

Predicting Best Design Trade-offs: A Case Study in Processor Customization

Marcela Zuluaga
ETH Zürich

Edwin V. Bonilla
NICTA & Australian National University

Nigel Topham
University of Edinburgh

Abstract—Given the high level description of a task, many different hardware modules may be generated while meeting its behavioral requirements. The characteristics of the generated hardware can be tailored to favor energy efficiency, performance, accuracy or die area. The inherent trade-offs between such metrics need to be explored in order to choose a solution that meets design and cost expectations. We address the generic problem of automatically deriving a hardware implementation from a high-level task description. In this paper we present a novel technique that exploits previously explored implementation design spaces in order to find optimal trade-offs for new high-level descriptions. This technique is generalizable to a range of high-level synthesis problems in which trade-offs can be exposed by changing the parameters of the hardware generation tool. Our strategy, based upon machine learning techniques, models the impact of the parameterization of the tool on the target objectives, given the characteristics of the input. Thus, a predictor is able to suggest a subset of parameters that are likely to lead to optimal hardware implementations. The proposed method is evaluated on a resource sharing problem which is typical in high level synthesis, where the trade-offs between area and performance need to be explored. In this case study, we show that the technique can reduce by two orders of magnitude the number of design points that need to be explored in order to find the Pareto optimal solutions.

I. INTRODUCTION

The increasing complexity of very large-scale integration (VLSI) designs makes the generation of fully optimized systems a big challenge. As Moore's law continues to hold true, more resources can be placed in a single chip and larger system modules are moved from software to dedicated hardware in order to improve performance. Moreover, the rapid proliferation of electronic devices puts pressure on industry to offer not only high performance, multi-purpose devices but also extended battery lives, low silicon cost, and new generations of products in short periods of time. Thus, embedded systems designers have to customize their solutions in order to meet strict requirements: performance, cost, power consumption and time-to-market. As design trade-offs become more complex, and large design spaces define competing objectives, the use of Computer Aided Design (CAD) techniques becomes essential in order to achieve optimality. Similarly, the use of higher levels of abstractions to define systems, in which designers can comfortably reason, requires advanced behavioral synthesis methods that can automatically generate gate-level specifications. Given the high level description of a task, many different hardware modules may be generated in

order to meet its behavioral requirements. The characteristics of the generated Register-Transfer Level (RTL) description can be tailored to favor energy efficiency, performance, accuracy or area. The inherent trade-offs between such metrics need to be explored in order to choose a solution that meets design and cost expectations.

In this paper, we present a novel technique to quickly discover the design space of solutions, of a subset of high level synthesis problems, by exploring only parameter solutions that are likely to lead to optimal trade-offs. This technique is generalizable to a range of high-level synthesis problems in which trade-offs can be exposed by parameterizing the hardware generation tool. Based on previously-explored design spaces, predictive modeling is used to generate the parameterization that the hardware generation process requires in order to directly find the optimal trade-off solutions, given the characteristics of a new input. Our strategy uses machine learning techniques to capture the impact of the parameterization of the tool on the target metrics, given the characteristics the inputs. Thus, a predictor is able to suggest a subset of parameters that are likely to lead to optimal hardware implementations, and therefore the need to exhaustively explore a large parameter space is removed. This proves to be of particular value when the evaluation of each design point is expensive, or when local explorations need to be carried out repetitively as explorations at higher levels of the design take place.

The proposed technique is evaluated on a resource sharing problem, which is typical in high level synthesis, where the trade-offs between area and performance need to be explored. With this application, we show in detail how the technique is conditioned to a particular process. Moreover, we demonstrate that in comparison with an exhaustive exploration of the design space, the predictor reduced by two orders of magnitude the number of executions of the resource-sharing process that are required in order to find the optimal trade-offs.

II. TRADE-OFFS IN HARDWARE SYNTHESIS

Opportunities to exploit design trade-offs can be found at all levels of system design. The primary design trade-off in hardware generation processes is typically between area and performance. As more resources are allocated, execution time and/or throughput can be improved. These trade-offs can be exposed by parameterizing the hardware generation process (or algorithm), where parameters represent design choices that are taken along its execution in order to generate a single solution.

Typical parameters used in hardware generation processes are: number of execution units, amount of resources to share, memories to allocate, data types and sizes, algorithm choice, pipeline stages, unrolling factors, etc.

In CAD for hardware synthesis, there are two important types of processes in which this parameterization can be applied. Some processes convert an abstract specification of a task into an RTL description, while others convert an RTL description into a gate level netlist.

A particular solution, generated with a fixed set of parameters, can be evaluated with the objectives (or *metrics*) that better suit the design goals. Commonly used metrics are: execution time, energy consumption and cost. Solutions might be optimal or suboptimal in comparison to others, in terms of the target metrics. Every combination of parameter values affects differently each input, and each parameter has a unique impact when evaluating trade-offs in the design space. Therefore, the parameter space has to be explored exhaustively for every new input to the process. After this exploration is completed, the Pareto-optimal solutions can be extracted. A solution is said to be Pareto-optimal when no other solution is better in all of the target metrics.

III. PREDICTING BEST DESIGN TRADE-OFFS

As exhaustive processes are extremely time-consuming, machine-learning approaches can be used to predict the combination of parameters that results in Pareto-optimal solutions, based on previously explored design spaces. The ideal scenario being that the hardware generation process is executed only as many times as Pareto points can be found in the design space.

Machine learning techniques can detect patterns such as that similar inputs respond similarly to some parameter configurations. These similarities across inputs can be inferred from a number of characteristics (or *features*) extracted from them. In the following sections we show that a model learned on training data from previous explorations can be used to speed up the exploration of a design space created by a new input.

A. Construction of the Predictor

A predictive model aims at capturing the patterns of the parameter values that generate a Pareto-optimal point in a given process. Therefore, as the goal of the model is to find all optimal trade-offs in the design space, different regions of the parameter space need to be explored. Training samples are extracted from previously fully-explored spaces by coupling input features with each set of parameter values that led to a Pareto solution. Figure 1 illustrates the general process of generating predictions (parameter values) for an unseen input. We describe four high-level steps:

- 1) **Selection of Training Inputs:** In this step we select a subset of the training inputs (from our previously explored design spaces) that are closest to the unseen input, for which we are interested in making predictions.
- 2) **Clustering of Parameter Values:** Those parameter values associated with the selected training inputs are grouped into clusters so that they partition the parameter

space according to their values. Parameter sets that belong to the same cluster are expected to have similar values, and parameter sets that belong to different clusters are expected to have different characteristics.

- 3) **Extraction of Distributions over Parameter Values:** In order to model the similarities (or *patterns*) of parameters within each cluster we extract a distribution over the parameter values for each cluster.
- 4) **Generation of Predicted Parameter Values:** We then draw samples from these distributions, i.e parameter values, that will be used as the predicted parameters for the unseen input, and for which we wish to evaluate their performance.

The exploration in every cluster is expected to target different regions of the Pareto curve, and is based on the parameters values taken from the training cases most similar to the unseen input. We now introduce some notation and describe in more detail each of the steps mentioned above.

B. Problem Definition

Let us denote the D -dimensional vector \mathbf{f} as a vector of D features associated with an input, and the $N_{\text{train}} \times D$ matrix $\mathbf{F}_{\text{train}}$ as the matrix containing the features for all N_{train} training inputs. Additionally, let us denote $\mathbf{f}_{\text{unseen}}$ as the vector of features for the unseen input, which we will also refer to as the test input. Similarly, let $\mathbf{P}_{\text{train}}$ be the $N_{\text{train}} \times N_{\text{param}}$ matrix of parameter values for all training inputs, where N_{param} is the number of parameters. Our goal is to predict a set of parameter values $\mathbf{P}_{\text{unseen}}$ for the unseen input that have best performance in terms of Pareto optimality.

C. Feature Extraction and Selection of Training Inputs

The selection of features is important to any machine learning-based technique as these have a direct impact on how the model differentiates between inputs and how it can gauge similarities. Features should describe the key characteristics of the input that may affect the response when changing the value of the parameters. Given an unseen set of features $\mathbf{f}_{\text{unseen}}$, along with the set of training features $\mathbf{F}_{\text{train}}$, we compute the Euclidean distances from $\mathbf{f}_{\text{unseen}}$ to each training feature vector in $\mathbf{F}_{\text{train}}$. We select the k training cases closest to the new case by choosing those training inputs with the smallest Euclidean distances to the unseen case. We refer to the features of these k selected training inputs as $\mathbf{F}_{k\text{-train}}$ and to the associated parameters as $\mathbf{P}_{k\text{-train}}$.

D. Clustering of Parameter Values

Our goal here is to group the parameter values $\mathbf{P}_{k\text{-train}}$ obtained in the previous step into c different clusters. Many clustering algorithms have been proposed in the machine learning literature, with *k-means* being one of the most commonly used in practical applications. Here we use a Gaussian Mixture Model (GMM, see e.g. [1], Ch. 9), which is known to be a probabilistic generalization of k-means, with the advantage that each cluster is associated with a Gaussian distribution. Points in $\mathbf{P}_{k\text{-train}}$ are clustered into c groups, thus forming

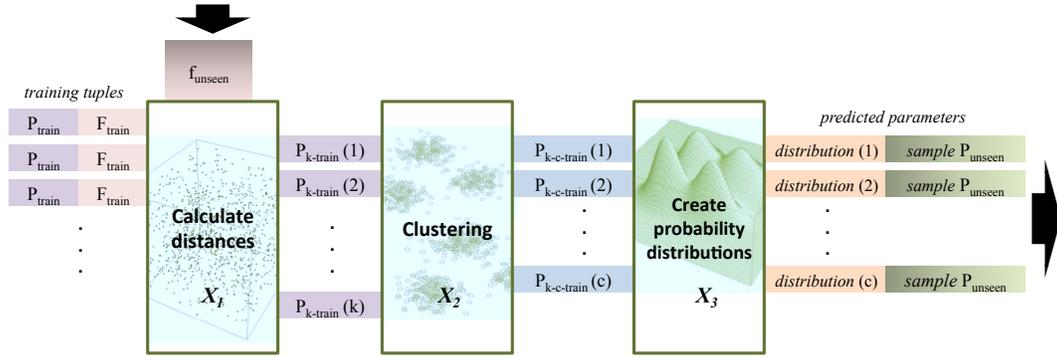


Fig. 1. Graphical representation of the internal processes that must take place in order to predict the Pareto parameter-values for a new input. We are given the features of the new input f_{unseen} and a set of feature-parameter training pairs $(\mathbf{F}_{\text{train}}, \mathbf{P}_{\text{train}})$ obtained from previously explored design spaces. X_1 returns the set of parameters associated to the k training inputs that are closest to f_{unseen} . These parameters are represented by the matrix $\mathbf{P}_{k\text{-train}}$. X_2 returns the elements of $\mathbf{P}_{k\text{-train}}$ clustered in c groups ($\mathbf{P}_{k\text{-c-train}}$). X_3 returns the probability distribution of each cluster. Subsequently, parameter settings $\mathbf{P}_{\text{unseen}}$ are obtained from sampling each probability distribution.

regions in the parameter space that can be independently explored. Therefore, every vector in $\mathbf{P}_{k\text{-train}}$ is categorized into one of the C clusters and then referred to as $\mathbf{P}_{k\text{-c-train}}$. We fit a Gaussian Mixture Model using the expectation-maximization algorithm (EM, [2]).

E. Extraction of Distributions over Parameters

As mentioned above, one of the advantages of using GMMs for clustering is that we have a probabilistic model from which we can extract the distribution corresponding to each cluster straightforwardly. In particular, at the end of a GMM run, we obtain a Gaussian distribution for each cluster, which we will use to draw parameter values as candidates to be a Pareto optimal point on our unseen (test) input.

F. Generation of Predicted Parameter Values

Finally, we use the set of Gaussians from the previous step in order to generate our predictions $\mathbf{P}_{\text{unseen}}$. In order to achieve this we draw samples from these distributions one-by-one in a round-robin fashion. $\mathbf{P}_{\text{unseen}}$ contains the values of the parameters that will be used in the hardware generation process.

G. Number of Clusters c and Neighbors k

Having defined the procedure to generate a prediction, the values assigned to k (number of neighbors) and c (number of clusters) are key to the finalization of the model for future use. Every pair of values (k, c) creates a new model that will predict differently. For this reason, several configurations should be evaluated in order to choose these values. It is impractical to evaluate all of the possible configurations for the pair (k, c) . Therefore, a reasonable list of possible values can be taken for consideration. Given these, cross-validation can be used for model selection [3]. In short, with cross-validation we learn the best combination (k, c) from the training data.

H. Performance Measure

In practice, the model will suggest values of parameters in $\mathbf{P}_{\text{unseen}}$ to be used in the hardware generation process given

a set of features extracted from the input. It will attempt to suggest points in the parameter space that will lead to a Pareto point in the available design space.

The performance of the model can be a measure of how many points need to be suggested in order to ensure that all of the possible optimal points are found. Such points compose the Pareto-curve that can be found during an exhaustive exploration of the parameter space and will be referred to as the *true Pareto-curve*. Therefore, the number of points that are needed in order to find the true Pareto-curve will be used as a metric and will be referred to as R . Consequently, for future explorations, the hardware generation process will have to be executed only R times. R will be determined with the experiments that evaluate the model.

Additionally, there is a need to measure, for a given R , the similarity between the true Pareto-curve and the curve that has been found so far. The latter is referred to as *known Pareto-curve*. A metric d_{pp} is created for this purpose and calculated by summing up the distances from each point of the true Pareto-curve to the closest point of the known Pareto-curve after R explorations. In order to normalize the distance-based metric, this sum is divided by the length of the true Pareto-curve. Figure 2 indicates how these distances are found.

IV. CASE STUDY: RESOURCE SHARING IN PROCESSOR CUSTOMIZATION

When new instructions (or *custom instructions*) are added to the Instruction-Set Architecture (ISA) of a processor, new execution units need to be added to its hardware implementation. Thus, while more instructions are added to a processor, the area can grow to the point that static power becomes critical. An alternative to reduce the energy consumption and die area of a customized processor is to share resources amongst the custom instruction hardware modules, creating instead a module that can be configured to execute several instructions. Resource sharing is then a process that generates hardware modules that are attached to the execution stage of a processor. The input to this process is a set of Data-Flow

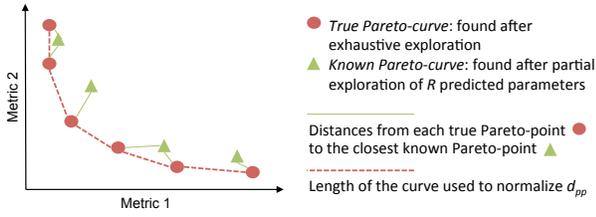


Fig. 2. Visualization of the components needed to calculate d_{pp} , which measures the distance between the known Pareto-curve and the true-Pareto curve.

Graphs (DFGs), where each DFG represents the operations performed by a custom instruction. These graph representations are extracted from the target application code. Resource sharing involves merging the DFGs of two or more custom instructions which contain a similar subgraph. Therefore, the output of the process is a set of DFGs composed of at most as many DFGs as the input set. Aggressive merging could considerably increase the latency of the instructions. In order to control the effect of merging on the execution latency of the custom instructions, [4] proposed a heuristic that parameterizes a resource-sharing process in order to explore the design space of implementation alternatives. Implementation alternatives represent trade-offs between custom instruction execution latency and area. Thus, solutions that aggressively share resources amongst the custom instructions might present the highest data-path latency. The merging process proposed in [4] is primarily parameterized by three threshold values: α_T , β_T and θ_T . These parameters constrain the impact of merging on the custom instruction at different stages of the resource sharing algorithm. When $\alpha_T = 1$, $\beta_T = 1$ and $\theta_T = 1$, the process maximizes sharing amongst the input custom instructions in order to obtain the minimum-area solution. On the other hand, when $\alpha_T = 0$, $\beta_T = 0$ and $\theta_T = 0$, no resource sharing is performed and the minimum instruction data-path latency is obtained. The design space of intermediate solutions that represent trade-offs between area savings and instruction data-path latency can be explored by varying α_T , β_T and θ_T values in the range [0,1]. The merging process can be further parameterized in order to enable the creation of multi-function operators such as adder-subtractors, through the binary parameter *MultiOp*, and/or to enable the compression of custom instruction operators that allow synthesis optimizations to create modules such as multiply-adders and carry-save adders, through the binary parameter *grouping*.

Every solution found by the parameterized resource-sharing process is characterized by the metrics: total area and weighted average critical path of the custom instructions. The critical path of the custom instructions is weighted by their execution frequency. The features that were chosen to quantitatively describe every input (or set of DFGs) are: number of DFGs (or number of custom instructions to implement); and standard deviation, 1st quartile, 2nd quartile and 3rd quartile of the set of critical paths of the graphs, weighted with their corresponding

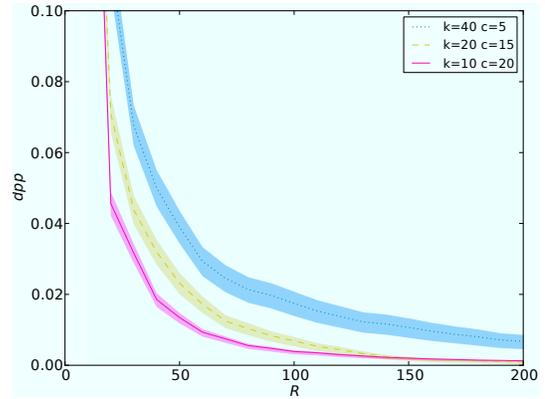


Fig. 3. Average results for 3 of the (k, c) configurations that were tested. Other configurations are not included for better readability of the plot. Shaded areas correspond to the 95% confidence interval of the mean values.

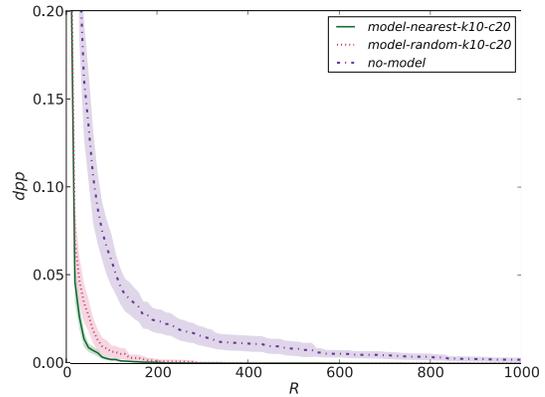


Fig. 4. Experimental evaluation of *model-nearest-k10-c20*. Results obtained across 10 experiments are averaged. Shaded areas correspond to the 95% confidence interval of the mean values.

execution frequency.

A. Generating the Training Data

A total of 95 training cases were obtained from 56 benchmarks taken from the UTDSP [5] and SNU-RT [6] benchmark suites. Custom instruction identification was performed on each benchmark using an implementation of *ISEGEN* [7]. The smallest training case contains 5 custom instructions, and the largest training case contains 26. The design space of resource-sharing solutions was fully explored for every training case. The exploration of this space was done by executing the resource-sharing algorithm iteratively with different values of the parameters α_T , β_T and θ_T , varying from 0 to 1 in steps of 0.05 (resulting in 21 different values). In turn, every set of values for α_T , β_T and θ_T , was run with the four combinations of values for the parameters *multiOp* and *grouping*. This resulted in a design space of 37,044 points. After this exploration, training input-output pairs are composed by coupling the features extracted from the training case and the parameter configurations associated with each Pareto point.

B. Experimental Evaluation of the Model

Values in $k = \{10, 20, 40\}$ and in $c = \{5, 10, 15, 20\}$ were considered. Then, a different model was created from every possible combination (k, c) . Experiments were performed in two rounds of cross-validation. In the first round, k and c are chosen while in the second round, the definite (k, c) configuration is tested. For the sake of evaluating the final configuration on data that has not been used to choose the values of k and c , the training set is partitioned in two. 90% of the training set is used to evaluate the different (k, c) configurations with a *leave-one-out cross-validation* strategy [3]. The 10% that is not used for this first round of cross-validation is used to test a model that takes the definite (k, c) configuration and that is trained with the above-mentioned 90%. This process is repeated with every 10% of the initial set of training cases. Hence, on every round, 90% is used to test every configuration (k, c) . In order to demonstrate the generalization of our technique by evaluating it on completely unseen cases, the parameters k and c are tuned using only the training set and never the test set.

1) *Choosing k and c Values:* First, the 10 experiments of leave-one-out cross-validation are performed. These correspond to every 10% left out. In every experiment, each configuration (k, c) is evaluated. Figure 3 shows average values of d_{pp} obtained with three of the (k, c) pairs tested over the 10 experiments. The results of all (k, c) configurations are not shown for better readability of the plot. The configuration with $k = 10$ and $c = 20$ showed smaller d_{pp} values for most values of R . Thus, the number of neighbors k was fixed to 10 and the number of clusters c was fixed to 20. The model with this configurations will be referred to as *model-nearest-k10-c20*.

2) *Evaluating the Model:* The second round of cross validation takes place by testing *model-nearest-k10-c20* on every 10%, while training every time on the remaining 90%. Figure 4 shows the averaged mean values of d_{pp} and their 95% confidence intervals, over the 10 experiments, for every R , in increments of 10. For comparison, we also show the results obtained when using a modified model, referred to as *model-random-k10-c20*, which makes its predictions based on 20 random training samples instead of the 20 nearest ones. This comparison allows an evaluation of the effectiveness of the features in extracting the relevant characteristics of the set. Additionally, these results are compared with a random exploration of the parameter space. This is, instead of using a model to suggest parameter combinations, these are generated as a vector composed of random numbers uniformly distributed in the range allowed by the parameters.

When using *model-nearest-k10-c20*, at approximately $R = 200$, d_{pp} stabilizes to its smallest value. This means that after 200 parameter configurations suggested by the model, the majority of the inputs find the true Pareto-curve in the resource-sharing design-space. When the model is used for future predictions, it will generate 200 parameter configurations to parameterize the resource-sharing algorithm. Thus,

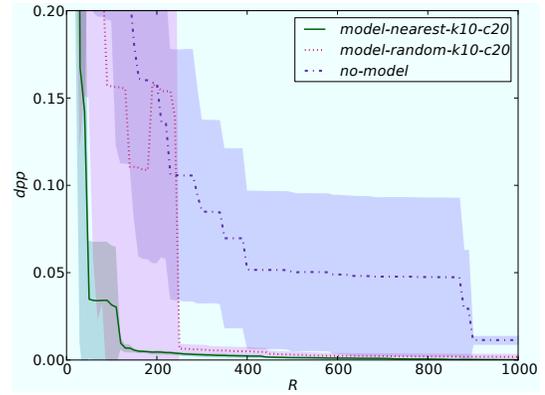


Fig. 5. Results of predicting parameter values for an input set generated from the Coremark application. Shaded areas correspond to the 95% confidence interval of the mean values.

the Pareto curve found after these 200 executions will be the same or very close to the Pareto curve that would be found if the algorithm was executed exhaustively for all possible parameter configurations. At this point, the parameters of the model: $k = 10$, $c = 20$, and $R = 200$ have been determined.

C. Practical Usage of the Model

In this section, *model-nearest-k10-c20* is evaluated with CoreMark [8], an application that was not used to generate the training sets. 40 custom instructions that were extracted from this application constitute the input set for the predictor.

The results of this experiment are shown in Figure 5. The figure shows the results of predicting parameter values using *model-nearest-k10-c20* and *model-random-k10-c20*. These predictions are contrasted with *no-model* or random prediction. The value of d_{pp} was measured at every 10th execution of the resource-sharing algorithm (every 10th parameter configuration explored). As seen in the figure, after 200 parameter configurations tested ($R = 200$), d_{pp} values were low enough to conclude that the majority of the area-speedup trade-offs in the resource-sharing design-space had been found. This results also show that the model *model-nearest-k10-c20* performs remarkably better than *model-random-k10-c20* and *no-model*.

The actual running times of these explorations were measured. Experiments were performed on a workstation equipped with 4 Xeon processors running at 3 GHz, and 4 GB of RAM. An exhaustive exploration of 37,044 points in the design space took 15,757 minutes to complete, while the exploration of the 200 parameter configurations suggested by the model took 111 minutes. Thus, the predictor achieved a speedup of $141\times$ in running time over a exhaustive exploration. This confirms the predictive power of the model, and demonstrates important time savings in the exploration of new design spaces.

V. RELATED WORK

Multiobjective optimization has been a hot topic of research for several decades. Multiple approaches for approximating the Pareto surface of a multi-dimensional objective space have been proposed. Evolutionary algorithms, being one of the

most popular of these approaches, have proven to be robust and powerful search mechanisms for tackling the exploration of highly complex design spaces. These algorithms aim at evolving a population to converge to Pareto solutions by emulating natural evolution, supported by concepts such as fitness, elitism, and mutation. The multi-objective nature of the problem raises several challenges to these approaches. Recent works on this topic aim at overcoming these challenges; such as maintaining a diverse population, and defining appropriate fitness functions to suit the multiple objectives [9], [10].

In the context of CAD, [11] proposed an unsupervised Monte-Carlo exploration, together with a statistical analysis that allows capturing key characteristics of the design space in high level synthesis processes. [12] proposed a genetic algorithm to solve a problem of digital circuit optimization through the development of specific structures and procedures. [13] suggested a heuristic based on Pareto simulated annealing to explore the design trade-offs generated by the parameterization of a combined design-space of architectural parameters and source-program transformations.

Optimization techniques such as evolutionary algorithms, simulated annealing, and Monte-Carlo methods are well suited for design spaces in which finding the real Pareto front by exhaustive exploration is computationally infeasible. In this paper, we target design spaces for which it is feasible to find a good approximation of the Pareto front in order to create training samples. Thus, we are able to suggest a set of good parameter configurations by only extracting the characteristics of the object to be transformed by the underlying process.

In the context of processor customization, the importance of resource sharing in application-specific unit synthesis has been stressed by several researchers [14], [15]. [4] showed that resource sharing creates a multi-objective design space, since aggressively sharing resources leads to large custom-instruction data-path latencies. Moreover, it has been shown that there is a design space of trade-offs between area savings and instruction latency that designers can explore. An interaction with the exploration at the instruction selection level has been tackled in [16]. However, as [16] proposes an iterative search, a large number of explorations at the implementation level need to be carried out. Thus, an exhaustive exploration of the resource-sharing design-space does not permit the scalability of the selection process.

VI. CONCLUSIONS

This paper has presented a novel, yet highly practical, predictive model that can be used to quickly find the optimal implementation trade-offs across a range of hardware synthesis tasks where parameterization exposes a large set of design points. This is a relatively common scenario in electronic design automation, from high-level synthesis, through logic synthesis and even at the physical implementation level.

A concrete application of the proposed technique has been presented in the context of processor customization. In this application, for every new input to the process, a large design space of solutions, each with different levels of resource

sharing, is available. Exhaustive search will always find the optimum solution, but is prohibitively expensive in practice.

In this case study, the predictive model is shown to reduce by two orders of magnitude the number of executions of the resource-sharing algorithm that are required in order to find Pareto-optimal solutions, compared to an exhaustive exploration of the design space.

Thus, it is shown that learning techniques that extract patterns from previously-explored spaces can be used effectively in order to solve complex problems that create large design spaces but that are likely to give rise to more efficient designs.

ACKNOWLEDGEMENTS

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program. MZ and NT would like to thank EPSRC for their financial support under grant EP/D50399X/1.

REFERENCES

- [1] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, August 2006.
- [2] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum Likelihood from Incomplete Data via the EM Algorithm," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 39, no. 1, pp. 1–38, 1977.
- [3] R. Kohavi, "A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection," in *IJCAI'95*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995, pp. 1137–1143.
- [4] M. Zuluaga and N. Topham, "Design-Space Exploration of Resource-Sharing Solutions for Custom Instruction Set Extensions," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 28, no. 12, pp. 1788–1801, 2009.
- [5] C. Lee and M. Stoodley, "UTDSP benchmark suite," 1992, <http://www.eecg.toronto.edu/corinna/DSP/infrastructure.html>, 2010.
- [6] "SNU-RT real time benchmarks," 2010, <http://archi.snu.ac.kr/realtime/benchmark2010>.
- [7] P. Biswas, S. Banerjee, N. Dutt, L. Pozzi, and P. Ienne, "ISEGEN: an Iterative Improvement-Based ISE Generation Technique for Fast Customization of Processors," *IEEE Trans. VLSI Syst.*, vol. 14, no. 7, 2006.
- [8] "CoreMark," 2010, <http://www.coremark.org/home.php>.
- [9] S. Knzli, L. Thiele, and E. Zitzler, "Modular Design Space Exploration Framework for Embedded Systems," *IEEE Proceedings Computers & Digital Techniques*, vol. 152, no. 2, pp. 183–192, 2005.
- [10] C. Coello, G. B. Lamont, and D. Veldhuizen, *Evolutionary Algorithms for Solving Multi-Objective Problems (Genetic and Evolutionary Computation)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.
- [11] B. D., A. Bogliolo, and L. Benini, "Statistical Design Space Exploration for Application-Specific Unit Synthesis," in *DAC'01*. ACM Press, 2001, pp. 641–646.
- [12] Z. Salcic, G. Coghill, and B. Maunder, "A Genetic Algorithm High-Level Optimizer for Complex Datapath and Data-flow Digital Systems," *Applied Soft Computing*, vol. 7, no. 3, pp. 979 – 994, 2007.
- [13] G. Agosta, G. Palermo, and C. Silvano, "Multi-objective Co-Exploration of Source Code Transformations and Design Space Architectures for Low-Power Embedded Systems," in *SAC'04*. New York, NY, USA: ACM, 2004, pp. 891–896.
- [14] P. Brisk, A. Kaplan, and M. Sarrafzadeh, "Area-Efficient Instruction Set Synthesis for Reconfigurable System-on-Chip Designs," in *DAC'04*. New York, NY, USA: ACM Press, 2004, pp. 395–400.
- [15] N. Moreano, E. C. d. S. Borin, and G. Araujo, "Efficient Datapath Merging for Partially Reconfigurable Architectures," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 24, pp. 969 – 980, Jul. 2005.
- [16] M. Zuluaga and N. Topham, "Exploring the Unified Design-space of Custom-instruction Selection and Resource Sharing," in *SAMOS X*, Jul. 2010.