Sherlock: A Multi-Objective Design Space Exploration Framework

QUENTIN GAUTIER, University of California San Diego
ALRIC ALTHOFF, Tortuga Logic
CHRISTOPHER L. CRUTCHFIELD and RYAN KASTNER, University of California San Diego

Design space exploration (DSE) provides intelligent methods to tune the large number of optimization parameters present in modern FPGA high-level synthesis tools. High-level synthesis parameter tuning is a time-consuming process due to lengthy hardware compilation times—synthesizing an FPGA design can take tens of hours. DSE helps find an optimal solution faster than brute-force methods without relying on designer intuition to achieve high-quality results. Sherlock is a DSE framework that can handle multiple conflicting optimization objectives and aggressively focuses on finding Pareto-optimal solutions. Sherlock integrates a model selection process to choose the regression model that helps reach the optimal solution faster. Sherlock designs a strategy based around the multi-armed bandit problem, opting to balance exploration and exploitation based on the learned and expected results. Sherlock can decrease the importance of models that do not provide correct estimates, reaching the optimal design faster. Sherlock is capable of tailoring its choice of regression models to the problem at hand, leading to a model that best reflects the application design space. We have tested the framework on a large dataset of FPGA design problems and found that Sherlock converges toward the set of optimal designs faster than similar frameworks.

CCS Concepts: • **Hardware → Modeling and parameter extraction**; *Design rules*

Additional Key Words and Phrases: Design space exploration, optimization, design automation, FPGA

ACM Reference format:
Quentin Gautier, Alric Althoff, Christopher L. Crutchfield, and Ryan Kastner. 2022. Sherlock: A Multi-Objective Design Space Exploration Framework. *ACM Trans. Des. Autom. Electron. Syst.* 27, 4, Article 33 (March 2022), 20 pages.
https://doi.org/10.1145/3511472

1 INTRODUCTION

Optimizing a **high-level synthesis (HLS)** design involves a lengthy process of refactoring the code and tuning directives. An HLS designer can modify the amount of exploitable parallelism, pipelining, memory structure, and data types to balance the design's throughput, resource utilization, power consumption, and other relevant optimization objectives. Any design change can result in a drastic modification of the underlying hardware architecture, which forces the designer to perform time-consuming synthesis runs to obtain accurate estimates of

Authors' addresses: Q. Gautier, C. L. Crutchfield, and R. Kastner, University of California San Diego, 9500 Gilman Drive, La Jolla, CA 92161; emails: qkgautier@gmail.com, {ccrutchf, kastner}@ucsd.edu; A. Althoff, Tortuga Logic, 75 E. Santa Clara Street, San Jose, CA 95113; email: alric@tortugalogic.com.

1084-4309/2022/03-ART33 $15.00
https://doi.org/10.1145/3511472

throughput, resource usage, and power. One synthesis run can easily take multiple to tens of hours.

HLS tools provide many directives, resulting in a large design space that must be explored intelligently. These directives create a highly complex design space that is often non-linear [7, 17] and generally contains mutually exclusive objectives (e.g., resource usage vs. throughput). Therefore, it is crucial for the designer to find the set of optimal designs along all the objectives. In a context where compiling and/or running each design takes a long time, evaluating all possible designs is infeasible, and thus the designer is forced to selectively sample the design space to find the best results.

Sherlock is a **design space exploration (DSE)** framework that uses active learning to evaluate and intelligently explore the HLS design space. Sherlock can quickly reach the set of Pareto-optimal designs by minimizing the initialization size, and performing a sample selection entirely driven by the estimated optimal set, using a strategy that balances exploration and exploitation based on an underlying design space surrogate model.

Sherlock's active learning process is based on a regression model that iteratively provides design space estimates. We tested Sherlock using multiple types of regression models, spanning from complex models often used in active learning literature to simpler consensus-based interpolation kernels that help to reach an almost optimal solution faster. To make Sherlock more flexible and adaptable, we create a model selection strategy based on the **multi-armed bandit (MAB)** problem that rewards the models directly improving the actual Pareto front. With this strategy, the framework can quickly decrease the importance of models that do not provide correct estimates. It can then leverage all models relative to their positive contribution to reach the optimal designs faster.

Sherlock uses several unique features in its DSE process. First, Sherlock focuses on accurately modeling only the designs on or near the Pareto front. This reduces the problem size complexity, focuses the DSE, and generally allows the exploration to converge quickly. Second, our results show that a one-size-fits-all approach to design space modeling is not effective; HLS design spaces are unique, and thus they cannot all be accurately modeled using a single surrogate model. Sherlock adaptively selects from different surrogate models (Gaussian process, random forest, **radial basis functions (RBFs)**) to determine the one that most accurately reflects the design space under consideration. Finally, Sherlock balances between exploration and exploitation in selecting its samples to evaluate. Exploration aims to maximize the uncertainty of the model while exploitation samples points that are most likely to be Pareto optimal. Our results show that this sampling process plays an important role in DSE.

This article describes the Sherlock DSE framework. The major contributions are as follows:

- We show that the design space model plays an important role in DSE. Sherlock uses a model selection strategy to adapt it to a wide variety of design spaces.
- Sherlock uses an adaptive exploit versus explore sample selection strategy. We show that this strategy is effective in quickly converging to the Pareto front.
- We compare Sherlock to other DSE tools and determine that Sherlock generally outperforms those tools for FPGA HLS DSE.
- We release Sherlock as open source (https://git.io/JKuFz).

Section 2 introduces the problem and notations, describes Sherlock's core active learning algorithm, and presents the method to select regression models. Section 3 contains the results of running Sherlock against multiple FPGA benchmarks that cover a wide variety of applications. We discuss related work in Section 4 and conclude in Section 5.

2 SHERLOCK

Sherlock is a DSE strategy that uses active learning to iteratively improve the known set of optimal designs, and it relies upon a surrogate model to estimate the design space. Sherlock allows for different models and provides a model selection techniques that can automatically select between the available models to best match the underlying design space.

This section starts by defining the DSE problem and introducing formal definitions. Then, it provides details about the Sherlock algorithm. This includes the ability to use different surrogate models to model the underlying design space.

2.1 Scope and Definitions

A *design space*[1] is composed of both an *input space* and an *output space*. The input space is a set of FPGA HLS designs that met the application's functional requirements. The differences in the designs can be described through the definition of different design parameters, also known in the DSE literature as *knobs*. In FPGA HLS DSE, knobs are related to loop unrolling factors, pipeline initiation intervals, the number of work items/work groups, and so forth. The output space is defined by the optimization *objectives* set by the designer of the application. The major objectives for FPGA HLS DSE relate to throughput and resource utilization. Evaluating a design sample to obtain these objectives requires fully synthesizing and implementing HLS designs to a bitstream, integrating them into a larger system, and evaluating them using a dataset representative of the target application [7, 23]. These results form the output space, represented by a matrix y of dimensions $(m \times o)$, where o is the number of objectives. Each row of y is a fully resolved sample of the FPGA HLS design with specified input space knobs.

More formally stated, the input space X is defined as $X = \{k_1 \times k_2 \times \cdots \times k_n\} \in \mathbb{X}^{m\times n}$, where k_i is a knob vector containing all the possible values for this knob, and in this case, $\mathbb{X} = \mathbb{R}$. The resulting matrix has n columns for each knob, and m rows for each unique and valid combination of knob values (i.e., m design candidates). Knobs take n values, for $n \in [2, \infty]$. They can be discrete, categorical, or continuous. Continuous knobs can generally be discretized by knowing the bounds of the knob and choosing a reasonable set of values based on the target platform (powers of 2, regular grid, etc.). Categorical knobs are interpreted as numerical values. The output space $y \in \mathbb{Y}^{m\times o}$ is a matrix of m rows for each design candidate and o columns for each optimization objective. The final design space S is the combination of the input and output space: $S = \{(X_i, y_i)\}$.

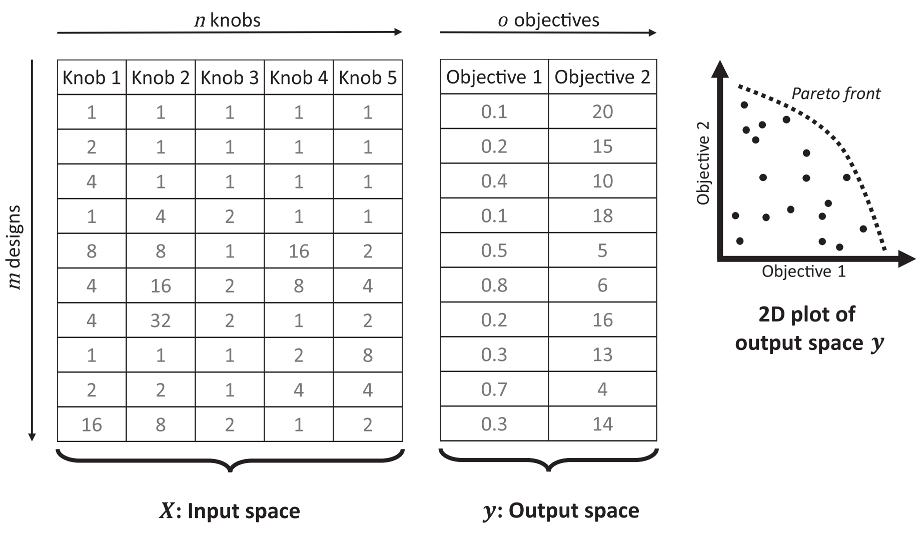
DSE aims to find the set $P \subseteq S$ of design candidates that are optimal on at least one objective. As a multi-objective optimization problem, this set P corresponds to the set of Pareto-optimal designs—that is, designs that cannot be improved on one objective without decreasing another, also known as the Pareto front. Since evaluating a design candidate is a time-consuming process, DSE should determine the designs on the Pareto front while sampling as few points as possible.

Figure 1 provides an example of a design space. The input space consists of five knobs ($n = 5$), and the output spaces has two objectives ($o = 2$). There are 10 designs ($m = 10$), each corresponding to a unique knob vector. The knob settings here are discretized to powers of 2. The graph on the right depicts the output space. In this case, we aim to maximize both the objectives, and thus the Pareto front consists of designs toward the upper right area of the output space.

A key element of DSE is estimating the function $f : \mathbf{R}^m \rightarrow \mathbf{R}^o$ such that

$$y = f(X).$$

[1]A design space is the combination of an input space and an output space, although the designation is often informally used to refer to either the input space or, output space, or both.



Knob 1	Knob 2	Knob 3	Knob 4	Knob 5
1	1	1	1	1
2	1	1	1	1
4	1	1	1	1
1	4	2	1	1
8	8	1	16	2
4	16	2	8	4
4	32	2	1	2
1	1	1	2	8
2	2	1	4	4
16	8	2	1	2

Objective 1	Objective 2
0.1	20
0.2	15
0.4	10
0.1	18
0.5	5
0.8	6
0.2	16
0.3	13
0.7	4
0.3	14

Fig. 1. Representation of a design space S. The example input space X consists of m design candidates, each with $n = 5$ knob values. The output space y is a set of $o = 2$ objectives where larger is better. Graphing the output space shows the Pareto front P, which are the solutions that are optimal w.r.t. at least one objective.

We assume that f can potentially be non-linear and thus would require a large number of samples to be estimated. Furthermore, since the rows of y generally come from measurements (running time, accuracy, etc.), the output space may not be deterministic. For example, multiple runs of an application would lead to different throughput depending on the system state. Thus, a high-quality DSE tool needs to be robust to noise in order to build a stable model of f.

Since the goal of DSE is to find the Pareto front P, it is not strictly necessary for the DSE framework to accurately model the entire design space S; it only needs to understand the design space around the Pareto front. This is especially important in scenarios where sampling the output space is expensive.

DSE outputs an estimated Pareto front $\hat{P}$. To understand the quality of the estimated Pareto front, a metric is needed to compare the estimated Pareto designs with the actual Pareto front. **Average Distance to Reference Set (ADRS)** [19] measures the average normalized distance between the estimated Pareto front $\hat{P}$ and the actual Pareto front P (i.e., the reference). ADRS computes the distance from every estimated Pareto design to the closest point on the actual Pareto front. An ADRS equal to 0 indicates that every estimated Pareto point is on the actual Pareto front. An increasing ADRS indicates that the estimated Pareto front is moving away from the actual Pareto front. Thus, ADRS is commonly used to compare estimated Pareto fronts where smaller is better.

2.2 Base Algorithm

Sherlock uses active learning [27] to find the Pareto-optimal designs. Active learning is a subfield of machine learning that aims to intelligently sample and better model a problem with limited labels. Active learning techniques target scenarios where one can pose only a limited number of queries. Thus, they must carefully consider which designs to sample and learn from the results. For HLS DSE, the queries are the HLS candidate designs and the resulting labels are objective metrics of the final implementation of that HLS design (e.g., resource usage and throughput). A query of the design space involves time-consuming synthesis compilations, implementation on an FPGA system, and evaluation of the system using real data. This can easily take hours and often

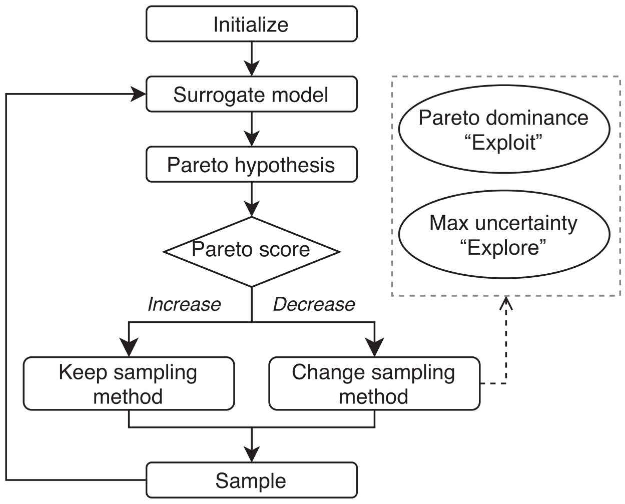


Fig. 2. Sherlock uses an active learning approach that iteratively selects designs to sample while balancing exploitation and exploration of the surrogate model.

ALGORITHM 1: Sherlock Static Model Algorithm

```
Input  : $X$
$s_1, s_2 = TED(X)$
$K = \{s_1, s_2\}$
mode ← "explore"
prevscore = $-\infty$
while $|K| <$ sample budget do
    $\hat{f} \leftarrow g(.;K)$
    $\hat{y}, H \leftarrow \hat{f}(X)$
    $\hat{P}, H_{\hat{P}} \leftarrow$ pareto($\hat{y}, H$)
    curscore ← $\max_{s \in \hat{P}}$(scores($\hat{y}$))
    if curscore < prevscore then
        mode ← next(mode)
    end
    if mode = "explore" then
        $i \leftarrow \mathrm{argmax}(H_{\hat{P}})$
    else if mode = "exploit" then
        $i \leftarrow \mathrm{argmax}_{s_i \in S \setminus K}$(scores($\hat{y}$))
    end
    $K \leftarrow K \cup \{s_i\}$
    prevscore ← curscore
end
```

overnight or even longer for the largest designs. Thus, the number of queries should be minimized since it involves a series of time-intensive operations.

Sherlock works iteratively by (1) creating a surrogate model of the design space to formulate a hypothesis of the Pareto-optimal designs, (2) selecting the next candidate to sample, (3) sampling the chosen design to obtain objective values, and (4) refining the model based on the new sample. The loop continues until a stopping criterion is met. The specifics of Sherlock's active learning workflow are summarized in Figure 2, and pseudocode is presented in Algorithm 1.

The goal of Sherlock is to *exploit* a surrogate design space model to find the Pareto-optimal designs as quickly as possible. In other words, Sherlock aims to improve the Pareto front of known designs at each iteration. However, if the surrogate model is not accurate, Sherlock should opt to *explore* the design space by sampling the design with the most uncertainty with the hopes that it will lead to a better surrogate model. Furthermore, Sherlock aims to learn the best surrogate design space model for the given design space. Finally, Sherlock focuses its exploration on designs at or around the estimated Pareto front in an attempt to keep the sampling process in an optimal space.

2.2.1 Initialization. As a starting point of the algorithm, we choose an initial number of samples to evaluate and collect their objective values. We build a set K of known designs (for which y_i is known) that will grow with future iterations.

$$K = \{(X_1, y_1), (X_2, y_2), \ldots\} \tag{1}$$

Before the initial sampling, we do not possess any information on the output space, and therefore the sample selection must occur based on the input space only. Sherlock uses the **transductive experimental design (TED)** [34] algorithm to provide a representative set of samples. TED intelligently selects the initial sampling space K such that the sampled designs are hard to predict while remaining representative of the entire input space. TED is only used for the initial sampling. After that, Sherlock decides on the next samples to select based on an exploit/explore strategy that is detailed in the following. Our experiments set $K = 5$.

2.2.2 Formulation of the Pareto Hypothesis. The goal of Sherlock is to focus the learning method on the Pareto front of the design space. The intuition is that it is important to accurately assess the Pareto designs. Likewise, it is less important to accurately model non-Pareto points, especially those that are far from optimal.

Sherlock estimates the Pareto front through the use of a surrogate model. A surrogate model is defined as a function $\hat{f}$:

$$\hat{f} \leftarrow g(K), \tag{2}$$

$$g : S \rightarrow (\mathbb{X} \rightarrow \mathbb{Y}), \tag{3}$$

where g is a supervised learning method for regression. We use the surrogate model to provide an estimate of the entire output space:

$$\hat{y}, H \leftarrow \hat{f}(X), \tag{4}$$

where $\hat{y}$ is the estimated output space and H is the uncertainty of each estimation. Sherlock then extracts the estimated Pareto designs using that surrogate model. More precisely, Sherlock calculates the estimated Pareto front $\hat{P}$ from the estimated output space $\hat{y}$ along with a measure of the uncertainty of the Pareto front estimate $H_{\hat{P}}$:

$$\hat{P}, H_{\hat{P}} \leftarrow \text{pareto}(\hat{y}, H), \tag{5}$$

where *pareto* is a function to extract the set of Pareto-optimal designs.

2.2.3 Sample Selection. Sampling is the process of choosing one design s_i to evaluate and obtain its output value. The result is an increased set of known designs:

$$K \leftarrow K \cup \{s_i\}. \tag{6}$$

Sherlock must decide at every iteration the index i of the next candidate design to sample. Sherlock focuses on sampling designs on the Pareto front. To reach these designs, the algorithm has two options: increase the understanding of the design space near the Pareto front (*explore*) or directly

ALGORITHM 2: Score Algorithm

```
Input  : $\hat{y}$
Output : Scores: array of size $m$
for $j \in [1..o]$ do
    Sum[$j$] = $\sum_i \hat{y}[i, j]$
end
for $i \in [1..m]$ do
    Scores[$i$] = $\sum_j (\hat{y}[i, j] * m - \text{Sum}[j])$
end
```

sample a design on the estimated Pareto front (*exploit*). The *explore* sampling mode chooses the Pareto design that has been estimated by the surrogate model with the highest uncertainty:

$$i \leftarrow \text{argmax}(H_{\hat{P}}). \tag{7}$$

This mode increases the confidence of the estimated Pareto front. The *exploit* mode selects the design from those that have not already been sampled with the largest estimated Pareto dominance (*scores*):

$$i \leftarrow \underset{s_i \in S \backslash K}{\text{argmax}}(\text{scores}(\hat{y})). \tag{8}$$

When the surrogate model has a good estimate of the space around the Pareto front, this step leads to picking a more optimal design.

Sherlock opts to switch between explore/exploit modes when the maximum score of current estimated Pareto designs, as calculated by Algorithm 2, is less than the previous maximum score. Any decrease of maximum score results in a mode change to counteract the decreasing score and thus the decrease in estimation quality.

Algorithm 2 describes the score function. It compares the output value of each design (scaled by the number of designs) against the sum of the output of all other designs. Recall that o is the number of optimization directives and m is the number of design candidates. The first `for` loop calculates the sum of each output optimization objective across all of the design candidates. This is used like an average of all the optimization values. The second `for` loop calculates a score for each design candidate. The score is a summation of how much each of that design candidate's objectives compare to the average value of that objective (stored in $Sum[]$). As $Sum[]$ represents the average objective value scaled by m, $Scores[]$ represents the distance from the average, scaled by m. A large score indicates that the design candidate's objectives are much greater than the average value of those objectives. Thus, a large score indicates that that candidate is located far away from the center of mass of the hypervolume and is thus a high-quality design that is likely to be on or near the Pareto front. A negative score indicates that the design candidate's objectives are generally worse than the average value of the objectives and thus a poor design candidate.

Sherlock chooses between the two sampling modes based on the improvement of the predicted Pareto front. To measure the improvement, Sherlock generates a score for the prediction from a surrogate model and monitors its changes. When in exploit mode, Sherlock applies the scores function to design candidates on the estimated Pareto front that have not been already been sampled (i.e., those not in K) and picks the design candidate from that set to sample. On each iteration, Sherlock compares the current maximum score for all designs on the estimated Pareto front to the maximum score from the previous iteration. If the score is decreasing, Sherlock switches the sampling mode between explore and exploit.

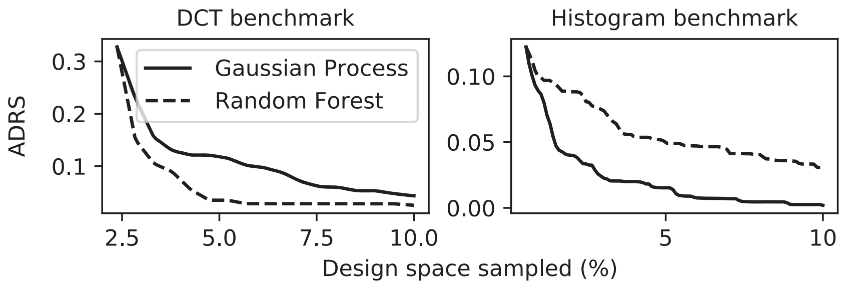


Fig. 3. Illustration of the difference of performance using Sherlock with two regression models on two benchmarks. The left graph shows that the random forest better models the DCT design space, whereas the Gaussian process is best for the histogram design space (right graph).

2.3 Surrogate Model

Sherlock can use any surrogate model based on a regression algorithm that provides a prediction uncertainty. There exist two major methods to provide an uncertainty in a regression model: the ensemble technique and Bayesian learning.

Ensemble techniques create multiple regression models, each with a subset of the known points. The predictions of different models are aggregated by voting or averaging. The uncertainty of predictions can be estimated by computing the variance over all the models. A popular ensemble model is the random forest predictor based on a set of decision trees, and it is used in several active learning frameworks.

Bayesian learning techniques leverage Bayes' theorem to progressively update statistical distributions based on provided evidence. Typically, a regression model starts with a prior distribution over its weights and combines it with the likelihood from known points to create a posterior distribution. The parameters of the posterior distribution can be used to compute a measure of uncertainty. A popular example is Gaussian process models that create a prior distribution over functions by using a kernel to express the correlation between points.

Sherlock can use any of these types of learning algorithms. We experiment with random forest and the Gaussian process because they can generally model complex design spaces, and we also create a consensus-based RBF interpolator, which provides kernel-based interpolation of unknown data and is faster to compute than a Gaussian process.

Most design spaces are inherently different, as they represent a wide variety of relationships between input variable and output design goals. Some design spaces can be modeled by a simple linear equation, whereas others require a more complex model. This is reflected by the performance of different surrogate models in active learning frameworks.

Figure 3 presents the results of running Sherlock on two different benchmarks (DCT and Histogram) using two different surrogate models. We plot ADRS against the percentage of design space sampled. Intuitively, as more designs are sampled, Sherlock does a better job of estimating the Pareto front (i.e., the ADRS moves toward 0). A lower ADRS indicates a better Pareto front estimate; an ADRS = 0 corresponds to perfectly estimating the Pareto front. In the DCT design space, a random forest surrogate model causes DSE to converge toward the Pareto front faster. In the histogram design space, a Gaussian process model makes DSE converge faster. The major point that we want to emphasize is that that different models perform better on different benchmarks. A random forest better models the DCT design space, whereas a Gaussian process is best for the histogram design space.

A common solution to pick a model is to test the algorithm on similar design spaces, and choose the most efficient one, possibly by cross validation. Instead, we propose to learn the best model

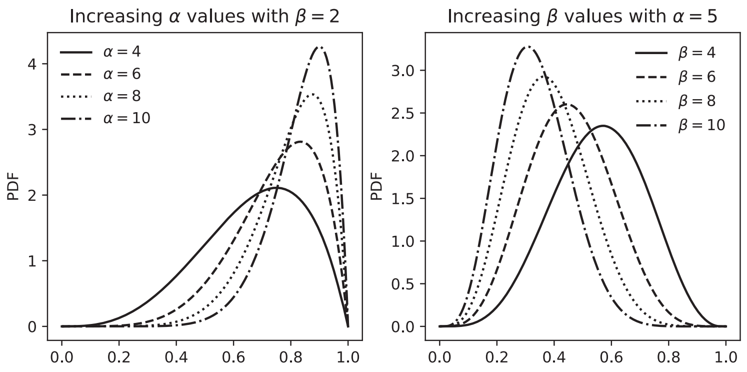


Fig. 4. Illustration of varying parameters for beta distribution PDFs. The left graph shows α being varied while β is held constant. The right graph shows α held constant and β varied. Increasing α makes a model to be more likely selected in the future, whereas increasing β makes it less likely to be selected.

using an MAB strategy that iteratively updates the importance of each model based on the Pareto set improvement.

2.4 Surrogate Model Selection Algorithm

At each iteration of the active learning process, we want to choose a surrogate model g among a pool of models $G = \{g_1, g_2, \ldots\}$. We start with no prior knowledge of which model performs better, and we want Sherlock to iteratively increase its reliance on models that generate good results.

Sherlock selects the surrogate model using an MAB strategy [28]. The MAB problem is a Bayesian optimization problem where we wish to determine the distribution of independent variables with unknown outcomes (the bandits), and choose the variable providing the best outcome. The various MAB algorithms provide a tradeoff between exploitation (observing the bandit with the best-known outcome) and exploration (observing other bandits to refine their distribution) [3].

In this case, we consider each model as a bandit. The outcome of observing one bandit is either an improvement in the current Pareto set or no improvement. In other words, we are trying to learn a Bernoulli distribution for each model. Consequently, we can select the prior distribution of the bandits as a beta distribution. We define the prior distribution with parameter θ for each model i as $P_i(\theta) = Beta(\alpha_i, \beta_i)$. We update these distributions by selecting one bandit and observing the outcome. A good choice of sampling algorithm is Thompson sampling [29], which provides a good tradeoff between exploration and exploitation [4]. The algorithm draws a random sample from each distribution, $\hat{\theta}_i \sim Beta(\alpha_i, \beta_i)\ \forall i$, then chooses the bandit with the largest sample value. The observation x of the selected bandit corresponds to the improvement of hypervolume over the known designs (hypervolume(K)), after we sample a design according to a strategy as defined in Section 2.2.3. In other words, if the model g_i improved the Pareto set, x is a positive outcome (i.e., $x = 1$) and otherwise $x = 0$. A value of $x > 0$ increases α_i, whereas a value of $x = 0$ increases the value of β_i. As can be seen in Figure 4, by increasing α_i and holding β_i constant, the likelihood that the distribution provides a larger value (closer to 1) is increased. Likewise, increasing β_i makes is more likely that smaller sample value will be selected (closer to 0). We use this updated function to compute the posterior distribution based on the outcome and use it as prior for the next iteration.

Algorithm 3 shows the details of the method and how it integrates with the Sherlock algorithm described in Algorithm 1. Note that we use an optional posterior reshaping factor r that changes the variance of the distributions. As a result, increasing the value of r favors exploitation over

ALGORITHM 3: Sherlock Model Selection Algorithm

```
Input: $X$, Models $\{g_1, g_2, \ldots, g_i, \ldots\}$, Reshape factor $r = 1$
$s_1, s_2 = TED(X)$
$K = \{s_1, s_2\}$
mode ← "explore"
prevscore = $-\infty$
Initialize: $\alpha_i = 1, \beta_i = 1 \; \forall i$
while $|K| <$ sample budget do
    $P_i(\theta) = Beta(\alpha_i, \beta_i) \; \forall i$
    $\hat{\theta}_i \sim Beta(\alpha_i, \beta_i) \; \forall i$
    $i = \text{argmax}(\hat{\theta}_i)$
    Choose model $g = g_i$
    $\hat{f} \leftarrow g(.; K)$
    $\hat{y}, H \leftarrow \hat{f}(X)$
    $\hat{P}, H_{\hat{P}} \leftarrow \text{pareto}(\hat{y}, H)$
    curscore ← $\max_{s \in \hat{P}}(\text{scores}(\hat{y}))$
    if curscore < prevscore then
        mode ← next(mode)
    end
    if mode = "explore" then
        $i \leftarrow \text{argmax}(H_{\hat{P}})$
    else if mode = "exploit" then
        $i \leftarrow \text{argmax}_{s_i \in S \setminus K}(\text{scores}(\hat{y}))$
    end
    $K \leftarrow K \cup \{s_i\}$
    prevscore ← curscore
    Compute $Hv = \text{hypervolume}(K)$
    $x = Hv > \text{prev}(Hv)$
    $\alpha_i = \alpha_i + x * r$
    $\beta_i = \beta_i + (1 - x) * r$
end
```

exploration (i.e., the model providing the best outcome gets selected more often), and the policy becomes more greedy. Increasing this value also has the side benefit that each positive outcome is given more consideration, and potential improvements from models later in the sampling process will re-adjust their importance faster. It provides a small chance to switch the most important model during the sampling process.

A model selection algorithm is valuable when the design space model changes and our past research demonstrated substantial differences between HLS FPGA design space algorithms [7]. Thus, the next question is whether Sherlock is capable of learning the best model. We generate two synthetic design spaces optimized for different regression models (random forest and Gaussian process), and we run the model selection process to verify that the algorithm can choose the proper model for each design space. The ADRS curves in Figure 5 compare the results of Sherlock using a single model and using model selection. In both datasets, model selection performs as well as the best model, or better. On the bottom, we also plot the calculated mean of the beta distribution associated with each model when running model selection. As expected, the model performing better keeps a larger mean. The spikes in the curve correspond to the reshaping factor (set to 10) designed to amplify the increase of the mean when a model performs better only later in the sampling process.

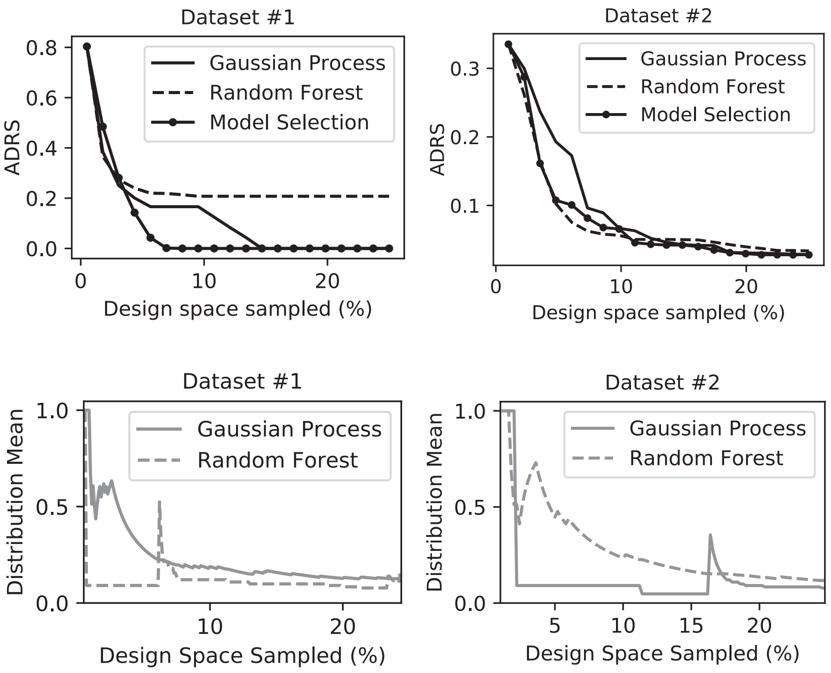


Fig. 5. Performance of the model selection algorithm on two simulated datasets. On top are the ADRS curves, and on the bottom are the calculated means of the beta distributions.

3 RESULTS

3.1 Experimental Setup

We implement Sherlock using Python 3 with the NumPy and SciPy libraries. We implement four types of surrogate models: a Gaussian process with a Matern kernel using the GPy library, a random forest from the scikit-learn library, and RBF interpolation algorithms with a consensus decision, with both a multiquadric basis and a thin plate basis. We compute an error metric based on the ground truth design spaces using the ADRS metric [19]. ADRS measures the average normalized distance between the estimated Pareto front and the reference Pareto front. The closer it is to 0, the better the estimation is. We are interested in the evolution of the ADRS value in function of the number of samples. For a better comparison between benchmarks, we normalize the number of samples to the size of the design space and report the results in function of the percentage of the space sampled. The goal of Sherlock is to produce a curve that converges to zero as fast as possible. To summarize our results, we compute the **area under the curve (AUC)** for a section of the curve (up to 30% of the space sampled) as a measure of how fast the algorithm converges toward a good solution.

We compare our results to the ATNE algorithm [13] implemented in Python 3, the ϵ-PAL [37] algorithm implemented in MATLAB, TED [34] implemented in Python 3, and the Flash [16] algorithm implemented in Python 2. The ATNE implementation leverages one of the following models: a decision tree, random forest, or RBF with a multiquadric or thin plate basis. TED only considers the input space and does not use an iterative approach. We ask TED for a certain percentage of designs to sample and increase this number to explore the design space. For all other iterative algorithms, we set the number of initial samples to 5. In ATNE, we monitor the ADRS for each design sampled until it converges and stops. For ϵ-PAL, we set the ϵ to 0, monitor the ADRS curve for each sample, and complete the curve with the samples from the estimated Pareto set after the algorithm converges. All non-deterministic algorithms are run multiple times on each benchmark, and we compute the average ADRS curve.

3.2 Dataset

We test our algorithm on a set of FPGA benchmarks that cover different types of applications. These benchmarks are FPGA applications outfitted with knobs that can be tuned, along with a test dataset to run them and measure their throughput. The knobs are software-defined parameters that translate into architecture changes. They cover typical FPGA optimizations such as pipelining, unrolling, partitioning, and some optimizations specific to each application (sliding window width, etc.). The different combinations of values for knobs create unique designs that are functionally equivalent but produce a different outcome in terms of logic utilization and throughput.

The majority of our dataset consists of the Spector benchmarks [7]. The Spector benchmarks are compiled using the Altera OpenCL SDK. Each design takes multiple hours to compile, making DSE essential. In total, all of the Spector benchmarks required more than 20,000 hours of compilation time. All designs of these applications have been compiled and executed on a Terasic DE5 board with a Stratix V FPGA to create design spaces containing between 200 and 1,500 designs each. We also add the Iterative Closest Point (ICP) algorithm from our earlier work [8] that is implemented on FPGA using OpenCL and uses knobs to create a similar design space as in Spector, with 1,276 designs. We use these provided design spaces that consist of the knob values, the actual FPGA area utilization, and the measured throughput of each design. Please consult the Spector repository and technical paper for more information [7].

The goal of a DSE framework is to search a predefined space for optimal designs. Therefore, our ground truth consists of the set of Pareto-optimal designs in each design space. We run Sherlock by considering that the outcome of each design is unknown and let the tool incrementally find and improve an estimated Pareto set. We can then compute the ADRS metric between the estimated Pareto set and the ground truth set that we defined initially.

3.3 Results

The first experiment aims to understand the effect of different surrogate models. We do this by comparing the results of Sherlock using four different regression models. We show that the regression model has a large role in the quality of results. Additionally, the experiment aims to understand how the general Sherlock technique compares to different styles of DSE techniques (specifically ATNE, ϵ-PAL, TED, and Flash).

Figure 6 presents the ADRS curve for ATNE with the four models: ϵ-PAL, TED, Flash, four versions of Sherlock with statically defined surrogate models, and Sherlock with model selection. The experiment varies the percentage of designs that are sampled shown on the x-axis. We calculate the average ADRS value across all benchmarks at each sample. The ADRS is calculated against the ground truth design spaces that were evaluated and provided for each of the benchmarks. Intuitively, the ADRS should decrease as the number of sampled designs increases.

We run Sherlock with four different regression models: Gaussian process, random forest, RBF (multiquadric), and RBF (thin plate) as well as model selection. The results focus on the first 30% of the design spaces; 30% is a large budget for most applications with a slow evaluation time, and in our test cases, it is always sufficient to reach an ADRS below 1% with the best regression model.

Figure 6 indicates that ϵ-PAL is generally inferior to all of the techniques. TED has slightly better but still overall poor results. ATNE performs poorly when the sampled design space is smaller (less than 10%) but improves as the number of samples increases. Sherlock using a static random forest model is the worst-performing version of Sherlock followed by Sherlock Gaussian process.

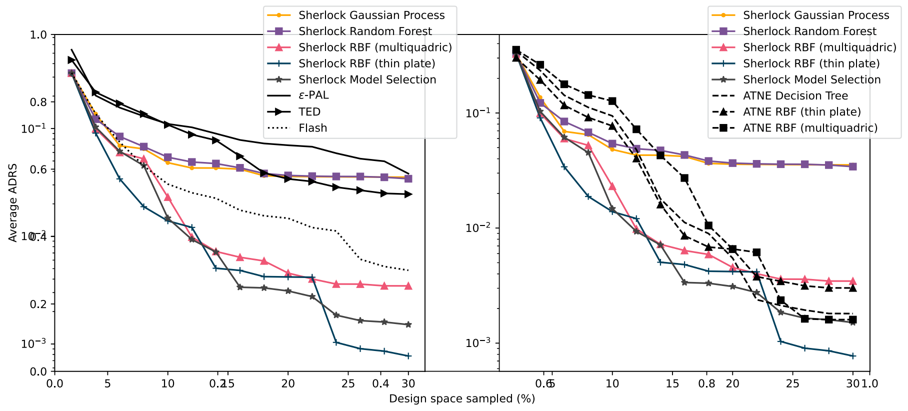


Fig. 6. Average performance of several algorithms on all FPGA benchmarks. We plot the error (ADRS—lower is better) against the percentage of design space sampled. We test Sherlock and ATNE with multiple regression models.

The two versions of Sherlock using the RBF model perform the best out of all of the techniques. Model selection appears to generally follow the RBF model, providing most of the benefit of RBF without needing to commit to a single model. Comparing ATNE with RBF and Sherlock with RBF, it is clear to tell that RBF does generally model the data better, but it does not account for all of the improvement provided by Sherlock over ATNE. Figure 6 shows that, regardless of the model, Sherlock performs better or similar to other algorithms in the first 5% of the spaces. After that, Sherlock performs differently based on the chosen model. On average, using the RBF interpolator with a thin plate kernel provides the best average ADRS.

However, the average ADRS curves do not entirely reflect the performance on individual benchmarks and is especially skewed by the FIR benchmark, which is more difficult for several versions of Sherlock due to some local minima. Thus, although it is tempting to just set Sherlock to always use the RBF thin plate surrogate model, our later results indicate that different models better estimate different design spaces. This motivates the benefits for tailoring the model toward the design space as Sherlock does with its model selection algorithm (Algorithm 3).

To better understand the results across benchmarks, we use the AUC over the first 30% of the design space sampled as a measure of the quality of a particular DSE algorithm on each individual benchmark. The AUC metric calculates the area of a given ADRS curve, which provides a measure of the convergence of an ADRS curve. This metric puts more weight at the beginning of the curve, where we expect the error to decrease quickly.

Figure 7 shows the AUC for ϵ-PAL, ATNE with four models, Flash, TED, and five versions of Sherlock for each benchmark. As before, we show Sherlock using four fixed models and add the results of Sherlock using model selection. The results indicate that the model plays a large role in the quality of the results. For example, Sherlock using a Gaussian process has a substantially lower AUC than all of the other techniques in the BFS sparse design space, whereas Sherlock using a random forest model provides the best results for DCT. Histogram is best using Sherlock RBF multiquadric. Sherlock RBF thin plate is not the best in any individual benchmark but generally performs well across the benchmarks as indicated in Figure 6. Thus, the model clearly plays an important role in obtaining the best results. Furthermore, one model does not clearly dominate in terms of consistently providing the best result across all applications.

Sherlocks's model selection algorithm dynamically chooses the model, which can change during the DSE process, with the goal of finding the model that best guides the search for the Pareto front.

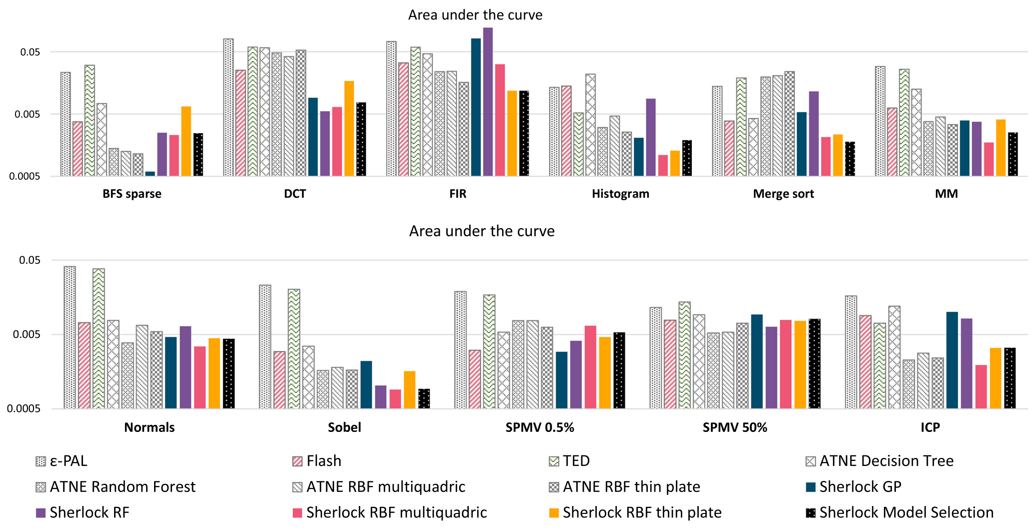


Fig. 7. Performance of ϵ-PAL, ATNE with four models, Flash, TED, and five versions of Sherlock across the individual benchmarks. We compute the area under the ADRS curve as a measure of convergence over the first 30% of design space sampled. A lower value means that the algorithm reaches a better estimation of the Pareto front faster. The Sherlock results include four versions using different regression models: Gaussian process (GP), random forest (RF), and two RBFs—multiquadric and thin plate. Model selection is a version of Sherlock that uses Algorithm 3 to dynamically switch between those four different models with the goal of automatically adapting to the best model. ATNE also uses four models: decision tree, random forest, RBF with a thin plate basis, and RBF with a multiquadric basis. ATNE with a decision tree model is the original implementation.

In theory, the model selection results should be equal to or superior than the best results of the Sherlock fixed model. This is clearly not the case, as the Model selection results are inferior to one or more the Sherlock fixed models in most of the benchmarks. For example, in the Histogram benchmark, Sherlock RBF multiquadric and Sherlock RBF thin plate both give better performance than Sherlock with Model selection. This indicates that the model selection process is not perfect (i.e., a good model selection algorithm should statically pick only one model if that would give the best results). Thus, an ideal model selection algorithm would have just picked the RBF multiquadric for the entire duration. Clearly, this is not what happened. However, model selection can provide superior results; model selection performs better than all the Sherlock fixed models in merge sort. And on average, model selection works very well as we discuss in the following.

Figure 8 shows the average AUC over all benchmarks. A smaller AUC represents a faster convergence of the algorithm toward the true Pareto front. The best-performing static model on average (RBF with a thin plate kernel) produces an AUC 1.7× smaller than Flash, 2.7× smaller than ATNE, 4.5× smaller than TED, and 5× smaller than ϵ-PAL. However, the performance of each model varies with individual benchmarks. Although we could reasonably pick the model with the best average result, we would ideally allow Sherlock to decide the best surrogate model.

In many cases, the performance of Sherlock with model selection is comparable to Sherlock with the RBF interpolator, and better in some cases. On average, this algorithm performs better than the other solutions without having to choose a particular model. The AUC is about 2× smaller than Flash, 3× smaller than ATNE, and 6× smaller than ϵ-PAL. The Gaussian process and random forest implementations of Sherlock do not converge very well in many benchmarks as a result of being stuck in local minimum regions. The model selection process can more easily avoid these local

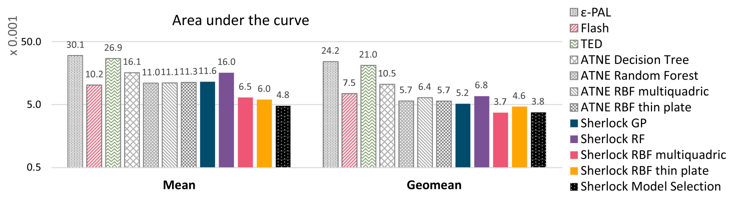


Fig. 8. Average AUC for the benchmarks presented in Figure 7. We present the arithmetic mean and the geometric mean to take into account the variability of the results. The values are scaled by 1,000 for readability.

minima by selecting a different model. Certain models such as FIR filter contain a local minimum that creates a high variance in the performance of different models, and the selection process clearly helps in this case. Conversely, a benchmark like SPMV 0.5% depends more on the learning rate of the chosen model, and the overhead of choosing between multiple models is more obvious.

Figure 9 shows the ADRS curves for all of the benchmarks and all of the algorithms. These are the most complete results that have been summarized previously using average ADRS (Figure 6) and AUC (Figures 7 and 8). The results provide some more information about the variability of the performance of different DSE algorithms on the different designs. Rarely does one model dominate across an entire design sample size; some models work better in the lower sample regime, whereas other models do better with more samples.

Based on the results presented, Sherlock is able to provide similar and often better results compared to similar frameworks. Paired with its model selection feature, Sherlock is able adapt itself well to a previously unknown design space. Although model selection generally performs competitively, it is possible to choose a model that fits the space better than Sherlock's general model selection. This indicates that model selection is important. Sherlock does a good job of model selection, but it could certainly be improved.

3.4 Execution Time

We collected the execution time of each of the DSE algorithms on every dataset. These times are plotted in Figure 10. These algorithms were collected on an AMD Ryzen 2700 with 16 GB of RAM and an Nvidia 1060 with 6 GB of VRAM on a system running Ubuntu 20.04 LTS. Additionally, we estimate the time to evaluate a sample design (i.e., perform a complete synthesis). The exact synthesis times per design are not available for the Spector benchmarks. We assume that each sample takes an average of 2 hours to generate, which was the average time cited in the Spector benchmarks [7]. In general, sample generation time may vary greatly and be difficult to predict. Regardless, sample evaluation dominates the overall time for DSE.

Sherlock is generally among the lowest when it comes to total execution time. This is largely due to Sherlock's ability to quickly converge to the Pareto front, leading to a reduced number of samples required to be evaluated, as shown in Figure 9.

4 RELATED WORK

FPGA HLS tools allow designers to explore vastly different architectures using built-in optimizations including pipelining, memory optimization, and bitwidth optimizations [9]. Efficiently exploring the design space is important since FPGA HLS is a time-consuming and costly process. DSE techniques enable the designer to specify potential optimization options and determine the best specifications that produce the most optimal architectures. DSE facilitates faster and more effective architectural optimizations and reaching optimal solutions more quickly than a manual search.

Sherlock targets DSE problems where the evaluation of any individual sample is a computer-intensive and time-consuming process (on the order of hours or more). For example, compiling

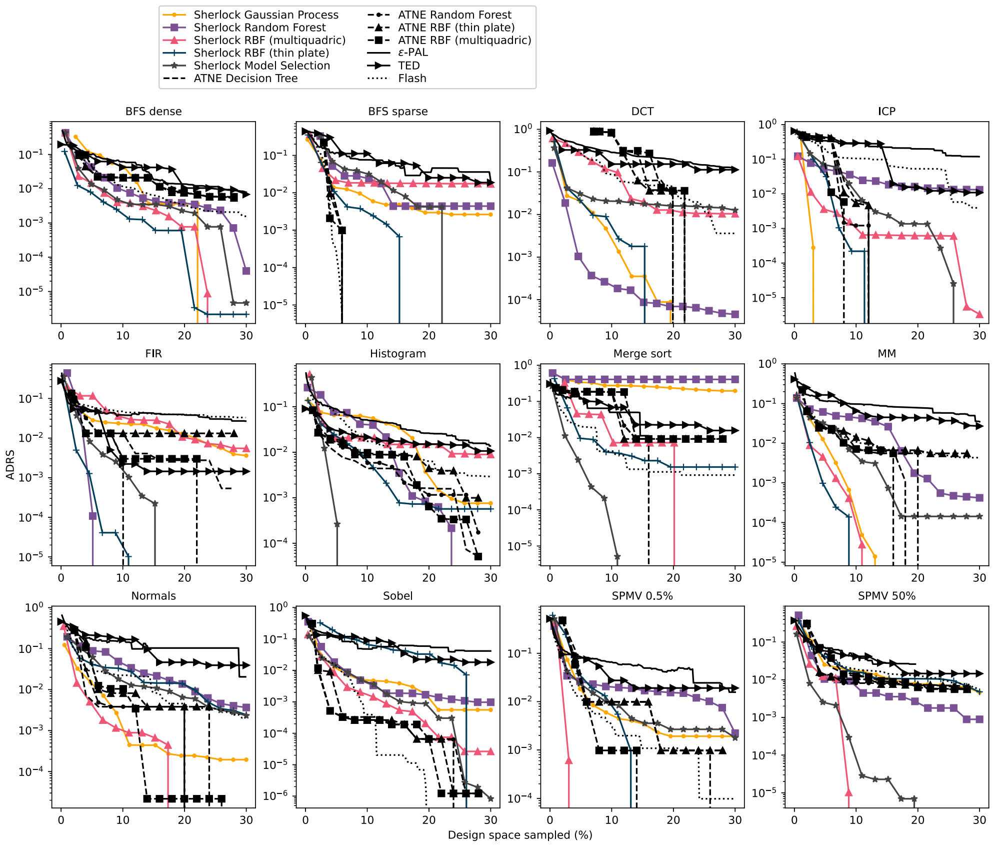


Fig. 9. Comparison of the ADRS curves for multiple algorithms on all FPGA benchmarks.

an HLS design to bitstream can easily take hours primarily due to a physical synthesis process. In these DSE scenarios, limiting the number of synthesis runs is very important, as each single design not sampled can save hours of optimization time. In such scenarios, one cannot perform a brute force exploration of the entire space, and they are limited to evaluating only a small number of samples.

Many strategies exist to effectively explore design spaces to find the optimal set of design parameters [21, 25]. We describe those that are most relevant to our work with respect to the type of DSE problem that they attempt to solve and the strategies that they employ to perform the exploration process.

Evaluation-based methods measure the exact quantity of the target objectives to optimize. In hardware design, this method consists of compiling an architecture specification to the target system, running the compiled design using an application-specific dataset, and measuring the throughput, area utilization, power consumption, and other optimization goals. These measurements give an accurate representation of how the design actually performs at the cost of long synthesis times. One solution to accelerate this evaluation process is to parallelize it and employ a smart division of the space to attribute the computing resources. Xu et al. [32] use an MAB algorithm to balance the computing resources between different portions of the space and leverages the exploration-exploitation tradeoff of MAB to iteratively allocate more resources to the optimal subspaces.

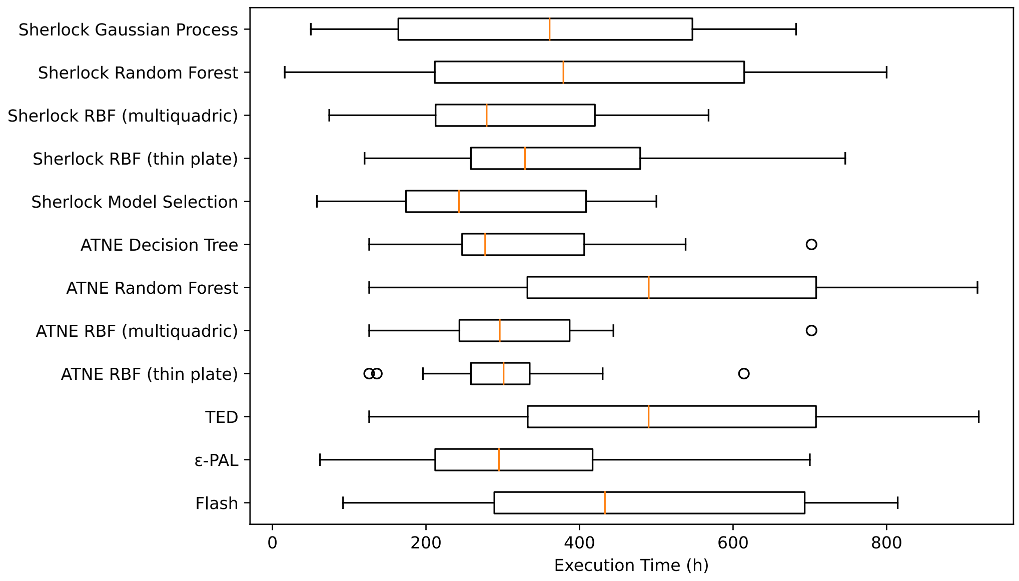


Fig. 10. Comparison of the execution time for multiple algorithms on all FPGA benchmarks. Sample generation is estimated at an average of 2 hours per sample [7].

Predictive DSE methods develop an analytical model for the design space, which allows for faster and more comprehensive sampling by using this approximated model. This enables the rapid exploration of the design space by sampling more potential solutions. DeSpErate++ [11] uses a scheduling system to predict and efficiently simulate the designs and quickly explores the design space of the analytical model. Another approach builds compute and memory models based on the compiler internal representation that can be used as surrogate models to significantly speed up the DSE process [30, 31]. Their performance prediction error varies between 4% and 16%. Schafer [22] explores the design space by building a probabilistic model for each type of optimization. He performs a fast exploration by using an ant colony optimization algorithm and obtains an error of 1.7% on SystemC benchmarks. Lin-Analyzer [35] and MPSeeker [36] propose tools to analyze the structure of the HLS code directly before the synthesizing steps occur. Their model achieves a 400 to 4,000× speedup and a final accuracy of 95%. Generally, predictive-based methods exploit features specific to the optimized application. These specific features make predictive approaches difficult to generalize to other applications. Sherlock learns the appropriate model for the application at hand, which allows it to generalize better than these predictive approaches that rely on application-specific features.

Metaheuristics are commonly used as a general optimization framework for DSE. Simulated annealing is a probabilistic technique for approximating a function that naturally lends itself to assumption-free modeling of a design space [6, 24, 26]. Evolutionary DSE [1, 5, 15, 20] uses genetic algorithms to converge toward an optimal solution. Particle swarm optimization [14, 18] generates a population of particles searching the space of designs and using various metrics to advance toward the optimal solutions. These techniques require a large number of samples to converge, and therefore are more adapted to problems where the evaluation of each sample is a fast process, but the space is too large to be evaluated entirely, which usually does not apply to FPGA HLS.

Iterative machine learning algorithms aim to minimizes the number of designs to evaluate while maximizing the quality of the DSE model. Liu and Carloni [10] describe an iterative approach based on TED, randomized selection, and random forests. ATNE is an active learning framework based on non-Pareto elimination [13]. ATNE creates multiple regression models that estimate the design space from different subsets of the known data, then computes an elimination threshold from the variance of the predictions. Based on this threshold, designs predicted to be dominated are eliminated from consideration. The algorithm then samples a new design and iterates until convergence. By focusing on estimating only the Pareto-optimal designs, ATNE can quickly converge toward the Pareto front. The Hypermapper framework [2, 17] performs active learning by modeling known designs with a random forest regression algorithm and simultaneously sampling all predicted Pareto-optimal solutions. The algorithm iterates until a sampling budget is reached. This framework is optimized toward large design spaces where it is reasonable to perform a high number of design evaluations (100 to 300 samples per iteration). Their framework is applied to the optimization of Simultaneous Localization And Mapping (SLAM) algorithms. ϵ-Pareto Active Learning (ϵ-PAL) [37] is an improvement over PAL [38] and uses a Gaussian process as a regression model to predict an uncertainty region for each predicted design output. By using these uncertainty regions, the algorithm can discard non-optimal designs with high accuracy and progressively build a predicted Pareto-optimal set with an ϵ margin. The sampling process is based on minimizing the uncertainty of the predictions. Sherlock is similar to ϵ-PAL and ATNE, but without the pruning step, which can potentially eliminate optimal designs from consideration, especially when the regression model is not adapted to the design space being searched.

Flash [16] is a method to explore the space of possible configurations for software systems. Flash is a sequential model-based optimization that uses decision trees to iteratively sample configurations. It uses the information from prior selected samples to inform the best future samples. Our results show that Sherlock performs better than Flash in most cases.

Sherlock aims to address the issue that one model cannot adequately address all design spaces. SPIRIT [33] determines the Pareto front using iterative refinement while employing spectral analysis to determine uncertainty in the design space model. They study six different response surface models. They note that different models provide different benefits and decide on RBF because it is generally best at modeling their DSE problem. Although our results agree that RBF is a good model on average (see Figure 6), we show that different design spaces are modeled better by different models (see Figure 8), which indicates that statically determining a model is not an optimal solution. Sherlock adaptively chooses the best model based on the design space and will likely be able to better model designs spaces that RBF does not model accurately.

OpenTuner [1] develops an MAB-based technique to choose which search algorithm should be used to select samples. Sherlock uses a similar technique, tailored to our active learning search algorithm, that can choose which regression model performs better on each design space.

Prospector [12] uses Bayesian optimization that builds a probabilistic model of the design space and iteratively determines the best points to sample. Prospector's Bayesian Optimization Unit uses a squared exponential kernel as a Gaussian process to create a model of the design space. Sherlock uses a Gaussian process as one of its models. The results when Sherlock uses the Gaussian process do not perform as well as other models on average, although it does perform very well on some benchmarks. Prospector uses PESMO to select the sample points with the goal of reducing the entropy of the Pareto front. PESMO focuses on exploration at the beginning of the process and exploitation near the end. Sherlock changes its focus between exploration and exploitation dynamically depending on the progress of the DSE search.

5 CONCLUSION

We presented Sherlock—an evaluation-based, multi-objective, DSE framework. Sherlock is an active learning algorithm, heavily focused on improving the set of optimal designs at each iteration, and as such it converges very quickly toward a low-error solution. Sherlock is capable of using specific models if a similar space is known, but Sherlock excels in scenarios where the design space is not already known by providing an intelligent way to select a model to represent the space by using its MAB-based algorithm. We have tested our framework with multiple regression models that present a wide variance in the quality of results on different benchmarks. In general, we have found that simple RBF interpolation functions perform better than traditional random forest or Gaussian process models on FPGA design spaces. The results of this model selection are consistent over multiple benchmarks and provide better average performance.

REFERENCES

[1] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U. O'Reilly, and S. Amarasinghe. 2014. OpenTuner: An extensible framework for program autotuning. In *Proceedings of the 2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT'14)*. 303–315.

[2] B. Bodin, L. Nardi, M. Z. Zia, H. Wagstaff, G. S. Shenoy, M. Emani, J. Mawer, et al. 2016. Integrating algorithmic parameters into benchmarking and design space exploration in 3D scene understanding. In *Proceedings of the 2016 International Conference on Parallel Architecture and Compilation Techniques (PACT'16)*. 57–69. https://doi.org/10.1145/2967938.2967963

[3] Sébastien Bubeck and Nicolo Cesa-Bianchi. 2012. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *arXiv preprint arXiv:1204.5721*.

[4] Olivier Chapelle and Lihong Li. 2011. An empirical evaluation of Thompson sampling. In *Advances in Neural Information Processing Systems 24*, J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger (Eds.). Curran Associates, 2249–2257. http://papers.nips.cc/paper/4321-an-empirical-evaluation-of-thompson-sampling.pdf.

[5] Amin Zhou, Bo-Yang Qu, Hui Li, Shi-Zheng Zhao, Ponnuthurai Nagaratnam Suganthan, and Qingfu Zhang. 2011. Multiobjective evolutionary algorithms: A survey of the state of the art. *Swarm and Evolutionary Computation* 1, 1 (2011), 32–49. https://doi.org/10.1016/j.swevo.2011.03.001

[6] Piotr Czyżżak and Adrezej Jaszkiewicz. 1998. Pareto simulated annealing—A metaheuristic technique for multiple-objective combinatorial optimization. *Journal of Multi-Criteria Decision Analysis* 7, 1 (1998), 34–47.

[7] Q. Gautier, A. Althoff, P. Meng, and R. Kastner. 2016. Spector: An OpenCL FPGA benchmark suite. In *Proceedings of the 2016 International Conference on Field-Programmable Technology (FPT'16)*. 141–148. https://doi.org/10.1109/FPT.2016.7929519

[8] Quentin Gautier, Alric Althoff, and Ryan Kastner. 2019. FPGA architectures for real-time dense SLAM. In *Proceedings of the 2019 IEEE 30th International Conference on Application-Specific Systems, Architectures, and Processors (ASAP'19)*. IEEE, Los Alamitos, CA.

[9] Ryan Kastner, Janarbek Matai, and Stephen Neuendorffer. 2018. Parallel programming for FPGAs. *arXiv preprint arXiv:1805.03648*.

[10] Hung-Yi Liu and Luca P. Carloni. 2013. On learning-based methods for design-space exploration with high-level synthesis. In *Proceedings of the 50th Annual Design Automation Conference (DAC'13)*. ACM, New York, NY, Article 50, 7 pages. https://doi.org/10.1145/2463209.2488795

[11] G. Mariani, G. Palermo, V. Zaccaria, and C. Silvano. 2015. DeSpErate++: An enhanced design space exploration framework using predictive simulation scheduling. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34, 2 (Feb. 2015), 293–306. https://doi.org/10.1109/TCAD.2014.2379634

[12] Atefeh Mehrabi, Aninda Manocha, Benjamin C. Lee, and Daniel J. Sorin. 2020. Bayesian optimization for efficient accelerator synthesis. *ACM Transactions on Architecture and Code Optimization* 18, 1 (2020), 1–25.

[13] Pingfan Meng, Alric Althoff, Quentin Gautier, and Ryan Kastner. 2016. Adaptive threshold non-Pareto elimination: Re-thinking machine learning for system level design space exploration on FPGAs. In *Proceedings of the 2016 Design, Automation, and Test in Europe Conference and Exhibition (DATE'16)*.

[14] Vipul Kumar Mishra and Anirban Sengupta. 2014. MO-PSE: Adaptive multi-objective particle swarm optimization based design space exploration in architectural synthesis for application specific processor design. *Advances in Engineering Software* 67 (2014), 111–124. https://doi.org/10.1016/j.advengsoft.2013.09.001

[15] Caitlin T. Mueller and John A. Ochsendorf. 2015. Combining structural performance and designer preferences in evolutionary design space exploration. *Automation in Construction* 52 (2015), 70–82. https://doi.org/10.1016/j.autcon.2015.02.011

[16] V. Nair, Z. Yu, T. Menzies, N. Siegmund, and S. Apel. 2020. Finding faster configurations using FLASH. *IEEE Transactions on Software Engineering* 46, 7 (2020), 794–811. https://doi.org/10.1109/TSE.2018.2870895

[17] L. Nardi, B. Bodin, S. Saeedi, E. Vespa, A. J. Davison, and P. H. J. Kelly. 2017. Algorithmic performance-accuracy trade-off in 3D vision applications using HyperMapper. In *Proceedings of the 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW'17)*. 1434–1443. https://doi.org/10.1109/IPDPSW.2017.107

[18] G. Palermo, C. Silvano, and V. Zaccaria. 2008. Discrete particle swarm optimization for multi-objective design space exploration. In *Proceedings of the 2008 11th EUROMICRO Conference on Digital System Design Architectures, Methods, and Tools*. 641–644. https://doi.org/10.1109/DSD.2008.21

[19] G. Palermo, C. Silvano, and V. Zaccaria. 2009. ReSPIR: A response surface-based Pareto iterative refinement for application-specific design space exploration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28, 12 (2009), 1816–1829.

[20] Maurizio Palesi and Tony Givargis. 2002. Multi-objective design space exploration using genetic algorithms. In *Proceedings of the 10th International Symposium on Hardware/Software Codesign (CODES'02)*. ACM, New York, NY, 67–72. https://doi.org/10.1145/774789.774804

[21] Jacopo Panerati, Donatella Sciuto, and Giovanni Beltrame. 2017. Optimization strategies in design space exploration. In *Handbook of Hardware/Software Codesign*, S. Ha and J. Teich (Eds.). Springer, 189–216.

[22] Benjamin Carrion Schafer. 2016. Probabilistic multiknob high-level synthesis design space exploration acceleration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 3 (2016), 394–406.

[23] Benjamin Carrion Schafer and Anushree Mahapatra. 2014. S2CBench: Synthesizable systemC benchmark suite for high-level synthesis. *IEEE Embedded Systems Letters* 6, 3 (2014), 53–56.

[24] Benjamin Carrion Schafer, Takashi Takenaka, and Kazutoshi Wakabayashi. 2009. Adaptive simulated annealer for high level synthesis design space exploration. In *Proceedings of the 2009 International Symposium on VLSI Design, Automation, and Test*. IEEE, Los Alamitos, CA, 106–109.

[25] Benjamin Carrion Schafer and Zi Wang. 2019. High-level synthesis design space exploration: Past, present, and future. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 10 (2019), 2628–2639.

[26] Paolo Serafini. 1994. Simulated annealing for multi objective optimization problems. In *Multiple Criteria Decision Making*. Springer, 283–292.

[27] Burr Settles. 2009. *Active Learning Literature Survey*. Computer Sciences Technical Report 1648. University of Wisconsin–Madison.

[28] William R. Thompson. 1933. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika* 25, 3–4 (1933), 285–294.

[29] William R. Thompson. 1933. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika* 25, 3–4 (1933), 285–294. http://www.jstor.org/stable/2332286

[30] S. Wang, Y. Liang, and Wei Zhang. 2017. FlexCL: An analytical performance model for OpenCL workloads on flexible FPGAs. In *Proceedings of the 2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC'17)*. 1–6. https://doi.org/10.1145/3061639.3062251

[31] Zeke Wang, Bingsheng He, Wei Zhang, and Shunning Jiang. 2016. A performance analysis framework for optimizing OpenCL applications on FPGAs. In *Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*. IEEE, Los Alamitos, CA, 114–125.

[32] Chang Xu, Gai Liu, Ritchie Zhao, Stephen Yang, Guojie Luo, and Zhiru Zhang. 2017. A parallel bandit-based approach for autotuning FPGA compilation. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'17)*. ACM, New York, NY, 157–166. https://doi.org/10.1145/3020078.3021747

[33] Sotirios Xydis, Gianluca Palermo, Vittorio Zaccaria, and Cristina Silvano. 2014. SPIRIT: Spectral-aware Pareto iterative refinement optimization for supervised high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34, 1 (2014), 155–159.

[34] Kai Yu, Jinbo Bi, and Volker Tresp. 2006. Active learning via transductive experimental design. In *Proceedings of the 23rd International Conference on Machine Learning*. ACM, New York, NY, 1081–1088.

[35] Guanwen Zhong, Alok Prakash, Yun Liang, Tulika Mitra, and Smail Niar. 2016. Lin-Analyzer: A high-level performance analysis tool for FPGA-based accelerators. In *Proceedings of the 53rd Annual Design Automation Conference (DAC'16)*. ACM, New York, NY, Article 136. https://doi.org/10.1145/2897937.2898040

[36] G. Zhong, A. Prakash, S. Wang, Y. Liang, T. Mitra, and S. Niar. 2017. Design space exploration of FPGA-based accelerators with multi-level parallelism. In *Proceedings of the Design, Automation, and Test in Europe Conference and Exhibition (DATE'17)*. 1141–1146. https://doi.org/10.23919/DATE.2017.7927161

[37] Marcela Zuluaga, Andreas Krause, and Markus Püschel. 2016. ϵ-PAL: An active learning approach to the multi-objective optimization problem. *Journal of Machine Learning Research* 17, 1 (Jan. 2016), 3619–3650. http://dl.acm.org/citation.cfm?id=2946645.3007057.

[38] Marcela Zuluaga, Guillaume Sergent, Andreas Krause, and Markus Püschel. 2013. Active learning for multi-objective optimization. In *Proceedings of the 30th International Conference on Machine Learning*. 462–470.

Received July 2021; revised January 2022; accepted January 2022