and T_{total} is the total search time. Then $cr_{\min}$ is defined as

$$cr_{\min} = \begin{cases} cr_{\min}^* + (100\% - cr_{\min}^*) \frac{1 + \cos(\pi w)}{2}, & 0 \leq w \leq 1, \\ cr_{\min}^*, & w > 1. \end{cases} \quad (9)$$

Manuscript submitted to ACM

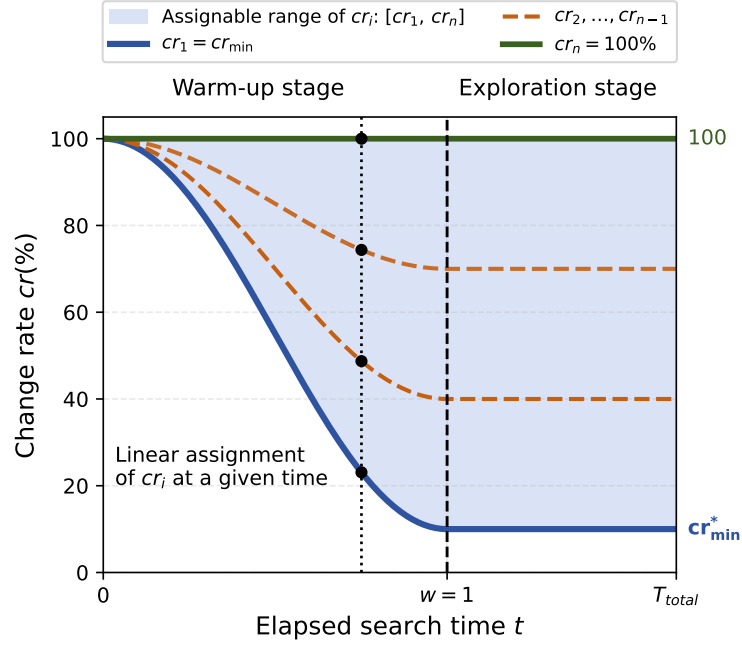


Fig. 3. Cosine-scheduled evolution and rank-based assignment of directive change rates. During the warm-up stage, the lower bound $cr_{\min}$ decreases smoothly from 100% to the target value $cr_{\min}^*$, and the directive-level change rates cr_i are linearly interpolated within $[cr_{\min}, 100\%]$ according to directive importance. After warm-up, $cr_{\min}$ remains fixed at $cr_{\min}^*$, and the assigned cr_i values remain unchanged.

As illustrated in Fig. 3, $cr_{\min}$ decreases during the warm-up stage and serves as the dynamic lower bound for assigning the directive-specific change rates cr_i . At each time step, the cr_i values therefore also change with the search progress. In this way, the sampling behavior gradually shifts from broader exploration toward stronger exploitation of the current Pareto frontier.

This sampling flow is repeated N times, each time identifying one design point to add to the candidate set, as shown in Lines 5–16 of Algorithm 1.

4.3 Candidate selection

In each round of exploration, we collect a large number of design points from the ranking-based sampling stage. These points are then evaluated by the surrogate model within Bayesian optimization, ranked by their acquisition function values derived from predicted QoR, and finally, the top-ranked design point is regarded as the most promising candidate, which will be further verified through HLS. When combined with pruning, each sampling point is projected onto the pruned design space (see Section 3.2), ensuring that exploration remains confined to this reduced space. To further enhance the efficacy of sampling, we monitor the QoR in each round of exploration and track the number of consecutive rounds without improvement. If the Pareto frontier remains unchanged for more than $C_{\max}$ rounds, we consider it unlikely that better solutions exist within the pruned space, so the exploration is redirected to the complete design space to search for additional optimization opportunities.

Manuscript submitted to ACM

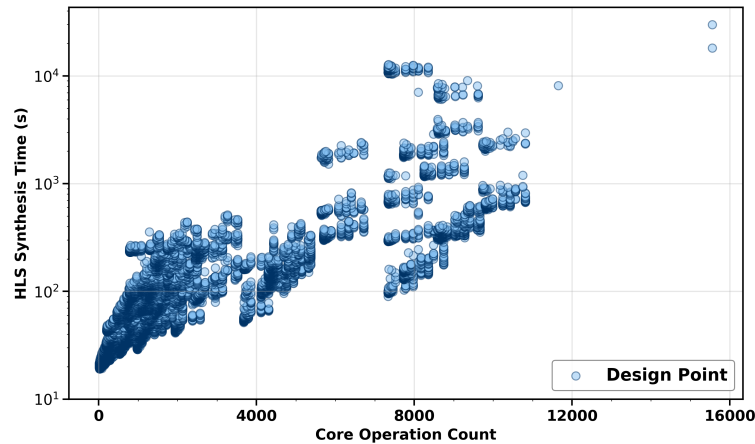


Fig. 4. Core operation count vs. HLS synthesis time for 17,334 design points of GEMM.

5 Cost-Aware Post-Filtering

After the prior two stages, i.e., pruning and sampling, AutoShrink 2.0 has generated a set of potentially promising candidates. However, evaluating these candidates through HLS can still be expensive. Typically, overly aggressive directive configurations, such as fully unrolling loops or completely partitioning arrays, may substantially increase the scheduling workload and lead to a surge of HLS runtime, while providing only marginal QoR improvements. Such evaluations waste a large fraction of the limited DSE budget. To address this issue, we propose a cost-aware post-filtering mechanism that is able to filter out potentially expensive candidates before launching full HLS evaluation. The key idea is to first use a lightweight proxy to assess the runtime cost of each candidate, and then use real runtime feedback from observed timeout cases to adaptively decide whether runtime-expensive candidates should still be evaluated. The overall module consists of two components: runtime proxy computation and a filtering mechanism.

5.1 Runtime proxy computation

To identify expensive candidates before running HLS, we first need a runtime proxy that can quickly indicate the runtime cost of a design point in advance. As reported in [1], the HLS runtime is highly correlated with the number of objects to schedule, which determines the overall scheduling workload. Since the internal scheduling objects of HLS tools are difficult to extract directly before synthesis, we approximate the scheduling workload using a set of representative core operations in the intermediate representation (IR), which reflect the computational and memory-access complexity that the scheduler needs to handle. These operations include arithmetic operations, memory-access operations, and address-computation operations, such as *fadd*, *fmul*, *fsub*, *add*, *mul*, *load*, *store*, *getelementptr*, and *bitcast*. The total number of these core operations is used as the runtime proxy, denoted as cnt_o .

To validate whether cnt_o is an effective runtime proxy, we examine the relationship between cnt_o and HLS runtime based on a set of design instances. Fig. 4 shows the evaluation results with the *gemm* application as an example. The cnt_o exhibits a positive trend with the logarithm of HLS runtime, suggesting that design points with larger cnt_o are more likely to incur significantly longer HLS runtime. This result suggests that cnt_o can serve as a coarse early indicator of HLS runtime.

Manuscript submitted to ACM

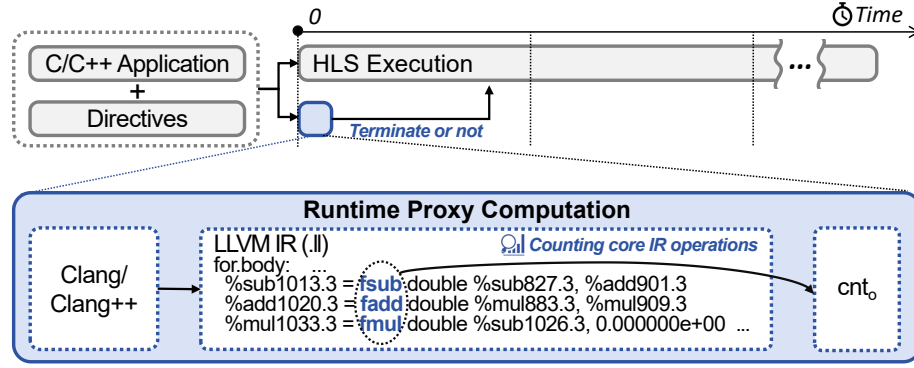
Fig. 5. Workflow for Extracting cnt_o from LLVM IR.

Fig. 5 illustrates how cnt_o is extracted in our implementation. Given a candidate configuration, we apply loop unrolling directives to the C/C++ program and use Clang and LLVM [19] to generate the IR of the design. We then profile the IR program and count the selected core operations to obtain cnt_o . This proxy computation is lightweight and can be overlapped with the HLS evaluation (QoR extraction) process, introducing negligible timing overhead.

5.2 Filtering Mechanism

After obtaining cnt_o , an intuitive filtering method is to set a fixed threshold and remove candidates whose cnt_o exceeds it. However, this naive thresholding has two limitations. First, a predetermined threshold may not generalize well across diverse C/C++ applications, because different C/C++ programs can exhibit different synthesis behavior even under similar operation counts. Second, simply discarding all over-threshold candidates can be overly aggressive, since cnt_o is only a coarse proxy of HLS runtime. Some over-threshold candidates may still be efficiently processed by HLS and yield high-quality QoR. The strict elimination of these candidates may degrade DSE effectiveness. To address these two limitations, we construct an application-specific threshold from observed timeout feedback and further apply a probabilistic filtering policy for over-threshold candidates. The filtering mechanism consists of three parts: adaptive threshold construction, soft filtering, and hard filtering. The overall procedure of the proposed filtering mechanism is summarized in Algorithm 2.

Adaptive threshold construction. First, we construct a flexible operation-count threshold that can adapt to different application scenarios. At the beginning, all candidates are evaluated by HLS. The threshold is constructed only after the first timeout is observed. Once a candidate times out, its cnt_o is added to a timeout history set S_{to} for the current application. Based on this set, we define the application-specific operation-count threshold as

$$Th_o = \min(S_{to}). \quad (10)$$

This definition uses the smallest cnt_o among observed timeout candidates as the current runtime-risk boundary. Therefore, each application can establish its own threshold based on actual timeout behavior, avoiding manual tuning or a fixed threshold.

Soft filtering. For over-threshold candidates, we do not immediately discard them. Instead, they are evaluated by HLS with a certain probability, denoted as P_{eval} . This probabilistic evaluation strategy is referred to as soft filtering. Specifically, for candidates with $cnt_o(x) \geq Th_o$, HLS evaluation is performed with probability P_{eval} ; otherwise, they

Algorithm 2: Adaptive Optcnt-Guided Candidate Filtering

Symbols: $\mathbf{x}$: current candidate configuration; $\text{cnt}_o(\mathbf{x})$: core operation count of candidate $\mathbf{x}$; S_{to} : set of $\text{cnt}_o(\mathbf{x})$ of timed-out candidates; Th_o : operation-count threshold; P_{eval} : probability of being selected for HLS evaluation, initialized to 1; Δp : step size for updating P_{eval} ; I_e : indicator of whether candidate $\mathbf{x}$ is successfully evaluated by HLS; *timeout*: indicator of whether the HLS runtime of $\mathbf{x}$ exceeds the predefined limit.

Input: $\mathbf{x}$, S_{to} , P_{eval} , Δp

Output: $QoR(\mathbf{x})$: QoR value assigned to candidate $\mathbf{x}$

```

1 Start HLS( $\mathbf{x}$ ) and obtain  $\text{cnt}_o(\mathbf{x})$  while HLS( $\mathbf{x}$ ) is running;
2  $I_e \leftarrow \text{True}$ ;
3 if  $S_{to} \neq \emptyset$  then
4    $Th_o \leftarrow \min(S_{to})$ ; // Adaptive threshold construction
5   if  $\text{cnt}_o(\mathbf{x}) \geq Th_o$  then
6     if  $P_{eval} < \text{Random}(0, 1)$  then
7       Terminate HLS( $\mathbf{x}$ );
8        $I_e \leftarrow \text{False}$ ; // Filtering
9 if  $I_e$  then
10   Wait until HLS( $\mathbf{x}$ ) completes or reaches the timeout limit;
11   if timeout occurs then
12     Terminate HLS( $\mathbf{x}$ );
13     if  $S_{to} = \emptyset$  or  $\text{cnt}_o(\mathbf{x}) \geq Th_o$  then
14        $P_{eval} \leftarrow \max(0, P_{eval} - \Delta p)$ ;
15      $S_{to} \leftarrow S_{to} \cup \{\text{cnt}_o(\mathbf{x})\}$ ;
16      $I_e \leftarrow \text{False}$ ;
17   else
18     if  $S_{to} \neq \emptyset$  and  $\text{cnt}_o(\mathbf{x}) \geq Th_o$  and  $P_{eval} > 0$  then
19        $P_{eval} \leftarrow \min(1, P_{eval} + \Delta p)$ ;
20 if  $I_e$  then
21    $QoR(\mathbf{x}) \leftarrow \text{true QoR of } \mathbf{x}$ ;
22 else
23    $QoR(\mathbf{x}) \leftarrow \text{penalty value}$ ;
24 return  $QoR(\mathbf{x})$ ;

```

are filtered out and treated as timed-out (Algorithm 2, Lines 2–8). The probability P_{eval} is dynamically updated based on evaluation outcomes, with a step size Δp . If an evaluated over-threshold candidate results in a timeout, P_{eval} is decreased by Δp . Otherwise, if it completes successfully, P_{eval} is increased by Δp (Algorithm 2, Lines 9–19). The update rule is defined as:

$$P_{eval} = \begin{cases} \max(0, P_{eval} - \Delta p), & \text{if a timeout occurs,} \\ \min(1, P_{eval} + \Delta p), & \text{otherwise.} \end{cases} \quad (11)$$

where Δp controls the tolerance of the filtering mechanism to timeout events. As more timeout samples are observed, P_{eval} gradually decreases, leading to an adaptive tightening of exploration in high-risk regions. For applications with frequent timeouts, this mechanism quickly suppresses high-cost evaluations. In contrast, for applications with few timeouts, P_{eval} can recover, reducing the risk of filtering out potentially high-quality candidates.

Hard filtering. When P_{eval} reaches 0, it has no opportunity to increase since no over-threshold candidate is evaluated, and all candidates with $cnt_o(\mathbf{x}) \geq Th_o$ are thereafter treated as timed-out, corresponding to a hard filtering stage. Entering this stage indicates that the application is highly prone to timeouts, and further exploration in high-cost regions is unlikely to be beneficial. Although a small number of potentially good candidates may be filtered out, the computational savings outweigh the loss in timeout-prone scenarios, thereby improving overall exploration efficiency.

For timed-out candidates, including those treated as timed-out, a worst-case penalty is assigned to their QoR and incorporated into surrogate training. Although their true QoR is unavailable, this penalty feedback enables the surrogate to learn which regions are likely to be runtime-expensive, thereby discouraging similar recommendations in subsequent iterations.

Overall, this mechanism leverages runtime feedback to filter out runtime-expensive candidates, thereby improving the efficiency of DSE under limited time budgets.

6 Experimental Results

6.1 Experimental Setup

AutoShrink 2.0 is implemented based on OpenBox [20], with Bayesian optimization [30] serving as its optimization backbone. Specifically, it adopts a Gaussian process surrogate model (RBF kernel) together with the multi-objective max-value entropy search acquisition function [2]. The complete framework is implemented on a workstation with dual Intel Xeon Gold 6230, 251 GiB memory, and Ubuntu 20.04, targeting AMD/Xilinx ZCU102 FPGA board with Vitis HLS 2022.2. The key parameters of AutoShrink 2.0 are configured as follows: $cr_{min}^* = 10\%$, $R_{warmup} = 40\%$, $N = 5000$, and $C_{max} = 100$ for importance-guided sampling, and $\Delta p = 1/2$ for cost-aware post-filtering.

We compare AutoShrink 2.0 with six representative DSE methods. These include three generic optimization methods: genetic algorithm (GA) [6], simulated annealing (SA) [34], and reinforcement learning (RL) [15], as well as three customized DSE methods as SOTA: Sherlock [12], Prospector [24], and ScaleHLS's exploration engine [36]. Note that Prospector can be regarded as the bare Bayesian optimization (BO) engine of AutoShrink 2.0, excluding the proposed comprehensive pruning, importance-guided sampling, and cost-aware post-filtering techniques. Among the evaluated baselines, Sherlock provides publicly available code, which is directly adapted to Vitis HLS for evaluation. The remaining methods are reimplemented according to their original papers and integrated into the same evaluation flow. In particular, ScaleHLS is adapted by mapping its transformation parameter space to a directive-level design space while retaining its Pareto-guided DSE strategy. All baselines use the hyperparameter settings reported in their original papers or recommended configurations when available, and explore the original full design spaces following their respective search procedures.

We conduct experiments on 12 representative benchmark applications from PolyBench [26], MachSuite [27], and Rosetta [39], covering 8 distinct domains such as linear algebra, molecular dynamics, and neural networks. QoR metrics are normalized to $[0, 1]$ using min-max scaling for each benchmark. We construct design spaces from a broad range of HLS directives, including (1) function-level directives: inline, dataflow, and loop merge; (2) loop-level directives: pipeline, unroll, and flatten; and (3) array-level directive: array partition. These directives define benchmark design spaces ranging from 7.78×10^3 to 1.08×10^{16} , as shown in Table 2. For each benchmark, all evaluated DSE methods start from the same set of 100 randomly sampled design points and are evaluated under a 24-hour runtime budget. To mitigate randomness, three independent runs are conducted for each benchmark, except for aes and gemm, which are evaluated over five independent runs due to their relatively large run-to-run variations. The initial design-point

Manuscript submitted to ACM

AutoShrink 2.0: Automating Large-Scale Pareto Optimization in HLS via Adaptive and Cost-Aware Search Space Shrinkage

15

Table 2. Benchmark design space sizes and ADRS comparison of AutoShrink 2.0 and baseline methods using Vitis HLS.

Benchmark	Design Space Size		ADRS (% Arithmetic Mean $\pm$ Standard Deviation)						
	Original	Pruned (ours)	GA	RL	SA	Sherlock [12]	ScaleHLS [36]	Prospector [24]	AutoShrink 2.0 (ours)
aes	1.24×10^{12}	2.01×10^7	122.44 $\pm$ 41.92	128.68 $\pm$ 41.81	44.61 $\pm$ 12.57	97.38 $\pm$ 22.21	65.54 $\pm$ 12.40	46.95 $\pm$ 10.70	22.67$\pm$9.98
atax	5.31×10^6	7.40×10^3	54.55 $\pm$ 12.22	51.75 $\pm$ 11.98	27.66 $\pm$ 14.03	54.81 $\pm$ 12.95	33.10 $\pm$ 11.68	22.90 $\pm$ 8.17	3.24$\pm$2.27
backprop	2.41×10^{14}	9.75×10^7	62.32 $\pm$ 21.54	65.76 $\pm$ 23.98	30.69 $\pm$ 19.01	51.51 $\pm$ 21.51	38.79 $\pm$ 14.40	21.56 $\pm$ 7.03	7.88$\pm$1.75
bicg	6.37×10^7	1.11×10^4	85.92 $\pm$ 8.42	70.03 $\pm$ 33.05	48.02 $\pm$ 28.39	86.27 $\pm$ 8.64	54.30 $\pm$ 6.70	51.86 $\pm$ 23.43	9.89$\pm$5.68
fft	1.08×10^{16}	1.37×10^{10}	82.00 $\pm$ 14.66	82.20 $\pm$ 8.99	41.27 $\pm$ 6.59	70.02 $\pm$ 8.92	45.92 $\pm$ 9.07	39.72 $\pm$ 11.84	18.93$\pm$3.07
gemm	1.42×10^7	3.01×10^4	81.81 $\pm$ 11.30	84.10 $\pm$ 13.26	63.82 $\pm$ 11.32	94.49 $\pm$ 19.52	67.71 $\pm$ 20.18	86.76 $\pm$ 24.51	11.32$\pm$6.49
gesummv	9.95×10^5	1.20×10^3	31.74 $\pm$ 17.78	21.08 $\pm$ 11.11	10.78 $\pm$ 2.06	24.83 $\pm$ 5.17	15.69 $\pm$ 11.55	14.89 $\pm$ 4.89	0.79$\pm$0.29
md_knn	2.10×10^5	9.00×10^1	12.32 $\pm$ 1.41	8.40 $\pm$ 4.33	10.09 $\pm$ 0.94	12.45 $\pm$ 1.51	7.69 $\pm$ 5.85	1.46 $\pm$ 0.80	0.27$\pm$0.11
mvt	1.91×10^8	1.11×10^4	66.10 $\pm$ 5.58	73.29 $\pm$ 38.82	42.98 $\pm$ 7.32	86.04 $\pm$ 23.79	55.22 $\pm$ 14.57	38.10 $\pm$ 3.02	11.96$\pm$3.47
spam_filter	4.29×10^9	1.62×10^6	231.33 $\pm$ 82.06	218.69 $\pm$ 73.37	130.05 $\pm$ 58.76	172.62 $\pm$ 88.00	159.69 $\pm$ 25.64	31.58 $\pm$ 17.99	1.89$\pm$1.05
spmv	7.78×10^3	3.00×10^1	17.90 $\pm$ 1.87	11.19 $\pm$ 0.81	10.96 $\pm$ 0.79	13.00 $\pm$ 1.98	10.96 $\pm$ 0.79	1.61 $\pm$ 0.74	0.00$\pm$0.00
stencil3d	1.22×10^{10}	7.13×10^6	115.31 $\pm$ 22.48	108.25 $\pm$ 19.82	73.99 $\pm$ 12.35	101.68 $\pm$ 4.64	61.72 $\pm$ 25.47	49.02 $\pm$ 18.41	11.99$\pm$3.28
Arith. Mean	/	/	80.31	76.95	44.58	72.09	51.36	33.87	8.40
Geo. Mean	/	/	61.56	54.77	33.97	56.01	38.38	21.27	5.17

sets differ across independent runs, resulting in different exploration trajectories. All evaluated DSE methods use a 30-minute timeout for each HLS evaluation to avoid excessive evaluation time for individual design points. Timed-out configurations in AutoShrink 2.0, Prospector, and Sherlock are assigned worst-case penalty values to update their surrogate models. In contrast, the remaining methods simply discard timed-out configurations because their search procedures do not utilize such feedback, while adding them to a blacklist to prevent repeated sampling.

To evaluate the quality of DSE results, we adopt the metric ADRS, as defined in Eq. 1. The ADRS compares the reference set with the precise Pareto frontier and a smaller ADRS indicates better quality. However, as benchmarks' design spaces reach up to 1.08×10^{16} , it is infeasible to traverse the complete design space to retrieve the precise Pareto frontier. To address this problem, we construct a reference Pareto frontier for each benchmark by aggregating all design points evaluated by all DSE methods and extracting the Pareto frontier from the aggregated design points. Note that the reference Pareto frontiers are updated dynamically as new methods are included. Consequently, the results reported in our preliminary work [37] are recomputed in this extended version because the reference Pareto frontiers evolve as additional design points become available.

6.2 Multi-Objective DSE Performance

We conduct multi-objective DSE experiments with AutoShrink 2.0 and the above methods to evaluate their ability to discover high-quality Pareto frontiers. As shown in Table 2, AutoShrink 2.0 delivers $8.01\times/9.68\times$ improvements in arithmetic/geometric mean ADRS over generic optimization methods (GA, RL, SA), and $6.24\times/7.46\times$ improvements over SOTA approaches (ScaleHLS, Sherlock, Prospector), confirming its effectiveness in large design spaces.

The six methods under comparison can be divided into two categories. The first category emphasizes the enhancement of surrogate models, with Sherlock and Prospector as representatives. They show arithmetic mean ADRS values of 72.09% and 33.87%, respectively, much worse than AutoShrink 2.0's 8.40%. These methods are more likely to waste the sampling budget on redundant or low-quality design points due to the lack of pruning. As a result, their surrogate models are trained with a large number of low-quality data, which undermines prediction accuracy and consequently reduces their ability to identify promising design points. The second category, including GA, RL, SA, and ScaleHLS, focuses on improving search engines during exploration. These methods yield arithmetic mean ADRS values between 44.58% and 80.31%, all worse than that of AutoShrink 2.0. Their search engines do not consider the varying impact of

Manuscript submitted to ACM

Table 3. Resource-constrained minimum-latency comparison under a 24-hour search budget.

Benchmark	AutoShrink 2.0		Best of Other Methods		AutoShrink 2.0 Latency Rank
	Minimum Latency	Time to Discovery (h)	Minimum Latency	Time to Discovery (h)	
aes	24	1.09	19	23.92	2
atax	524	2.58	524	3.17	1
backprop	3232	0.30	3126	0.45	6
bicg	341	0.43	341	4.42	1
fft	1315	7.71	1060	8.17	3
gemm	17140	5.73	9485	1.33	2
gesummv	299	0.12	299	6.20	1
md_knn	2219	12.91	2219	1.39	1
mvt	267	2.17	267	3.85	1
spam_filter	4104	10.26	4104	15.15	1
spmv	2523	1.47	2523	2.21	1
stencil3d	3351	0.66	3177	0.54	2

directives on QoR and simply perform uniform adjustment on all directives when searching for new design points. This practice hinders fine-grained tuning of crucial directives and ultimately reduces exploration efficacy.

In addition, the aforementioned methods implicitly suffer from timeout-dominated regions in large design spaces. For example, local perturbation strategies adopted by RL and SA often struggle to escape regions where many configurations exceed the synthesis time budget, leading to repeated evaluations of infeasible designs and inefficient use of the exploration budget. Overall, AutoShrink 2.0 incorporates comprehensive pruning, importance-guided sampling, and cost-aware post-filtering to effectively overcome the limitations of existing methods and achieve superior DSE performance.

6.3 Resource-Constrained Latency Optimization

While the preceding experiment evaluates Pareto-front quality, some design scenarios seek to minimize latency subject to hardware resource constraints. We therefore evaluate the minimum latency achievable within the target FPGA capacities of 274,080 LUTs, 2,520 DSPs, and 912 BRAM blocks. The results are extracted from the same multi-objective exploration runs rather than obtained through separate latency-only optimization. For each method and benchmark, we select the best resource-feasible latency obtained across all independent runs and record the earliest time at which that latency is reached. Among the six baseline DSE methods for comparison, the minimum-latency design point is selected and the corresponding discovery time is reported. Table 3 shows the results of this experiment under a 24-hour search budget, along with the latency rank of AutoShrink 2.0 among the seven methods. Methods with identical latency values are assigned the same rank.

The results show that AutoShrink 2.0 ties for the best observed resource-feasible latency on 7 of the 12 benchmarks and ranks within the top three on 11 benchmarks. Among these 7 tied-best latency cases, AutoShrink 2.0 discovers the corresponding design earlier on 6 benchmarks. For the remaining benchmarks, AutoShrink 2.0 does not reach the best observed resource-feasible latency, but its latency values are still close to the best observed values in most cases. The main exception is *gemm*, where the latency gap is relatively large. This gap is expected because these results are extracted from multi-objective exploration runs, where AutoShrink 2.0 optimizes multiple objectives rather than devoting the entire search budget to latency minimization alone. Overall, although AutoShrink 2.0 primarily targets multi-objective DSE, it still achieves competitive resource-feasible latency when viewed from this single-objective perspective.

Manuscript submitted to ACM

Table 4. Ablation study of AutoShrink 2.0 in Vitis HLS. (ADRS %, arithmetic mean $\pm$ std over independent runs).

Benchmark	Variants for Ablation				AutoShrink 2.0 (BO+CP+IGS*+CAF)
	BO	BO+CP	AutoShrink [37] (BO+CP+IGS)	BO+CP+IGS*	
Benchmarks (no timeouts)					
aes	46.95±10.70	28.20±6.05	24.29±5.45	22.67±9.98	reduced to BO+CP+IGS* (CAF is not triggered)
atax	22.90±8.17	9.27±2.09	7.96±2.16	3.24±2.27	
backprop	21.56±7.03	8.08±1.89	7.88±2.31	7.88±1.75	
bicg	51.86±23.43	18.81±2.05	13.87±3.61	9.89±5.68	
fft	39.72±11.84	72.01±21.54	21.27±3.81	18.93±3.07	
gesummv	14.89±4.89	3.78±1.72	3.41±1.94	0.79±0.29	
spam_filter	31.58±17.99	15.16±3.37	3.64±1.02	1.89±1.05	
spmv	1.61±0.74	2.11±0.43	0.29±0.25	0.00±0.00	
Arith. Mean	28.88	19.68	10.33	8.16	
Benchmarks (frequent timeouts)					
gemm	86.76±24.51	66.05±15.68	43.33±14.15	13.78±5.64	11.32±6.49
md_knn	1.46±0.80	1.06±0.34	0.28±0.01	0.21±0.05	0.27±0.11
mvt	38.10±3.02	38.53±5.63	25.56±8.26	15.70±3.33	11.96±3.47
stencil3d	49.02±18.41	33.27±2.56	24.77±12.72	17.97±8.45	11.99±3.28
Arith. Mean	43.84	34.73	23.49	11.92	8.89
Arith. Mean (overall)	33.87	24.69	14.71	9.41	8.40

Note: IGS* denotes the enhanced version of the IGS in our preliminary work [37], augmented with a warm-up stage.

6.4 Ablation Study

To understand the contribution of each component in AutoShrink 2.0, we conduct an ablation study. This study progressively incorporates comprehensive pruning (CP), importance-guided sampling (IGS) along with its enhanced version (IGS*), and cost-aware post-filtering (CAF) on top of the base Bayesian optimization (BO) framework. Note that IGS* and CAF are newly introduced in this paper, whereas CP and IGS are described by our preliminary work [37].

The results in Table 4 demonstrate that the average ADRS gradually decreases as each new component is integrated, indicating continuous DSE performance improvements. Taking the BO baseline (with an overall arithmetic mean ADRS of 33.87%) as a reference, incorporating CP improves the overall DSE performance by approximately 1.37 $\times$. CP eliminates redundant and inefficient regions spanning 2–6 orders of magnitude (Table 2), thereby producing a higher-quality design space and allowing the exploration process to focus on more promising configurations. Although CP improves the overall exploration performance, its benefit is not uniform across all benchmarks. In particular, the degradation on fft may be related to its repeated data transposition and varying array-access strides. Further incorporating IGS forms our preliminary work, AutoShrink [37], and improves the overall performance to 2.30 $\times$. This result indicates that importance-guided sampling enables more effective adjustment of directives that have a larger impact on QoR, thereby improving the efficiency of design space exploration. Compared with IGS, the improved version IGS* additionally introduces a warm-up stage throughout the sampling process. With this improvement, the overall performance further increases to 3.60 $\times$, suggesting that maintaining stronger exploration in the early stage helps identify higher-quality initial Pareto solutions and benefits subsequent exploration. Finally, enabling CAF in the complete framework leads to the best result, achieving an overall improvement of 4.03 $\times$. CAF is mainly effective on timeout-prone benchmarks. For

Manuscript submitted to ACM

Table 5. Impact of $cr_{\min}^*$ on ADRS on the fft benchmark (no warm-up stage).

$cr_{\min}^*$ (%)	ADRS (%) (Mean $\pm$ Std. over 3 runs)
0	36.62 $\pm$ 2.68
10	36.23$\pm$5.37
20	44.92 $\pm$ 11.09
30	58.28 $\pm$ 17.85
40	56.69 $\pm$ 9.27
50	58.65 $\pm$ 20.15
60	82.75 $\pm$ 34.71
70	63.45 $\pm$ 10.66
80	76.49 $\pm$ 12.27
90	79.92 $\pm$ 11.27
100	75.43 $\pm$ 1.77

Table 6. Impact of R_{warmup} on ADRS on the fft benchmark ($cr_{\min}^* = 10\%$).

R_{warmup} (%)	ADRS (%) (Mean $\pm$ Std. over 3 runs)
0	35.56 $\pm$ 5.23
10	35.34 $\pm$ 6.02
20	35.05 $\pm$ 12.18
30	39.72 $\pm$ 8.57
40	29.81$\pm$11.24
50	31.8 $\pm$ 7.61
60	43.55 $\pm$ 1.55
70	34.37 $\pm$ 9.18
80	39.57 $\pm$ 13.35
90	44.54 $\pm$ 12.37
100	53.75 $\pm$ 6.1

benchmarks with frequent timeouts, introducing CAF further reduces the arithmetic mean ADRS from 11.92% to 8.89%, whereas for benchmarks without timeouts, CAF is not triggered and the framework reduces to BO+CP+IGS*. This result confirms that CAF effectively filters configurations that are likely to cause synthesis timeouts, preventing the exploration budget from being repeatedly wasted on high-cost design points and thereby further improving exploration efficiency.

Compared with the preliminary work [37], which integrates BO, CP and IGS, AutoShrink 2.0 further introduces the warm-up mechanism through IGS* and incorporates CAF, reducing the arithmetic mean ADRS from 14.71% to 8.40% and yielding a 1.75 $\times$ improvement. In summary, CP, IGS*, and CAF complement each other by improving exploration efficiency across the pre-exploration, exploration, and post-exploration stages, achieving the best overall performance when combined.

6.5 Trade-off between Exploration and Exploitation

As for importance-guided sampling, IGS* adopts two parameters, the minimum change rate ($cr_{\min}^*$) and the warm-up rate (R_{warmup}), to control the trade-off between *exploring unseen designs* and *exploiting the known Pareto frontier*. Larger $cr_{\min}^*$ and R_{warmup} reduce the degree of exploitation and increase the emphasis on exploration. We investigate how $cr_{\min}^*$ and R_{warmup} affect the effectiveness of sampling.

We demonstrate the results of tuning $cr_{\min}^*$ for IGS* alone, using fft, which has the largest design space, as a representative case, as shown in Table 5. Similar trends are observed across the other benchmarks. The lowest mean ADRS is observed when $cr_{\min}^* = 10\%$. This implies that, in addition to concentrating the search near the Pareto frontier, adding moderate random variations helps escape local optima by probing unevaluated regions, thereby striking a desirable balance between exploration and exploitation. Under this setting, decisions associated with important directives are preserved with a probability of 90%, providing dominant control relative to randomness.

Next, fixing $cr_{\min}^* = 10\%$, we vary the warm-up rate R_{warmup} . Results in Table 6 show that the best performance is achieved when $R_{\text{warmup}} = 40\%$. A shorter warm-up stage may lead to over-exploitation and thus insufficient exploration, whereas an excessively long warm-up delays the timely transition to fine-grained search. With $R_{\text{warmup}} = 40\%$, the first 40% of the search time gradually shifts from exploration to exploitation, while the remaining 60% mainly focuses on

Manuscript submitted to ACM

AutoShrink 2.0: Automating Large-Scale Pareto Optimization in HLS via Adaptive and Cost-Aware Search Space Shrinkage

19

Table 7. Impact of different operation-count threshold choices on ADRS. Results are reported as mean $\pm$ standard deviation over three runs.

Percentile	gemm	mvt	stencil3d	md_knn	Average ADRS (%)
0th	15.130 $\pm$ 8.41	12.031 $\pm$ 4.98	12.446 $\pm$ 4.81	0.327 $\pm$ 0.21	9.984
25th	18.689 $\pm$ 7.08	20.335 $\pm$ 13.82	15.265 $\pm$ 6.87	0.572 $\pm$ 0.49	13.715
50th	19.394 $\pm$ 13.49	15.951 $\pm$ 1.87	14.789 $\pm$ 1.79	0.257 $\pm$ 0.10	12.598
75th	16.065 $\pm$ 6.65	18.856 $\pm$ 13.85	13.205 $\pm$ 8.15	0.761 $\pm$ 0.32	12.222

exploitation, indicating that exploitation plays the dominant role once promising regions of the design space have been identified. Although nearby settings yield similar performance within the observed variability, we adopt $cr_{\min}^* = 10\%$ and $R_{\text{warmup}} = 40\%$ as representative choices rather than uniquely optimal ones, and use these values as the default settings for IGS*. Overall, the two parameters, $cr_{\min}^*$ and R_{warmup} , jointly balance exploration and exploitation across both directive-level sampling and the complete search schedule.

6.6 Analysis of Cost-Aware Filtering

Note that CAF involves two key parameters: the operation-count threshold Th_o used to identify runtime-risky candidates and the update step size Δp of the admission probability P_{eval} . The former determines the boundary of the high-runtime-risk region, whereas the latter controls the filtering strength for candidates exceeding this boundary. In this experiment, we evaluate the effect of CAF. We first examine the choices of Th_o , then determine an appropriate Δp , and finally evaluate the overall impact of CAF on HLS evaluation efficiency. To isolate CAF, the other two modules, CP and IGS*, are disabled in the parameter studies. For each parameter study, the reference Pareto frontier is independently constructed from all designs evaluated within that study.

Impact of operation-count threshold Th_o . We compare four choices of the operation-count threshold Th_o : the 0th, 25th, 50th, and 75th percentiles of the operation counts collected from timeout evaluations. A higher quantile admits more timeout-prone candidates, whereas a lower quantile imposes a more conservative runtime-risk boundary. Specifically, the 0th percentile corresponds to the minimum and the 50th percentile corresponds to the median.

The experiment is conducted on the timeout-prone benchmarks, namely *gemm*, *mvt*, *md_knn*, and *stencil3d*, with three independent runs for each setting. As shown in Table 7, the threshold at the 0th percentile achieves the lowest ADRS on three of the four benchmarks and the lowest overall ADRS of 9.984%. Since over-threshold candidates can still pass the soft filtering stage, a conservative threshold does not completely eliminate the possibility of exploring timeout-prone design points. Based on the evaluation results, we set the minimum timeout operation count (i.e., the 0th percentile) as Th_o in CAF.

Impact of the update step size Δp . We next investigate how the update step size Δp affects CAF. A smaller Δp adjusts the admission probability P_{eval} more gradually, resulting in greater tolerance toward over-threshold candidates, whereas a larger Δp leads to faster convergence of hard filtering. We evaluate five settings: $\Delta p = 1, 1/2, 1/3, 1/4$, and $1/5$. Since *gemm* has a large design space and frequently produces timeout-prone configurations, it is selected as a representative benchmark.

As shown in Table 8, disabling CAF results in only 9 successful evaluations within the fixed 24-hour budget and yields the worst ADRS (81.72%), indicating that a large portion of the time budget is wasted on timeout-prone configurations. Once CAF is enabled, the number of successful evaluations increases substantially under all Δp settings, reaching 19.3×

Manuscript submitted to ACM

Table 8. Impact of CAF and Δp on gemm.

Δp	Avg. Successful Evaluations / 24 h	ADRS (%) (Mean $\pm$ Std. over 3 runs)
CAF Disabled	9.0	81.72 $\pm$ 42.53
1	173.3	50.17 $\pm$ 21.75
1/2	461.7	28.46 $\pm$ 11.08
1/3	282.0	32.98 $\pm$ 9.25
1/4	552.0	46.81 $\pm$ 38.61
1/5	312.3	56.29 $\pm$ 47.85

Table 9. HLS evaluation efficiency comparison on four timeout-prone benchmarks under a 24-hour budget.

Metric	GA	RL	SA	Sherlock	ScaleHLS	Prospector	AutoShrink 2.0 w/o CAF	AutoShrink 2.0
Successful Evaluations	128.2	256.8	226.1	41.5	248.8	206.2	448.3	454.3
Timeout Evaluations	15.7	20.8	18.7	37.7	18.1	23.9	15.5	6.1
Timeout Ratio (%)	15.30	14.01	15.95	56.81	14.16	29.02	4.37	1.48

Note: With CAF enabled, AutoShrink 2.0 filters an average of 56.5 runtime-risky candidates per benchmark before HLS evaluation.

to 61.3 $\times$ that of the CAF-disabled case, while the ADRS is also significantly reduced. This result indicates that CAF effectively filters out configurations with excessive evaluation cost, allowing the framework to complete more valid HLS evaluations within the same time budget.

The ADRS values under different Δp settings generally reflect a trade-off between filtering aggressiveness and exploration flexibility. When Δp is too small, P_{eval} changes too gradually, allowing more high-cost candidates to be evaluated and reducing exploration efficiency. In contrast, an excessively large Δp imposes overly strict filtering and may prematurely discard promising configurations. The lowest ADRS of 28.46% is observed at $\Delta p = 1/2$, indicating that a moderate update step achieves a balanced trade-off between runtime cost control and exploration flexibility.

Overall evaluation efficiency. After fixing Th_o and Δp , we evaluate the overall efficiency of CAF. Specifically, we collect statistics on successful and timeout evaluations for AutoShrink 2.0, its variant without CAF, and the six baselines on four timeout-prone benchmarks, namely gemm, mvt, md_knn, and stencil3d. We adopt the *timeout ratio* to quantify the frequency of timeout occurrences, which is formally defined as the number of timeout evaluations divided by the total number of evaluations, including both successful and timeout evaluations.

Experimental results are shown in Table 9. AutoShrink 2.0 achieves the highest average number of successful evaluations and the lowest timeout ratio of 1.48% among all evaluated methods. This efficiency gain arises from two complementary sources. First, even without CAF, the AutoShrink 2.0 variant achieves the second-highest number of successful evaluations and the second-lowest timeout ratio, suggesting that the importance-guided sampling strategy itself can make effective sampling decisions and reduce the likelihood of allocating the search budget to timeout-prone configurations. Second, enabling CAF further filters an average of 56.5 runtime-risky candidates per benchmark, reduces the average number of timeout evaluations from 15.5 to 6.1, and decreases the timeout ratio from 4.37% to 1.48%, corresponding to a 66.1% reduction. Together with the ADRS results reported in Tables 2 and 4, these results indicate that CAF improves search-budget utilization by reducing unnecessary timeout evaluations.

Manuscript submitted to ACM

AutoShrink 2.0: Automating Large-Scale Pareto Optimization in HLS via Adaptive and Cost-Aware Search Space Shrinkage

21

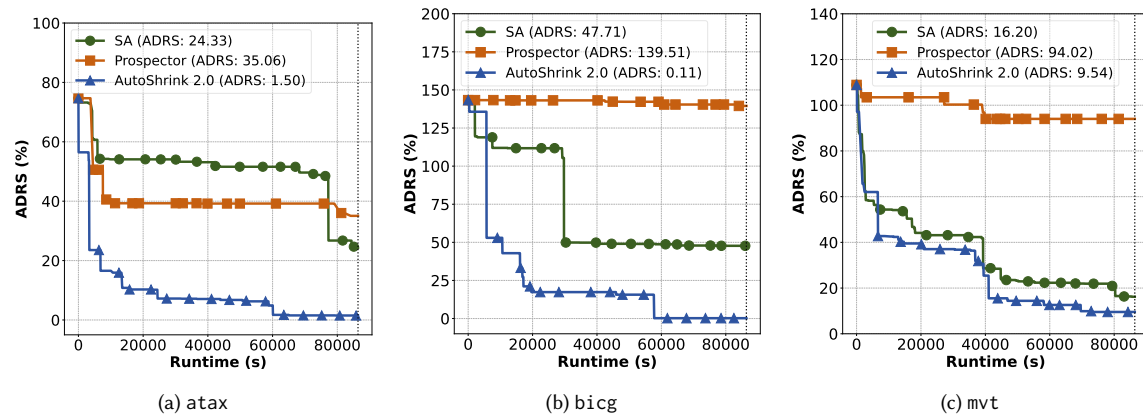


Fig. 6. ADRS curves of SA, Prospector, and AutoShrink 2.0 over runtime from the first independent run of each Catapult HLS benchmark.

6.7 black

Cross-Tool Evaluation

Table 10. Cross-tool validation in Catapult HLS. The ADRS values are reported as arithmetic mean $\pm$ standard deviation over three independent runs.

Benchmark	SA	Prospector	AutoShrink 2.0 (ours)
atax	10.88 $\pm$ 11.66	24.77 $\pm$ 14.62	1.28$\pm$0.20
bicg	30.78 $\pm$ 15.03	74.95 $\pm$ 55.93	0.33$\pm$0.26
mvt	10.93 $\pm$ 4.57	73.71 $\pm$ 17.75	8.99$\pm$1.22
Arith. Mean	17.53	57.81	3.53

To further evaluate the applicability of our method to a different HLS tool, we apply AutoShrink 2.0 to Catapult HLS [7], a mainstream HLS tool supporting both FPGA- and ASIC-oriented design flows. Compared with Vitis HLS, Catapult HLS provides a different set of available directives and QoR metrics. Specifically, the optimization space should be reconstructed using Catapult-specific directive configurations, including loop unrolling, pipelining, and array partitioning implemented through Catapult’s INTERLEAVE directive. The optimization objectives are defined by latency and area score according to the available QoR metrics.

When transferring AutoShrink 2.0 to the Catapult HLS flow, the optimization framework remains unchanged. CP applies Rule 3.1 in Table 1 to the reconstructed Catapult design space, while IGS* and CAF are both enabled and require no structural modification. IGS* updates directive importance using Catapult-specific QoR feedback, and CAF retains the same operation-count proxy and timeout-based filtering mechanism. Based on prior experimental results, we select SA and Prospector as representative SOTA DSE methods for comparison, since they achieve the best performance among the baselines. All evaluated methods follow the same experimental settings, including the time budget, evaluation timeout, and number of independent runs. We adopt the benchmarks atax, bicg, and mvt for evaluation, as the other benchmarks are not directly compatible with Catapult HLS.

Manuscript submitted to ACM

According to the results shown in Table 10, AutoShrink 2.0 achieves the best ADRS with small variations on all three benchmarks. Concretely, it achieves an arithmetic mean ADRS of 3.53%, outperforming SA and Prospector by 4.97× and 16.38×, respectively. Fig. 6 presents the ADRS evolution over time for an independent run of each benchmark. In these runs, AutoShrink 2.0 reduces ADRS more rapidly, continues improving after the baselines begin to plateau, and finally reaches the lowest ADRS on all three benchmarks. Overall, these results demonstrate that AutoShrink 2.0 maintains high solution quality and shows promising cross-tool applicability when transferred to a different HLS tool with distinct directive configuration spaces and QoR metrics.

7 Related Work

Automating design optimization in HLS has attracted substantial research interest. Recent studies on HLS DSE mainly focus on design space pruning strategies and exploration algorithms.

7.1 Design Space Pruning

Design space pruning strategies aim to reduce the design space before search, making runtime exploration more tractable. Kuang *et al.* [17] propose a tree-based pruning approach that considers the interactions among loop flattening, unrolling, and pipelining across different loop hierarchies. Liao *et al.* [21] introduce a subspace pruning method that eliminates design points that do not satisfy frequency requirements. Choi *et al.* [4] remove design points with multi-dimensional array partitioning to simplify routing in the back-end process. Existing methods mainly prune based on a single design constraint or a small subset of directives, which is not comprehensive enough to support large-scale DSE with various types of directives.

7.2 DSE Algorithm

Existing HLS DSE algorithms can be broadly divided into pre-trained predictor-based methods and online iterative methods. Pre-trained approaches aim to improve early-stage efficiency by replacing costly online calibration with offline-learned surrogates that can rapidly estimate the QoR of candidate designs. For example, Zhong *et al.* [38] proposed MPSeeker, which combines LLVM-based profiling with analytical and GBM-based estimation models to rapidly predict performance and resource usage for FPGA accelerators under multi-level parallelism, thereby identifying near-optimal pragma configurations without exhaustive HLS evaluation. Sohrabizadeh *et al.* [31] proposed GNN-DSE, which represents programs as graphs augmented with pragma flows and trains a cross-application GNN surrogate to evaluate design points in milliseconds, enabling Pareto search with explicit transfer capability to unseen kernels. Wu *et al.* [35] further proposed IRONMAN-PRO, an end-to-end learning framework that combines a GNN-based post-implementation predictor with an RL-based exploration engine and code transformation, aiming to provide flexible multi-objective DSE under user-specified constraints. Although these methods can significantly reduce the cold-start overhead of exploration, they typically rely on substantial offline training data, and their effectiveness may still degrade when transferred to unseen applications or design distributions. In contrast, online DSE methods iteratively use evaluation feedback from the target application to guide subsequent candidate selection, making them naturally suited to application-specific exploration. Motivated by this difference, this work adopts online DSE as its basic exploration paradigm rather than relying on pre-trained predictor-based DSE.

Online iterative HLS DSE methods can be roughly grouped into three categories. The first category includes generic black-box optimization methods. Ferrandi *et al.* [10] formulate HLS DSE as a multi-objective evolutionary optimization problem and employ an NSGA-II-based genetic algorithm to search for area-performance trade-offs.

Manuscript submitted to ACM

AutoShrink 2.0: Automating Large-Scale Pareto Optimization in HLS via Adaptive and Cost-Aware Search Space Shrinkage

23

Schafer et al. [28] propose an adaptive simulated annealing framework that explores pragma configurations through stochastic perturbation and adaptive cost balancing. Ferretti et al. [11] develop a lattice-traversing exploration strategy that searches local neighborhoods of promising configurations based on the observation that Pareto-optimal designs tend to cluster. These methods treat HLS DSE as a general optimization problem and improve solutions through generic search operators, stochastic perturbation, or local neighborhood traversal.

The second category includes online surrogate-guided optimization methods. Mehrabi et al. [24] propose Prospector, which applies Bayesian optimization to HLS directive tuning by iteratively fitting a probabilistic surrogate model and selecting promising candidates through acquisition-guided sampling. Gautier et al. [12] propose Sherlock, an active-learning-based multi-objective DSE framework that incrementally refines surrogate models around the estimated Pareto frontier and adaptively selects regressors using a multi-armed bandit strategy. Zou et al. [40] propose FlexWalker, which combines heterogeneous regressors, UCB-based model selection, and probabilistic sampling to improve robustness in multi-objective HLS DSE. These methods improve search efficiency by constructing or refining surrogate models from newly observed QoR results and using them to guide subsequent candidate selection.

The third category includes domain-specific customized search methods. Sohrabizadeh et al. [32] propose AutoDSE, a bottleneck-guided coordinate optimization framework that leverages compiler feedback and HLS reports to prioritize high-impact pragma parameters during iterative exploration. Ye et al. [36] propose ScaleHLS, a multi-level MLIR-based framework that exposes graph-, loop-, and directive-level transformations as tunable optimization interfaces for scalable HLS exploration. These methods exploit HLS-specific representations, compiler feedback, program structure, or customized search mechanisms to design more targeted exploration strategies.

However, despite their differences, existing online iterative methods mainly focus on search algorithms, surrogate construction, or handcrafted exploration heuristics, while paying limited attention to how candidate perturbation itself can be systematically guided to improve sampling efficiency in extremely large design spaces.

7.3 Handling Expensive HLS Evaluations

Existing HLS DSE studies mainly improve efficiency by reducing the number of costly evaluations through surrogate modeling, guided exploration, or pruning, whereas the explicit handling of design points with excessive HLS evaluation cost remains relatively underexplored [12, 24]. A few works treat overly expensive configurations using simple infeasibility rules [8, 32], but online and adaptive cost-aware control is still largely unexplored. In particular, most existing approaches focus primarily on reducing the number of sampled design points, while the runtime cost of individual HLS evaluations is less explicitly incorporated into the exploration loop. This distinction is important because fewer evaluations do not necessarily translate into shorter end-to-end exploration time when synthesis cost varies substantially across configurations. Consequently, a nontrivial fraction of the wall-clock budget may still be consumed by a small number of disproportionately expensive design points. This limitation motivates our cost-aware post-filtering strategy, which learns from observed timeout feedback online and dynamically controls the admission of high-cost candidates, thereby improving exploration efficiency under a fixed time budget.

8 Conclusion

This paper proposes AutoShrink 2.0, an HLS DSE framework tailored for large-scale design spaces. The key idea behind AutoShrink 2.0 is to shrink the overall difficulty of exploration from three complementary aspects: shrinking the design space itself, shrinking the effective search scope, and shrinking the evaluation cost. To this end, AutoShrink 2.0 integrates three key components: a comprehensive pruning strategy to remove redundant and suboptimal design points

Manuscript submitted to ACM

before exploration, an importance-guided exploration algorithm to focus the search on more influential directives during exploration, and a cost-aware post-filtering strategy to suppress runtime-expensive candidates and reduce unnecessary evaluation cost. Experimental results demonstrate that AutoShrink 2.0 consistently achieves the best ADRS in the multi-objective DSE comparison and finds competitive resource-feasible low-latency designs in the resource-constrained latency comparison. The evaluation-efficiency analysis further shows that cost-aware post-filtering reduces wasted evaluations on timeout-prone benchmarks. Ablation and parameter studies validate the complementary benefits of the three components, while Catapult HLS experiments indicate the promising cross-tool applicability of AutoShrink 2.0.

References

- [1] AMD/Xilinx Ltd. 2022. *Vitis High-Level Synthesis User Guide*. White Paper, UG1399 (v2022.2).
- [2] Syrine Belakaria, Aryan Deshwal, and Janardhan Rao Doppa. 2019. Max-value entropy search for multi-objective Bayesian optimization. *Advances in neural information processing systems* 32 (2019).
- [3] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *The 19th ACM/SIGDA international symposium on Field programmable gate arrays (FPGA)*. 33–36.
- [4] Young-kyu Choi and Jason Cong. 2018. HLS-based optimization and design space exploration for applications with variable loop bounds. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8.
- [5] Philippe Coussy and Adam Morawiec. 2008. *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer.
- [6] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6, 2 (2002), 182–197. doi:10.1109/4235.996017
- [7] Siemens EDA. 2024. *Catapult High-Level Synthesis and Verification*. Technical Report. <https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/>
- [8] Aggelos Ferikoglou, Andreas Kakolyris, Vasilis Kypriotis, Dimosthenis Masouros, Dimitrios Soudris, and Sotirios Xydis. 2024. Data-driven HLS optimization for reconfigurable accelerators. In *ACM/IEEE Design Automation Conference (DAC)*. 1–6.
- [9] Fabrizio Ferrandi, Vito Giovanni Castellana, Serena Curzel, Pietro Fezzardi, Michele Fiorito, Marco Lattuada, Marco Minutoli, Christian Pilato, and Antonino Tumeo. 2021. Invited: Bambu: an Open-Source Research Framework for the High-Level Synthesis of Complex Applications. In *ACM/IEEE Design Automation Conference (DAC)*. 1327–1330. doi:10.1109/DAC18074.2021.9586110
- [10] Fabrizio Ferrandi, Pier Luca Lanzi, Daniele Loiacono, Christian Pilato, and Donatella Sciuto. 2008. A multi-objective genetic algorithm for design space exploration in high-level synthesis. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 417–422.
- [11] Lorenzo Ferretti, Giovanni Ansaloni, and Laura Pozzi. 2018. Lattice-traversing design space exploration for high level synthesis. In *International Conference on Computer Design (ICCD)*. 210–217.
- [12] Quentin Gautier, Alric Althoff, Christopher L Crutchfield, and Ryan Kastner. 2022. Sherlock: A multi-objective design space exploration framework. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 27, 4 (2022), 1–20.
- [13] Oded Goldreich. 2010. *P, NP, and NP-Completeness: The basics of computational complexity*. Cambridge University Press.
- [14] Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. 2014. An efficient approach for assessing hyperparameter importance. In *International conference on machine learning (ICML)*. PMLR, 754–762.
- [15] Beakcheol Jang, Myeonghwi Kim, Gaspard Harerimana, and Jong Wook Kim. 2019. Q-Learning Algorithms: A Comprehensive Classification and Applications. *IEEE Access* 7 (2019), 133653–133667. doi:10.1109/ACCESS.2019.2941229
- [16] Arthur B Kahn. 1962. Topological sorting of large networks. *Commun. ACM* 5, 11 (1962), 558–562.
- [17] Huizhen Kuang, Xianfeng Cao, Jingyuan Li, and Lingli Wang. 2023. HGBO-DSE: Hierarchical GNN and Bayesian Optimization based HLS Design Space Exploration. In *International Conference on Field Programmable Technology (ICFPT)*. 106–114. doi:10.1109/ICFPT59805.2023.00017
- [18] Sakari Lahti, Panu Sjövall, Jarno Vanne, and Timo D Hämäläinen. 2018. Are we there yet? A study on the state of high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 38, 5 (2018), 898–911.
- [19] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization (CGO)*. 75–86.
- [20] Yang Li, Yu Shen, Wentao Zhang, Yuanwei Chen, Huaijun Jiang, Mingchao Liu, Jiawei Jiang, Jinyang Gao, Wentao Wu, Zhi Yang, et al. 2021. Openbox: A generalized black-box optimization service. In *The 27th ACM SIGKDD conference on knowledge discovery & data mining*. 3209–3219.
- [21] Yuchao Liao, Tosiron Adegbija, and Roman Lysecky. 2023. Efficient system-level design space exploration for high-level synthesis using pareto-optimal subspace pruning. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*. 567–572.
- [22] Ilya Loshchilov and Frank Hutter. 2017. SGDR: Stochastic Gradient Descent with Warm Restarts. In *International Conference on Learning Representations (ICLR)*. <https://openreview.net/forum?id=Skq89Scxx>
- [23] Chittaranjan A Mandal, Partha Pratim Chakrabarti, and Sujoy Ghose. 1998. Complexity of scheduling in high level synthesis. *VLSI DESiGN* 7, 4 (1998), 337–346.

Manuscript submitted to ACM

AutoShrink 2.0: Automating Large-Scale Pareto Optimization in HLS via Adaptive and Cost-Aware Search Space Shrinkage

25

- [24] Atefeh Mehrabi, Aninda Manocha, Benjamin C Lee, and Daniel J Sorin. 2020. Prospector: Synthesizing efficient accelerators via statistical learning. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 151–156.
- [25] Pierre G Paulin and John P Knight. 1989. Scheduling and binding algorithms for high-level synthesis. In *ACM/IEEE Design Automation Conference (DAC)*. 1–6.
- [26] L.-N. Pouchet. 2012. Polybench: The Polyhedral Benchmark Suite. <http://www.cs.ucla.edu/~pouchet/software/polybench>.
- [27] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. 2014. Machsuite: Benchmarks for accelerator design and customized architectures. In *IEEE International Symposium on Workload Characterization (IISWC)*. 110–119.
- [28] Benjamin Carrion Schafer, Takashi Takenaka, and Kazutoshi Wakabayashi. 2009. Adaptive simulated annealer for high level synthesis design space exploration. In *International Symposium on VLSI Design, Automation and Test*. 106–109.
- [29] Benjamin Carrion Schafer and Zi Wang. 2020. High-Level Synthesis Design Space Exploration: Past, Present, and Future. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 39, 10 (2020), 2628–2639.
- [30] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. 2015. Taking the human out of the loop: A review of Bayesian optimization. *Proc. IEEE* 104, 1 (2015), 148–175.
- [31] Atefeh Sohrabizadeh, Yunsheng Bai, Yizhou Sun, and Jason Cong. 2022. Automated accelerator optimization aided by graph neural networks. In *ACM/IEEE Design Automation Conference (DAC)*. 55–60.
- [32] Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong. 2022. AutoDSE: Enabling software programmers to design efficient FPGA accelerators. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 27, 4 (2022), 1–27.
- [33] Qi Sun, Tinghuan Chen, Siting Liu, Jianli Chen, Hao Yu, and Bei Yu. 2022. Correlated multi-objective multi-fidelity optimization for HLS directives design. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 27, 4 (2022), 1–27.
- [34] Peter JM Van Laarhoven and Emile HL Aarts. 1987. *Simulated annealing*. Springer.
- [35] Nan Wu, Yuan Xie, and Cong Hao. 2022. IRONMAN-PRO: Multiobjective design space exploration in HLS via reinforcement learning and graph neural network-based modeling. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 42, 3 (2022), 900–913.
- [36] Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. 2022. Scalehls: A new scalable high-level synthesis framework on multi-level intermediate representation. In *IEEE international symposium on high-performance computer architecture (HPCA)*. 741–755.
- [37] Yingxin Zeng, Binghao Cheng, Jianwang Zhai, Kang Zhao, and Zhe Lin. 2026. AutoShrink: Adaptive Search Space Shrinkage for Large-Scale Pareto Optimization of HLS Designs. In *Design, Automation & Test in Europe Conference (DATE)*.
- [38] Guanwen Zhong, Alok Prakash, Siqi Wang, Yun Liang, Tulika Mitra, and Smail Niar. 2017. Design space exploration of FPGA-based accelerators with multi-level parallelism. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1141–1146.
- [39] Yuan Zhou, Udit Gupta, Steve Dai, Ritchie Zhao, Nitish Srivastava, Hanchen Jin, Joseph Featherston, Yi-Hsiang Lai, Gai Liu, Gustavo Angarita Velasquez, Wenping Wang, and Zhiru Zhang. 2018. Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software Programmable FPGAs. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. 269–278.
- [40] Zheyuan Zou, Cheng Tang, Lei Gong, Chao Wang, and Xuehai Zhou. 2024. FlexWalker: An Efficient Multi-Objective Design Space Exploration Framework for HLS Design. In *International Conference on Field-Programmable Logic and Applications (FPL)*. 126–132.

Manuscript submitted to ACM