Multi-objective Optimization of Parametric Reconfigurable Designs Using Machine Learning Optimizer

Maciej Kurek Tobias Becker Wayne Luk

***Abstract*—This paper presents an enhancement of a novel technique that uses meta-heuristics and machine learning to automate the optimization of design parameters for reconfigurable designs. We previously developed Machine Learning Optimizer (MLO) which from a number of benchmark executions automatically derive the characteristics of the parameter space and creates a surrogate model through regression and classification. Based on this surrogate model, design parameters are optimized with meta-heuristics. We present a new multi-objective flavor of the MLO. We evaluate the extended algorithm using two applications as well as investing Field programmable gate array (FPGA) architecture exploration.**

I. Introduction

We shown it to be useful to construct surrogate models of fitness functions representing design quality of reconfigurable hardware designs [1], [2]. As these models are orders of magnitude faster to evaluate than the actual benchmarks and bitstreams, they can substantially accelerate optimization thus allowing for an automated approach. This is the motivation behind our development of the MLO tool which we apply to the problem of reconfigurable designs parameter optimization. Recently however; issues like power efficiency or size have become more prominent, expanding the problem into the multi-objective optimization domain. We present a new multi-objective flavor of MLO and evaluate it using a number of examples. The contributions of this paper are:

- A mathematical characterization of multi-objective optimization in reconfigurable applications. We extend the previous description [2] (Section III).
- A new multi-objective version of MLO. We show how multiple Bayesian regressors and multi-objective meta-heuristics can be interlinked (Section IV).
- An evaluation of the extended MLO approach using thre case studies: (a) a previously used [1] throughput of a quadrature based financial application with varied precision (Section V), and (b) and .

II. Background

FPGAs allow designs that are customized to the requirement of the application, designers are often confronted with a very large parameter space. As a result the parameter space exploration can take an immense amount of time. A number of researchers approach the problem of high-cost fitness functions and large design spaces in various fields [**?**], [**?**], [**?**], [**?**], [**?**] by having fitness functions combined with fast-to-compute surrogate models provided by a Gaussian Process (GP) for decreasing evaluation time. However most current surrogate models only consist of a regressor and do not take into account possible invalid configurations within the design space. Eliminating regions producing invalid designs trims the design space, substantially reducing the complexity of the problem.

The reconfigurability and customization of FPGA designs potentially increases performance and efficiency. Unfortunately, the optimization of reconfigurable designs often requires substantial effort from designers who have to analyze the application, create models and benchmarks and subsequently use them to optimize the design. This process often involves adjusting multiple design parameters such as numerical precision, degree of pipelining or number of cores. One could proceed with automated optimization based on an exhaustive search through design parameters which are derived from application benchmarks; however, this is unrealistic since benchmark evaluations involve bitstream generation and code execution which often takes hours of compute time. Surrogate models approximating fitness functions by substituting lengthy evaluations with estimations based on closeness in a design space have been investigated in reconfigurable computing [**?**]. The work covers surrogate models for circuit synthesis from higher level languages (HLL), rather than parameter optimization.

GP is a machine learning technology based on strict theoretical fundamentals and Bayesian theory [**?**], [**?**]. GP does not require a predefined structure, can approximate arbitrary function landscapes including discontinuities, and includes a theoretical framework for obtaining the optimum hyper-parameters [**?**]. An advantage of GP is that it provides a predictive distribution, not a point estimate.

A Gaussian process is a collection of random variables, any finite set of which have a joint Gaussian distribution. A Gaussian process is completely specified by its mean function $m(\mathbf{x})$ and the covariance (kernel) function $k(\mathbf{x}, \mathbf{x}')$:

$$\hat{f}(\mathbf{x}) \sim \mathcal{GP}(m(\mathbf{x}), k(\mathbf{x}, \mathbf{x}')) \tag{1}$$

The $k(\mathbf{x}, \mathbf{x}')$ expresses the covariance between pairs of random variables, and in regression analysis it expresses the relation between input-output pairs. This is based on a training set $\mathcal{D}$ of n observations, $\mathcal{D} = (\mathbf{x}_i, y_i)|i = 1, ...n$, where $\mathbf{x}$ denotes an input vector, y denotes a scalar output. The column vector inputs for all n cases are aggregated in the $D \times n$ design

matrix X, and the outputs are collected in the vector $\mathbf{y}$. The goal of Bayesian forecasting is to compute the distribution $p(\hat{f}|\mathbf{x}_*, \mathbf{y}, X)$ of the function $\hat{f}$ at unseen input $\mathbf{x}_*$ given a set of training points $\mathcal{D}$. Using Bayes rule, the predictive posterior for the Gaussian process $\hat{f}$ and the predicted scalar outputs $\hat{f}(\mathbf{x}_*) = y_*$ can be obtained.

Support Vector Machine (SVM) is a maximum margin classifier, which constructs a hyperplane used for classification (or regression) [**?**]. SVMs use kernel functions $k(\mathbf{x}, \mathbf{x}')$ to transform the original feature space to a different space where a linear model is used for classification. SVMs are a class of decision machines and so do not provide posterior probabilities. There is a training set $\mathcal{D}$ of n observations, $\mathcal{D} = (\mathbf{x}_i, t_i)|i = 1, ...n$, where $\mathbf{x}$ denotes an input vector, t denotes a target value. The column vector inputs for all n cases are aggregated in the $D \times n$ design matrix X, and the targets in the vector $\mathbf{t}$. The goal is to classify an unseen input $\mathbf{x}_*$ based on X and $\mathbf{t}$ by computing a decision boundary.

Particle Swarm Optimization (PSO) is a population-based meta-heuristic based on the simulation of the social behavior of birds within a flock [**?**]. The algorithm starts by randomly initializing N particles where each individual is a point in the $\mathcal{X} = \mathbb{R} \times ... \times \mathbb{R}$ search space. The population is updated in an iterative manner where each particle is displaced based on its velocity v_{id}. The criteria for termination of the PSO algorithm can vary, and usually are determined by a time budget. The x_{id} represents the dth coordinate of particle i from the set X_* of N particles, where particle is a point within $\mathcal{X}$. In the most basic form of PSO Eq. 2-3 govern movement of particles. $r_1 \sim U(0,1)$ and $r_2 \sim U(0,1)$ are two independent uniformly distributed random numbers, c_1 and c_2 are acceleration coefficients and p_{gd} and p_{id} are dth coordinates of the global best and personal best positions. p_{gd} is updated when a new global best fitness is found and p_{id} is updated when a particle improves over its best fitness.

$$v_{id} = v_{id} + c_1 r_1 (p_{id} - x_{id}) + c_2 r_2 (p_{gd} - x_{id}) \tag{2}$$

$$x_{id} = x_{id} + v_{id} \tag{3}$$

III. Optimization Approach

Traditionally, optimization of reconfigurable applications is carried out by building benchmarks and relevant tools, and the associated analytical models [**?**], [**?**]. This involves the following steps:

1) Build application and a benchmark returning design quality metrics.
2) Specify search space boundaries and optimization goal.
3) Create analytical models for the design.
4) Create tools to explore the parameter space.
5) Use the tools to find optimal configurations, guided by the models in step 3.
6) If result is not satisfactory, redesign.

In our approach the user supplies a benchmark along with constraints and goals, and the MLO automatically carries out the optimization (Algorithm 1). Our approach consists of the following steps:

1) Build application and benchmark returning design quality metrics.
2) Specify search space boundaries and optimization goal.
3) Automatically optimize design with MLO.
4) If result is not satisfactory, redesign or revised time budget and search space.

Our idea of surrogate modeling is illustrated in Fig. 1. The MLO algorithm explores the parameter space by evaluating different benchmark configurations as presented in the left figure. The results obtained during evaluations are used to build a surrogate model which provides a regression of the fitness function and identifies invalid regions of the parameter space. A meta-heuristic (currently PSO) guides the exploration of the parameter space using the surrogate model.

A. Parameter Space

The parameter space $\mathcal{X}$ of a reconfigurable design is spanned by discrete and continuous parameters determining both the architecture and physical settings of FPGA designs. A vector $\mathbf{x}$ represents a parameter configuration within the parameter space $\mathcal{X} = \mathcal{X}_1 \times ... \times \mathcal{X}_D$ such that any $\mathcal{X}_d \subseteq \mathbb{R}$. If $\mathcal{X}_d \subseteq \mathbb{Z}$, its discretization level is independent of other dimensions. $\mathcal{X}_d$ can be bounded with upper and lower limits U_d, L_d such that for all x_d, $L_d \leq x_{id} \leq U_d$. An example of a continuous parameter is core frequency and an example of a discrete parameter is the number of compute cores. For all discrete dimensions the step size, which we define as smallest distance between any two x_{id}'s, can vary. We might only be able to increase memory width in 16 bits increments.

B. Fitness Function

Given a parameter setting $\mathbf{x}$, the benchmark $b(\mathbf{x})$ returns a fitness metric vector which constitutes two values: $\mathbf{y}$, the scalar metric of fitness and t, the exit code of the application. Execution time and power consumption are examples of fitness measures. There are be many possible exit codes t, with 0 indicating valid $\mathbf{x}$'s. The designer can choose to extend the benchmark to return additional exit codes depending on the failure cause, such as configurations producing inaccurate results or failing to build.

We distinguish three different types of exit codes. The first type is exit code 0 indicating a valid design. The second type of exit codes indicate configurations that produce results yet fail at least one constraint making them undesirable. The third type of exit codes is used for configurations that fail to produce any results. The region of $\mathcal{X}$ that defines configurations $\mathbf{x}$ that produce y and satisfy all constraints is defined as valid region $\mathcal{V}$, regions with designs failing at least one constraint yet producing y are part of failed region $\mathcal{F}$, and the region with designs failing to produce y is the invalid region $\mathcal{I}$. If $\mathbf{x}_*$ does not produce a valid result, we assign a value that the designer assumes to be the most disadvantageous. Depending on whether we face a minimization/maximization problem,s either a high max_{val} or low min_{val} value will be assigned.

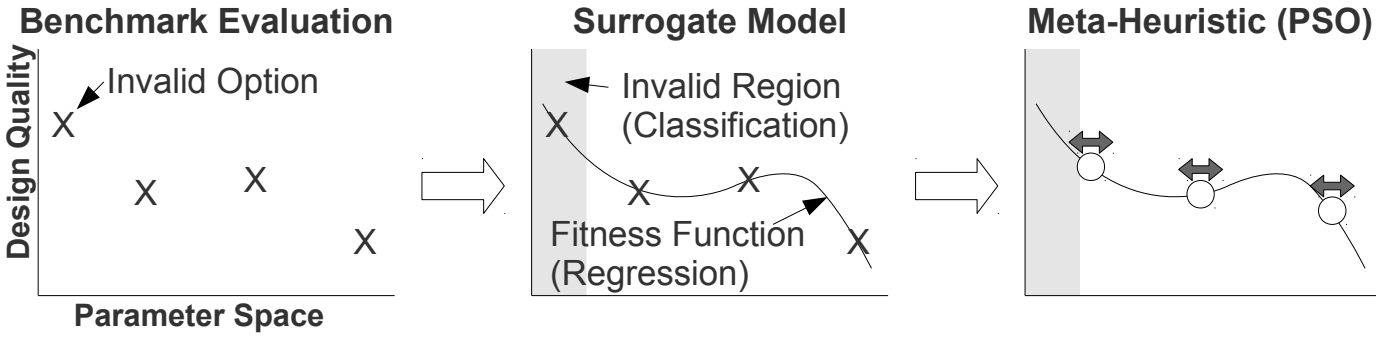

Fig. 1: Benchmark evaluations, surrogate model and model guided search.

$$f(\mathbf{x}) = \begin{cases} y & \mathbf{x} \in \mathcal{V} \\ max_{val} \vee min_{val} & otherwise \end{cases} \quad (4)$$

IV. MLO Surrogate Model

We integrate a Bayesian regressor $\hat{f}$ and a classifier to create a novel surrogate model for a given fitness function f. As illustrated in Fig 1, the problem we face is regression of f over $\mathcal{V}$ and $\mathcal{F}$ as well as classification of $\mathcal{X}$. We make use of Bayesian regressors to access the probability of prediction of $\hat{f}(\mathbf{x}_*)$ of non-examined parameter configurations $\mathbf{x}_*$. We use classifiers to predict exit codes of X_* across $\mathcal{X}$. Regressions are made using the training set obtained from benchmark execution $\mathcal{D}_r$, while classification is done using the training set $\mathcal{D}_c$. We invoke $regressor(\mathcal{D}_r, \mathbf{x}_*)$ for every particle in $\mathbf{x}_*$ to obtain the regression y_* and its probability $p(y_*|\mathbf{x}_*, \mathcal{D}_r)$, which we denote as ρ for simplicity. Class label t_* of particle $\mathbf{x}_*$ is predicted by the classifier $classifier(\mathcal{D}_c, \mathbf{x}_*)$.

Algorithm 1 MLO

```
for x_* ∈ X_* do
    x_*.fit ← f(x_*)   ▷ Initialize with a uniformly randomized set X_*.
end for
repeat
    for x_* ∈ X_* do
        y_*, ρ ← regressor(D_r, x_*)
        t_* ← classifier(D_c, x_*)
        if ρ < min_ρ and t_* = 0 then
            x_*.fit ← y_*
        else
            if t_* = 0 then
                x_*.fit ← f(x_*)
            else
                x_*.fit ← max_val or min_val
            end if
        end if
    end for
    X_* ← Meta(X_*)   ▷ Iteration of the meta-heuristic
until Termination Criteria Satisfied
```

We present our MLO in Algorithm 1. The algorithm's main novelty with respect to surrogate-based algorithms is the integration of a classifier to account for invalid regions of $\mathcal{X}$. We initialize the meta-heuristic of our choice with N particles X_* uniformly randomly scattered across $\mathcal{X}$. Each particle has an associated fitness $\mathbf{x}.fit$ and a position $\mathbf{x}$. For all $\mathbf{x}_*$ predicted to lie in $\mathcal{V}$ we proceed as follows. Whenever ρ returned by the regressor is smaller than the minimum required confidence min_ρ we use the y_*; otherwise we assume the prediction to be inaccurate and evaluate $f(\mathbf{x}_*)$. The meta-heuristic will avoid $\mathcal{I}$ and $\mathcal{F}$ regions as they are both assigned unfavorable max_{val} or min_{val} values. We construct the training sets $\mathcal{D}_c$ and $\mathcal{D}_r$ as described in Algorithm 2. Whenever $b(\mathbf{x}_*)$ is evaluated, $(\mathbf{x}_*, t_*)$ is included within the classifier training set $\mathcal{D}_c$. If exit code is valid ($t_* = 0$), then $(\mathbf{x}_*, y_*)$ is added to $\mathcal{D}_r$.

Although the MLO will converge towards an optimum, it is limited by heuristic search restrictions and as such it cannot guarantee to find the global optimum. Hence, it is crucial to specify the termination criteria. Determining MLO termination criteria is based on the optimization scenario and we present three possibilities where the user:

1) Has a limited compute time budget.
2) Requires only certain design quality.
3) Needs maximum performance, with a large budget.

A user can have a limited compute time budget when optimizing an application and the MLO can terminate once the budget has been reached. For example, we could allocate a number of machines for a 24 hour period. Alternatively, if the user only requires a certain performance, the MLO can be run until a configuration x is found that meets the required performance, and the optimization can be terminated. Lastly, if the MLO is used to maximize performance without a limited compute time budget, the MLO will terminate when the best found solution does not improve during a pre-defined amount of time.

Algorithm 2 $f(\mathbf{x})$

```
t, y ← b(x)
D_c ← (x, t)   ▷ Update the classifier's training set
if t ∈ F or t ∈ V then
    D_r ← (x, y)   ▷ Update the regressor's training set
end if
if t ∈ V then
    return y
else
    return max_val or min_val
end if
```

V. Evaluation

VI. Conclusions and Future Work

References

[1] M. Kurek and W. Luk, "Parametric Reconfigurable Designs with Machine Learning Optimizer," in *FPT*, 2012.

[2] M. Kurek, T. Becker, and W. Luk, "Parametric optimization of reconfigurable designs using machine learning," in *ARC*, 2012.