Automatic Optimization of Reconfigurable Applications with Non-uniform Design Space Exploration Cost

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 enhance MLO by making it aware of non-uniform computation time when varying parameters. This phenomena exists due to caching, non-linear bit-stream generation scalability and benchmark evaluation time. We evaluate our approach using two case studies.**

***Index Terms*—ptimization, surrogate modeling, PSO, GP, SVM, FPGAptimization, surrogate modeling, PSO, GP, SVM, FPGAo**

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. We now enhance the previous using our knowledge of the reconfigurable hardware design making MLO aware of cross parameter non-uniformity of the design computation time. The contributions of this paper are:

- A mathematical characterization of the parameter space for reconfigurable designs with non-uniform cost design spaces. The non-uniform cost is a direct result of circuit caching, non-linear bit-stream and benchmark evaluation time. (Section III).
- Adaptation of MLO to the non-uniform design spaces. (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

When developing reconfigurable applications, 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.

Field programmable gate arrays (FPGAs) allow designs that are customized to the requirement of the application. Reconfiguration is an additional benefit which allows the designer to modify designs at run time, potentially increasing 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.

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 Particle Swarm Optimization (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 which constitutes two values: 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.

C. Sampling parameter space

Previously we assumed that the cost $\mathfrak{C}$ of evaluation of a fitness function f for any x is constant, unfortunately this is not true in case of reconfigurable computing. Due to mapping and routing benchmark generation cost will increase non-linearly when increasing number of computational core, the higher the utilization the more difficult the problem becomes. In hardware description language (HDL) based systems features like caching of circuit subcomponents can change, interfering with the cost function $\mathfrak{C}$ of a given design. This implies that given three different parameter settings x_1, x_2, x_3 the order of f evaluations will change the cost $c($ of evaluation of each setting and possibly total setting. We define c in terms of x and t as having the following properties.

$$\mathfrak{C}(\mathbf{x}, s) \not= \mathfrak{C}(\mathbf{x}, s) \tag{1}$$

$$\mathfrak{C}(\mathbf{x_n}|\mathbf{x_{n-1}}...) \not= \mathfrak{C}(\mathbf{x}, s) \tag{2}$$

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

```
1: for $\mathbf{x}_* \in X_*$ do
2:     $\mathbf{x}_*.fit \leftarrow f(\mathbf{x}_*)$    ▷ Initialize with a uniformly randomized set $X_*$.
3: end for
4: repeat
5:     for $\mathbf{x}_* \in X_*$ do
6:         $y_*, \rho \leftarrow regressor(\mathcal{D}_r, \mathbf{x}_*)$
7:         $t_* \leftarrow classifier(\mathcal{D}_c, \mathbf{x}_*)$
8:         if $\rho < min_\rho$ and $t_* = 0$ then
9:             $\mathbf{x}_*.fit \leftarrow y_*$
10:        else
11:            if $t_* = 0$ then
12:                $\mathbf{x}_*.fit \leftarrow f(\mathbf{x}_*)$
13:            else
14:                $\mathbf{x}_*.fit \leftarrow \max_{val}$ or $\min_{val}$
15:            end if
16:        end if
17:    end for
18:    $X_* \leftarrow Meta(X_*)$    ▷ Iteration of the meta-heuristic
19: 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 $\mathcal{X}$.

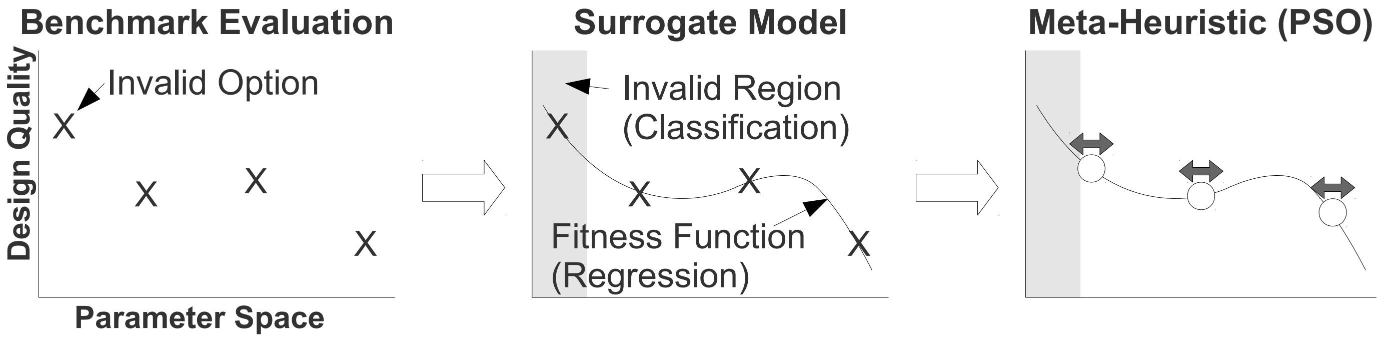

Fig. 1: Benchmark evaluations, surrogate model and model guided search.

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.

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.

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

```
1: $t, y \leftarrow b(\mathbf{x})$
2: $\mathcal{D}_c \leftarrow (\mathbf{x}, t)$                ▷ Update the classifier's training set
3: if $t \in \mathcal{F}$ or $t \in \mathcal{V}$ then
4:     $\mathcal{D}_r \leftarrow (\mathbf{x}, y)$            ▷ Update the regressor's training set
5: end if
6: if $t \in \mathcal{V}$ then
7:     return $y$
8: else
9:     return $\max_{val}$ or $\min_{val}$
10: end if
```