Reducing Compilation Effort in Commercial FPGA Emulation Systems Using Machine Learning

Anthony Agnesina[1], Etienne Lepercq[2], Jose Escobedo[2], and Sung Kyu Lim[1]
[1]School of ECE, Georgia Institute of Technology, Atlanta, GA
[2]Synopsys Inc., Mountain View, CA
agnesina@gatech.edu

***Abstract*—This paper presents a machine learning (ML) framework to improve the use of computing resources in the FPGA compilation step of a commercial FPGA-based logic emulation flow. Our ML models enable highly accurate predictability of the final P&R design qualities, runtime, and optimal mapping parameters. We identify key compilation features that may require aggressive compilation efforts using our ML models. Experiments based on our large-scale database from an industry's emulation system show that our ML models help reduce the total number of jobs required for a given netlist by 33%. Moreover, our job scheduling algorithm based on our ML model reduces the overall time to completion of concurrent compilation runs by 24%. In addition, we propose a new method to compute "recommendations" from our ML model, in order to perform re-partitioning of difficult partitions. Tested on a large-scale industry SoC design, our recommendation flow provides additional 15% compile time savings for the entire SoC.**

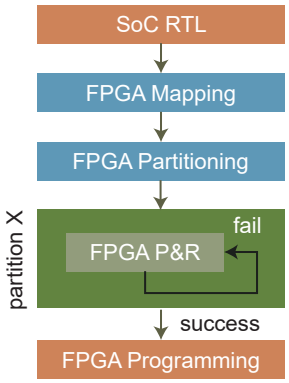

Fig. 1: Our multi-FPGA emulation scheme with FPGA recompilation.

I. Introduction

Modern System on Chip (SoC) designs are often larger and more complex than can be competitively tested under traditional hardware/software co-validation methods. They require billions of cycles of execution, which takes too long to simulate in software. Physical emulation using commercial FPGAs can overcome the time constraints of software emulation of an ASIC of up to a billion gates.

To achieve successful mapping of large ASIC designs, an emulator integrates many hundreds of FPGAs. Commercial FPGAs can provide larger capacity and faster runtime performance (up to 5MHz) compared with custom FPGAs or special-purpose custom logic processor-based architectures. However, these FPGAs do not befit the very high pin-to-gate ratio requirements of logic emulation systems [1]. Therefore, they often suffer from a time-consuming Place and Route (P&R) step that can quickly become the most dominating part of the entire implementation time [2]. As a new compilation run of hundreds of FPGAs might be needed for each design update, a compile time of multiple hours each is crippling.

The use of machine learning (ML) is already benefiting the semiconductor industry, with applications in formal verification and physical design [3] (e.g. yield modeling and predicting congestion hotspots). Our research suggests that ML can as well expedite the time-consuming P&R physical emulation step for FPGAs. Recently, ML has been employed to improve wirelength, delay or power of FPGA P&R solutions using Design Space Exploration of CAD tool parameters [4], [5], [6]. In [7], the authors show it is possible to predict the best Quality-of-Results (QoR) placement flow among a reduced set of candidate flows. However, none of these studies focus on important issues related to compile time, nor have been employed to predict compilation success of very high utilization designs (e.g. up to 75% lookup table (LUT) usage). Indeed, the basis of their exploration targets small traditional benchmarks or small FPGAs, which is far from the reality of crowded and complex consumer designs found in SoC emulation. The key contributions of this paper are as follows:

- We build a complete ML data pipeline framework, allowing for the extraction of numerous predictors.
- Using these predictors and our large-scale commercial FPGA compilation database, we build models delivering high predictability of P&R design qualities, runtime, and optimal mapping parameters of complex designs.
- We show how—by predicting P&R compilation results—we effectively improve the compile time and hardware cost of the P&R step of the emulation process.
- Using our ML model, we demonstrate how our "design recommendations" improve the quality of the partitioning, resulting in overall faster P&R steps.

II. Machine Learning Infrastructure

This work is intended to improve the compilation flow of multi-FPGA-based emulation systems, whose main steps are shown in Figure 1. A given SoC RTL is first translated into circuit representation. Next, the resulting netlist is partitioned across multiple FPGAs using a multilevel hierarchical ap-

This material is based upon work supported by the National Science Foundation under Grant No. CNS 16-24731 and the industry members of the Center for Advanced Electronics in Machine Learning.

978-1-7281-2350-9/19/$31.00 ©2019 IEEE

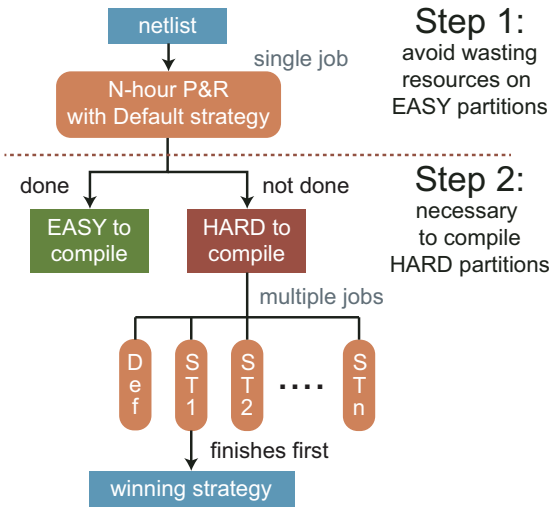

Fig. 2: Our two-step FPGA P&R flow. EASY netlist finishes in Step 1, while HARD continues with the default run, along with new jobs added, into Step 2.

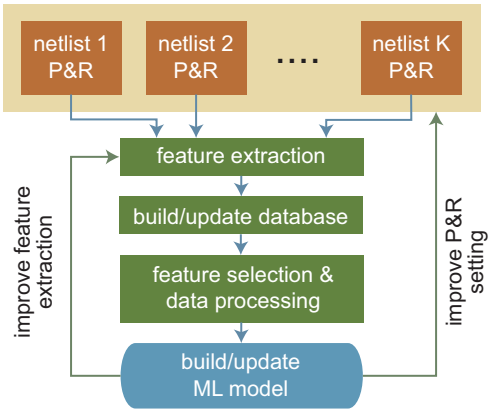

Fig. 3: Our machine learning framework. We update our database and ML models upon new compilation data being added.

proach. As simultaneous objectives need to be optimized during partitioning (e.g. hop counts, cut-sizes, maximum FPGA utilization, etc.), it is common that the required partitioning quality cannot be met without user input.

A. Target FPGA P&R Flow

After these steps, each individual design partition has to be placed and routed within each FPGA using an EDA or FPGA vendor software – Xilinx Vivado in our case. To perform P&R for a given netlist or partition, we either run multiple parallel explorations to find the best P&R solution, or launch Vivado with a Default strategy ($\equiv$ default settings) first. As the server grid used for compilation is occupied by multiple projects in parallel, where each project requires hundreds of individual compiles, limited machine resources can handle these P&R jobs. Thus, it is critical to launch as few jobs as possible for the given netlist. Hence, the Default strategy is initiated first as a standard, as shown in Figure 2.

If the Default run fails or does not finish in N hours ($N = 5$ hours, our "wall-time"), the compiler launches a set of additional P&R jobs in parallel. When one of the jobs terminates successfully, all the running tasks for this FPGA are aborted, and the pending tasks are cancelled. Here, each job implements a different strategy, i.e. a combination of Vivado parameters. In a traditional flow, the strategies selected are not design-related, but mostly dependent on the architecture of the target FPGA. The particularities of a given design are not taken into account, but the strategies that are launched are those that have worked with most success in the past. The "best" knob parameters of the P&R engine truly depend on many design-related factors, more than just the target FPGA family. In this paper, we will prove that these factors can be reduced to a small set of key features. If the P&R of any partition still fails despite using all these strategies, re-synthesis or re-partitioning of the complete design is necessary. Dealing with such tasks requires an engineer in the loop and involves iterating through the entire design cycle, which is time and effort-consuming.

It is therefore highly useful to determine the complexity (compile time and failure rate of the Default mapping strategy) of a given netlist *before* starting the long and critical P&R step. It is also important to extract the features that constitute a complex design, so that the emulation partitioner can perform an educated and improved partitioning. These needs are the primary goals of our machine learning framework.

B. Our Commercial Database

After every P&R run, we perform regular expression pattern matching on the emulator logs to extract the features of interest. To build the database we first retrieve almost every feature that may be of valuable information about the compilation process with little filtering. This initial effort, for each netlist, leads to around 800 features that contain the following information:

- multi-partitioning results (emulation environment)
- synthesized RTL design of each partition
- host machine used for compilation
- targeted FPGAs
- intermediate and final results of the P&R

The data pipeline integrated in our emulation tool is shown in Figure 3. The database is updated daily with new data coming from in-house consumer compilation runs. The ML models are also updated in an off-line setting, by refitting them on the entire database. To account for the frequent changes in the compiler (e.g. partitioning settings), we weight designs differently when building the models. Our weighting depends on their recentness, so that our models perform better on what is the current state of the compilation process. The ML models are used to drive the netlists P&R: choice of appropriate P&R strategies for each partition (see Section IV), and trigger preemptive re-partitioning with balancing (see Section VI).

Table I shows typical values of some features found in our database. Note that most of the design partitions are very large with some having more than a million LUTs or tens of thousands of Control (CTRL) sets.

TABLE I: Feature characteristics of our database (built for an industry's emulation system using commercial designs).

Single FPGA partition				
-	# LUTs	# data wires	# CTRL sets	P&R time
Mean	520K	1.2M	10K	186min
Range	[800, 1.9M]	[8.7K, 3.6M]	[100, 160K]	[25m, 1day]
Complete SoC				
-	# LUTs	# FFs	# DSPs	# Partitions
Mean	24M	19M	4.5K	40
Range	[1M, 312M]	[500K, 198M]	[0, 82K]	[4, 377]

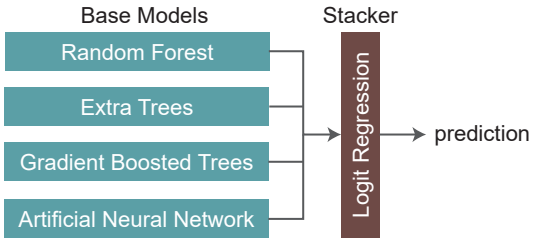

Fig. 4: Our model stacking strategy.

C. Feature Selection & Data Processing

Our database currently has around 1 million FPGA compilation entries of industry leaders' designs and is growing. Among them, we restrict ourselves to those designs with more than 20% filling rate. In addition, we reject the features that correspond to post-P&R knowledge (e.g. placement time, memory usage during routing, etc.) as they are part of what we want to predict. We then restrict our choice to 26 features directly available from the synthesized netlists before any P&R step, whose types are:

1) utilization based such as # LUTs, # Flip-Flops (FFs), # data wires, # CTRL sets, etc.
2) FPGA-based such as family, generation and amount of device resources
3) host machine-based such as # processors, CPU frequency, memory available, etc.

To further reduce the number of features, we try dimensionality reduction methods such as PCA and Autoencoders. But, they all result in a decrease of predictability performance. After feature selection, the data is processed to impute missing values, remove NaN and duplicate entries. Some benchmarks are recompiled many times for test purposes and this is not representative of the natural distribution of designs. Moreover, depending on their cardinality, we encode categorical features using one-hot or likelihood encoding. We scale numerical features to zero mean and unit variance. Skewed numerical features are also transformed by Box-Cox transformation.

III. Predicting EASY vs. HARD Netlist

Our first goal is to predict before any P&R attempt, which design partitions will end up being HARD (= hard-to-compile) or EASY (= easy-to-compile) so that we can skip the unnecessary wall-time of N hours and proceed to launch multiple strategies at time zero (see Figure 2). We state the formal definition of this problem as follows:

P1: EASY vs. HARD Netlist Classification	
Input	Netlist, target FPGA, default P&R strategy, wall-time
Output	Predict if the P&R using the default strategy will finish within the wall-time (= EASY) or not (= HARD)
Why?	If predicted right, we can skip step 1 in Figure 2 and directly start step 2, thereby saving resources used.

From the compile time and winning strategy information available in the database, we first compute the target variable EASY vs. HARD of each entry depending on the wall-time given. Our goal is to perform a supervised learning classification task, where we learn the database first using a training set. Next, for each *new* design in the test set, we use our ML model to map it to a binary label $\{0, 1\}$, where 1 corresponds to a HARD design and 0 an EASY design.

A. Model Construction & Experiment Settings

We use a powerful method used in ML called "Stacking" shown in Figure 4: 1st-level base models are fed to a 2nd-level meta-model stacker which generates the final prediction. Our base models are a combination of tree based models (XGBoost Gradient tree boosting, scikit-learn Random Forest and Extra Trees) and Artificial Neural Networks (built using Keras API). All these models have strengths and weaknesses. But, the Stacker, a simple Logistic Regression, outperforms each of them because it can highlight the benefits of each base model while discrediting where they perform poorly.

After data processing and filtering, we are left with a dataset of around 100K designs out of 1 million originally. We randomly shuffle the entries and select 90% of them for training (96,165 instances) and the remaining 10% (10,685) as the test set. We train and tune the hyper-parameters of the base models on the training set using stratified 5-folds cross-validation and Bayesian Search Optimization. The Stacker is then trained by 4-folds cross-validation on the training out-of-folds predictions of the base models. We use indexes different from the first level folds to avoid "data leakage" (causing overfitting), and tune the Stacker manually.

Because of the imbalanced nature of the problem (typical workload of 88% EASY vs. 12% HARD design partitions), our objective function is a mixture of the Area Under Curve (AUC, a rank statistic), and log-loss (a calibration statistic and strictly proper scoring rule), rather than accuracy, a metric that cannot grasp the pitfalls of imbalanced datasets. Here, the rare HARD class is the class of interest. Thus, our goal is to optimize the prediction capability on this class while staying over a reasonable accuracy on the majority EASY class. The F1-score captures this objective in our case.

B. Results & Analysis

1) Prediction Results: Depending on the request of the user, we utilize three different feature sets. The first one consists of building and testing the models using the features presented in Section II.C (= our baseline). The second one

TABLE II: Confusion matrix with our baseline feature set.

		Predicted Class	
		EASY	HARD
Actual Class	EASY	9163 (98%)	230
	HARD	260	1039 (80%)

TABLE III: Baseline vs. modified feature sets. We either remove CPU info or add super logic region (SLR) info in our modified sets.

feature set	accuracy	F1-score	AUC	log-loss
+ SLR	96.3%	0.86	98.5%	0.10
baseline	95.4%	0.81	97.2%	0.13
- CPU	93.8%	0.75	95.4%	0.16

excludes the information related to the host machine, which may not be easy to collect, e.g. the "free memory" feature that dynamically changes depending on other tasks running on the machine. Both of these levels can be categorized as "fast" prediction as they can be performed before any P&R step. The third one utilizes some information related to netlist partitioning such as Super Logic Regions (SLR) and Super Long Lines utilization of the FPGA devices.

We show in Table II the confusion matrix of our "baseline" classifier. The matrix is built using the decision probability threshold (to predict class membership) that maximizes the F1-score on the training set. In addition, Table III gathers the metrics of interest obtained by training and testing our stacked model based on the three feature sets aforementioned. We observe that if the user is willing to wait for the SLR partitioner to complete, or at least until it returns gate counts estimations, we can predict with even higher certainty the EASY and HARD classes. All in all, all our metrics confirm high predictability capability of our three ML models with a very low False Positive Rate of less than 2.5%. We also observed an expected gain of the stacked model in all the considered metrics compared with the base models (+4% accuracy, +6% F1-score, +3% AUC, and −0.07 log-loss).

2) Feature Importance: To highlight the key parameters driving the FPGA compilation complexity of a netlist, we compute from our models which features are the most important in the final EASY vs. HARD prediction. Widely used importance methods based on gain, weight or split count have showed to lead to inconsistencies. We thus decide to use the Shapley values [8] as feature importance, an attribution method inherited from coalitional game theory. Shapley values tell how to fairly distribute the "payout" (the predicted probability) among the different "players" (features). The feature importance of the top features is shown in Table IV. Our main observations are:

- The information on the host machine such as free memory and cache space are high impact features. It is expected as a heavily-loaded machine is, by experience, slower.
- We observe a predominance of the features related to LUT usage. This can be explained by the fact that about 30% of these LUTs are LUT6, spots of high connection traffic that directly impact congestion. A typical mapper usually reduces the competition for routing resources by mapping LUT6 to LUT4 in high-congestion areas.
- We note a large importance of the FPGA family. This can be explained by the influence on runtime of the differences in the internal architecture (routing, clock network and logic blocks) of the FPGAs leveraged in our emulation system, namely Xilinx Virtex-7 and UltraScale.
- After #LUT and #data wires, there is no clear outstanding feature. This confirms the fact that dimensionality reduction is detrimental, because any feature that we may remove plays a part in the predicted probability.

TABLE IV: Feature importance ranking based on their impact on output prediction.

Rank	Feature	Imp.	Rank	Feature	Imp.
1	#LUT	0.213	6	MemFree KB	0.050
2	#data wires	0.185	7	#CTRL sets	0.045
3	FPGA family	0.090	8	#clock wires	0.040
4	#LUT6	0.081	9	CPU cache KB	0.038
5	#FF	0.065	10	#Muxcy	0.036

C. Application to Wall-time Optimization

Earlier the EASY/HARD labels are originally computed for a 5-hour wall-time. We now decide to investigate the effects of reducing the wall-time (whose value can seem large and arbitrary) on our compilation process in terms of the overall compile time and hardware resources. However, the database is originally built on the results of the framework without prediction. We have no information on the "optimal" winning strategy and associated compile time of EASY designs. Instead, we only know that for the EASY jobs, the Default strategy finished in less than 5 hours. It is nonetheless possible to estimate from the database how much compile time we gain by launching additional strategies. To do so, we first find an upper bound of the compile time gain ratio $\widehat{\alpha_{CT}}$, defined as:

$$\widehat{\alpha_{CT}} = \mathbb{E}_{\boldsymbol{CT}\sim p_{\text{HARD}}(\boldsymbol{CT})}\left[\frac{ALL\left(\boldsymbol{CT}\right)}{DEF\left(\boldsymbol{CT}\right)}\right] \tag{1}$$

where $ALL\left(\boldsymbol{CT}\right)$ is the compile time when all strategies are tried concurrently (which then corresponds to the compile time of the fastest strategy), and $DEF\left(\boldsymbol{CT}\right)$ is the compile time of the Default strategy only. To bound $\widehat{\alpha_{CT}}$, notice that HARD designs—whose winning strategy is not Default—would have ran using Default only for at least 5 hours more than the recorded compile time. We use bootstrapping to show that $\widehat{\alpha_{CT}} \leq 0.67$ is verified almost surely.

Reducing the wall-time wt changes some previously EASY designs to HARD designs. In this case, HARD designs—whose winning strategy is not Default—do not see their compile time modified. However, HARD designs—whose winning strategy is Default—have their compile time modified as:

$$\mathcal{CT} = \begin{cases} CT & \text{if } CT \leq wt/(1-\widehat{\alpha_{CT}}) \\ wt + \widehat{\alpha_{CT}} \cdot CT & \text{otherwise.} \end{cases} \tag{2}$$

To show how our classifier improves the P&R process, we build the graph shown in Figure 5 to show the estimated

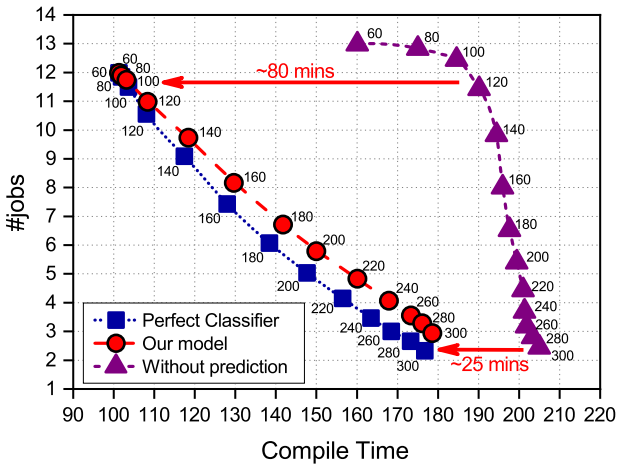

Fig. 5: Compile time improvement using our ML model. The numbers on the points indicate the wall time used. Our saving ranges from 25min to 80min depending on wall time used.

average compile time and number of jobs (#jobs) per netlist required to complete P&R of all test designs. This calculation is done based on our HARD/EASY prediction. We vary the wall-time and compare our ML model to a Perfect Classifier and to the non-ML framework presented in Figure 2. We consider a worst case scenario of 12 strategies used on a HARD design. Each wall-time corresponds to a new trained and tested model, resulting in a new F1-score-maximized confusion matrix. Our model deviates from the Perfect Classifier as the wall-time rises, as it causes the number of HARD designs available to decrease, producing a more and more imbalanced and therefore difficult classification problem.

The graph shows, for a fixed wall-time, that our prediction model improves the average compile time per design with limited effect on the average #jobs launched. The largest compile time gain is seen for a wall-time of 100min. However, this would also yield $\sim$ 11.5 jobs per design, which is a too high hardware cost. Reasonably, keeping our original wall-time of 300min (5 hours) still yields a reasonable compile time gain of 25min per design for less than one job launched.

IV. Predicting Winning Strategy

As shown in the previous section, our ML model help reduce the time used for FPGA compilation. To reduce hardware effort on top of that, we need to be able to predict the winning strategy to avoid launching more strategies than needed. We state the formal definition of this problem as follows:

P2: Winning Strategy Set Prediction	
Input	Netlist, target FPGA device, full strategy list
Output	A variable size subset of strategies that are likely to win (= finish FPGA compilation the fastest)
Why?	If predicted right, we can reduce the compilation time and the number of jobs required for the netlist.

TABLE V: Description of Default strategy and top 3 advanced strategies with highest success rates.

Name	Objectives
Default	Balances between timing closure effort and compile time. Runtime expensive algorithms are not used.
Strategy-1	Runs multiple passes of optimizations, with advanced placement and routing algorithms.
Strategy-2	Timing-driven optimization of SLR partitioning (by exploring SLR reassignments).
Strategy-3	Makes delays more pessimistic for long distance and higher fanout nets with the intent to shorten their overall wirelength.

A. Model Construction

We use the stacking and training/validation/testing methodologies presented in Section III.A, but modify the settings from binary to multiclass classification with a One-Vs-Rest (OVR) approach. We fit one classifier per class ($\equiv$ per strategy). Then, for each classifier, the class is fitted against all the other classes. Because the training sets are highly imbalanced with OVR, we follow [9] that modifies the target values so that the positive class has target $+1$ and the negative class has target $-1/(\#classes - 1)$. Because we have 4 base models and 21 classes, the input of our meta-model is 64 wide, which is large. To help with dimensionality, we use as meta-model a regularized version of the multinomial Logistic Regression.

Table V describes the objectives of the Default strategy as well as of the three strategies with highest success rates (% of times it is a Winner, excluding Default).

Our goal is to determine the winning strategy of HARD designs among the 21 available Vivado strategies. This is difficult in our framework for two reasons: First, the Default strategy is winning more often than not as it was launched first and kept running for 5 hours before any other strategy. The second reason is that not all strategies are fairly represented. Indeed, when the wall-time hits, not all 21 strategies are launched, but rather 3 or 4 are chosen, depending on the machine resources available and previous human experience with the strategies. As mentioned in Section II.A, these strategies are not design-related but rather decided by user experience.

B. Application to Job Minimization

Despite these complications, we find that predicting a set of candidate winning strategies is possible and enough to reduce the effort spent in FPGA compilation. Rather than picking a unique winning strategy, we select *multiple* strategies based on the probability vectors $\mathcal{P} = \{\mathcal{P}_i\}_{i \in \text{designs}}$ where $\mathcal{P}_i = \mathbb{P}(\text{design}_i) = (p_{L0}, \ldots, p_{L20})$ given at the output of our model. There are 21 contending thresholds, one per class L_i. The probabilities obtained yield a sense of confidence level. Deciding how to use these values is up to the user. In Table II and Section III.C, we chose to use a probability threshold to distinguish classes that maximized the F1-score. However, in our grid farm framework, time and effort embody our true

TABLE VI: Job minimization with our strategy predictor.

	# jobs	improve
no prediction	2.4	baseline
EASY/HARD classifier	2.9	-21%
perfect EASY/HARD classifier	2.2	8.3%
strategy predictor	1.6	33.3%
perfect strategy predictor	1.0	58.3%

utility functions, and optimizing these objectives will likely be at the detriment of the F1-score.

We perform thresholds tuning to minimize the overall #jobs. This problem can be mathematically formulated as:

$$\underset{T}{\operatorname{argmin}} \#jobs(CL(T,\mathcal{P}), S_{true}) \tag{3}$$

where S_{true} corresponds to the true winning strategies, and $CL(T,\mathcal{P})$ is the set of proposed strategies for each design obtained using thresholds T on the probability vectors $\mathcal{P}$. The #jobs function is expressed as:

$$\#jobs = \sum\nolimits_{i \in \text{designs}} \mathfrak{J}(i) \quad \text{with} \tag{4}$$

$$\mathfrak{J}(i) = \begin{cases} \mathbf{card}(CL(T,\mathcal{P}_i)) & \text{if } \{S_{true}\}_i \in CL(T,\mathcal{P}_i) \\ 12 & \text{otherwise.} \end{cases} \tag{5}$$

As this function is non-linear and not differentiable, we use Powell's method with an initial start point found by optimizing the F1-score of each class independently:

$$T_0 = (\underset{T}{\operatorname{argmax}} F1(L_0), \ldots, \underset{T}{\operatorname{argmax}} F1(L_{20})) \tag{6}$$

During training, we solve Equation (3) for each model and fold. The threshold vector used on the test set is then computed as the average of the cross-validation-folds thresholds. We obtain an accuracy on the test set of 67%, coinciding with an average size of strategy set proposed of $\mathbf{card}(CL) \approx 1.8$ and resulting $\#jobs \approx 5.2$ spent on HARD designs. We then use this strategy predictor in Step 2 of our pipeline shown in Figure 2 to see how the overall number of jobs is reduced. Comparison is done at original 5-hour wall-time. The new average #jobs is shown in Table VI and compared with the other flows. We observe that our strategy predictor combined with our EASY/HARD predictor provides 33% jobs savings.

V. Predicting Compile Time

To show how ML can beneficially affect productivity, we test our framework in regressing the compile time of P&Rs. We present how using the predicted values can improve computing farm utilization by optimizing the scheduling of jobs fired on the grid. We state the formal definition of this problem as follows:

P3: Compile Time Prediction	
Input	Netlist, target FPGA device, strategy used
Output	How long will the netlist compilation take?
Why?	If predicted right, we can assign it to the right server and thus make the best use of the computing resources.

A. Model Construction

The same model stacking and training/validation/testing methodology presented before is used but with regression versions of the models. In addition, the objective scoring becomes the Mean Absolute Error (MAE). We obtain a satisfactory R^2 of 0.85, showing enough correlation between predicted and actual compile times. A MAE value of 18min shows that on average the prediction is very accurate. But, a Root Mean Square Error of 37min shows it also exhibits large variations of correctness.

B. Application to Job Scheduling

Using the built ML model presented above, we predict first hand how much time each P&R job is going to take. Even if the prediction is not perfect, we use this value to our advantage to perform an improved scheduling of the jobs fired on the server grid. By that, we mean reduce the makespan of the logical schedule, i.e. the time difference between the start and finish of the sequence of jobs. We employ a modified version of an enhanced heuristic Longest Processing Time-based scheduling algorithm called SLACK [10], with time complexity of $O(n \log n)$ and whose description is:

ML-based SLACK heuristic

Input: m machines and n jobs, predicted compile times $\widetilde{CT}_j$
Output: near-optimal job schedule

1. Sort jobs by non-increasing $\widetilde{CT}_j$.
2. Consider $\lceil n/m \rceil$ tuples of size m given by jobs $1,\ldots,m; m+1,\ldots,2m$, etc. If $n \pmod m \neq 0$, add dummy jobs with null compile time in the last tuple.
3. For each tuple, compute the associated slack, namely $\widetilde{CT}_1 - \widetilde{CT}_m, \widetilde{CT}_{(m+1)} - \widetilde{CT}_{2m}, \ldots, \widetilde{CT}_{(n-m+1)} - \widetilde{CT}_n$.
4. Sort tuples by non-increasing slack and then fill a list with consecutive jobs in the sorted tuples.
5. Apply List Scheduling to this job ordering.
6. Return makespan computed using the actual matching CT_j.

In the interest of simplicity, we assume that at a given time, one job is associated to one machine and this machine only. We repeat scheduling 5000 times on $n = 100$ (a typical value in our grid) randomly sampled concurrent design partitions/individual compiles of the test set. The mean makespan obtained using our scheduling is shown in Figure 6. We compare with what was done in a non-ML environment, which by lack of knowing the P&R times, was utilizing a greedy scheduling based on the #LUTs. To see how the number of machines affects the scheduling benefits, we vary the number of machines and carry out the experiments again. We observe that our ML-based scheduler shows makespan improvements regardless the number of machines, with the largest savings of 24% obtained at $m = 40$ with roughly 200min savings on a 900min makespan. Cumulated over a 7-day week, this leads to savings of more than one and a half days.

VI. ML-based Design Recommendations

Partitioning quality can influence tremendously the P&R runtime and success rate. A poor partitioning can result in

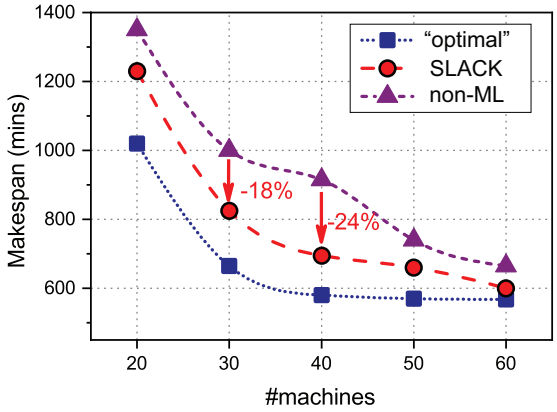

Fig. 6: Makespan improvement of our runtime regression-based SLACK scheduler. The "optimal" conducts SLACK scheduling using known, not ML predicted, compile time. The "non-ML" method assigns the largest netlist (in terms of # LUT) to the first available machine.

a large number of HARD partitions. If even one partition remains unroutable, the emulation flow shown in Figure 1 must be restarted from the partitioning step. If feature importance gives fundamental insights on the compilation features that largely make designs complex, these values are relative to the complete model and dataset. Here, we search to improve the compilation framework from "inside" the tool. This starts with providing "recommendations" on how to modify a given HARD partition to turn it into an EASY one. We state the formal definition of this problem as follows:

P4: Design Recommendation	
Input	HARD netlist, target FPGA device, trained ML model
Output	Recommendations on feature modification so that the given HARD netlist becomes EASY
Why?	The overall compilation time reduces with the new EASY netlist.

A. *Construction of Recommendations*

The authors of [11] show how individual decisions can be explained using class probability gradients. Motivated by their approach, we propose to construct recommendations based rather on probability "vectors". If the gradient indicates the direction of the steepest move from the test point, this information is local and the change in probability is mostly infinitesimal. In our case, we are interested in significant probability changes (to go under the HARD/EASY threshold), while changing the netlist as little as possible: First, to provide simple and practical recommendations to the partitioning engine ($\sim$ 2 to 3 features to change together at most). Secondly, to avoid under-populating the FPGAs too much, which cannot be done when constrained to a fixed number of partitions.

The main components of the algorithm are:

- We only consider "likely" moves by sampling from the learned distribution of the data, estimated using Kernel Density Estimation (KDE). The best kernel found is the radially symmetric kernel and the optimal bandwidth matrix $\boldsymbol{H}$ is selected by Least squares cross-validation. The KDE Probability Density Function and kernel are defined as such:

$$\hat{f}(\boldsymbol{x};\boldsymbol{H}) = n^{-1}\sum_{i=1}^{n} K_{\boldsymbol{H}}(\boldsymbol{x}-\boldsymbol{X_i}) \tag{7}$$

where

$$K_{\boldsymbol{H}}(\boldsymbol{u}) = |\boldsymbol{H}|^{-1/2} K(\boldsymbol{H}^{-1/2}\boldsymbol{u}) \tag{8}$$

and

$$K(\boldsymbol{u}) \propto (1-\|\boldsymbol{u}\|^2)\mathbb{1}(\|\boldsymbol{u}\|^2 \leq 1) \tag{9}$$

- We use a similarity distance between two partitions of the form

$$d(a,b) = \sum_{i\in\text{features}} |a_i - b_i|^{\alpha_i} \tag{10}$$

A small α_i corresponds to a prioritized feature to select. Using such a distance allows us to fix features that cannot change (e.g. FPGA) and to translate our priorities when re-partitioning. In particular, it is easier for us to generate constraints on LUT/FF/IO counts rather than net counts.
- We move recursively in a greedy manner, selecting at each iteration the one feature providing the largest $\Delta\mathbb{P}(a,b)/d(a,b)$, subject to a sufficiently large $\Delta\mathbb{P}(a,b)$. Thereby, we avoid changing too many features.

The description of our algorithm, which runs in less than 5min, is as follows:

VECTOR (x_0, S, M, X_{train}, ϵ)

Input: partition x_0, feature set S, model M: $x \mapsto \mathbb{P}(x)$
train data X_{train}, class probability threshold ϵ
Output: modified partition x_{recom}

1. Define similarity distance d;
2. F = LEARN_DISTRIBUTION_DATA (X_{train});
3. Current point: $x_{\text{recom}} \leftarrow x_0$;
4. Sampling boundary: $\delta \leftarrow \alpha$;
5. **while** ($\mathbb{P}(x_{\text{recom}}) \geq \epsilon$)
6. **for** (each s in S)
7. $Q(s)$ = SAMPLE (F, $x_{\text{recom}}(s)$, δ);
8. $V(s) = \max_{x\in Q} \dfrac{\mathbb{P}(x_{\text{recom}}) - \mathbb{P}(x)}{d(x_{\text{recom}}, x)}$ subject to $\Delta\mathbb{P} \geq t$;
9. Select feature $f = \operatorname{argmax}_{s\in S} V(s)$;
10. **if** (f empty)
11. increase δ;
12. **else**
13. update $x_{\text{recom}} \leftarrow x(f) : x \in Q(f) \mathrel{\hat{=}} V(f)$;
14. **endwhile**
15. return x_{recom};

Compared with other approaches, such as LIME [12], our method provides a definite value to change rather than just a direction of change. Also, in LIME data points are sampled from a fixed distribution that ignores the correlation between features. This can lead to unlikely data points which can then be used to learn local explanation models.

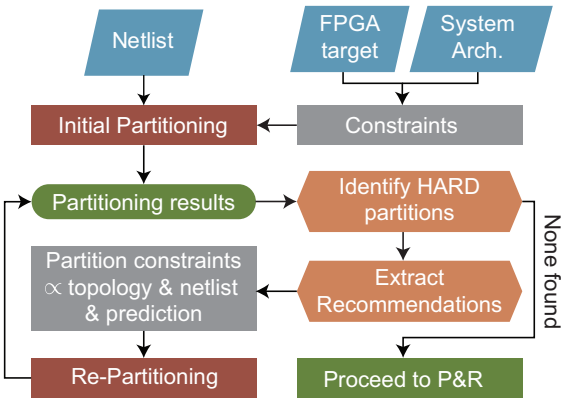

Fig. 7: Our Recommendation flow. The model built in Section III is used to identify HARD partitions. Then, the algorithm VECTOR generates the recommendations used to define new mapping constraints.

B. Re-partitioning Results

We generate recommendations to the partitioning engine inside the flow as shown in Figure 7, before any P&R step. Once a first automatic partitioning completes, we identify HARD partitions using the predictor of Section III. Our algorithm VECTOR then provides the recommended changes in these partitions, translated each to simple rules such as: *remove x LUT6 and remove y BRAMS from partition Pz*. Based on the topology of the multi-FPGA system (positions of FPGAs and inter-FPGA communication resources), the hierarchical netlist, as well as the resources available on EASY partitions, a new partition mapping file is generated. To fasten re-partitioning, the partitioner uses as input the resulting assignments from the previous partition with the balanced modules obtained from the recommendations, so that most of the design is set in place. This provides high level of stability in the results. For example, if a recommendation shows one partition has critical utilization of LUTs, a typical constraint is to remove a highly combinational module from the HARD partition. This module has to be placed on a EASY partition without endangering the fixed system constraints (maximum hop-count, time-division-multiplexing ratios, etc.). This trade will most likely make the receiving EASY partition "harder". As even minor FPGA changes can affect the P&R, the resulting changes in probability of involved partitions are computed from the trained model and the viability of the recommendation is assessed.

We show in Table VII the results of our recommendation flow applied to a commercial SoC design that contains 12.5M LUTs, 5.3M FFs and 155K multiplexed IOs. The chosen benchmark is harnessing 14 partitions, where 6 of them are HARD. For fair comparison of the runtimes, the partitions are all compiled in the same settings, i.e. on the same machines and all using Default strategy. Our ML model classified the hardness of all the partitions correctly. Our algorithm VECTOR identified two partitions with critical utilization of LUTs (↓500K) and FFs (↓300K) respectively; modules adding to such sizes were found and displaced to an EASY partition without too much increase in IO-cut. After re-partitioning, the compile time of the considered HARD partitions reduces by 32% and 68% respectively. On the other hand, the EASY partition degrades reasonably. Overall, the compilation time of the complete design reduces by 15%, with savings of 326min. Note that the re-partitioning step only takes $\sim$ 45min. Thus, our recommendations-augmented partitioning flow provides more FPGA-P&R-friendly partitions, resulting in overall faster P&R steps.

TABLE VII: Compile time (CT) improvements using our recommendation flow. We use a commercial SoC design partitioned to 14 netlists. Instances are re-partitioned across HARD-0, HARD-1, and EASY-0.

	total CT	worst CT	HARD-0 netlist	HARD-1 netlist	EASY-0 netlist
init. partition	2205	524	524	361	35
after re-partition	1879	357	357	115	139

VII. Conclusions

Our machine learning framework allows accurate handling of runtime intensive netlists as well as appropriate compilation strategies. Our study derives an effective way to improve the trade-off between compile time vs. number of jobs by varying the wall-time. Integrated in our emulation system, our ML models prove to reduce compilation cost by optimally schedule runs on the server grid. This results in 24% makespan savings. Our automatic strategy selection results in 33% jobs savings. Our new method to propose recommendations is shown to be effective in improving the quality of the partitioning, consequently speeding up the overall compile time.

References

[1] R. Tessier, "Multi-FPGA Systems: Logic Emulation," in *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann, 2008, ch. 30.

[2] W. N. Hung and R. Sun, "Challenges in Large FPGA-based Logic Emulation Systems," in *Proceedings of the 2018 International Symposium on Physical Design*. New York, NY, USA: ACM, 2018.

[3] A. B. Kahng, "Machine Learning Applications in Physical Design: Recent Results and Directions," in *Proceedings of the 2018 International Symposium on Physical Design*. New York, NY, USA: ACM, 2018.

[4] N. Kapre, B. Chandrashekaran, H. Ng, and K. Teo, "Driving Timing Convergence of FPGA Designs through Machine Learning and Cloud Computing," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2015.

[5] A. Mametjanov *et al.*, "Autotuning FPGA Design Parameters for Performance and Power," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2015.

[6] C. Xu *et al.*, "A Parallel Bandit-Based Approach for Autotuning FPGA Compilation," ser. FPGA '17. New York, NY, USA: ACM, 2017.

[7] G. Grewal *et al.*, "Automatic Flow Selection and Quality-of-Result Estimation for FPGA Placement," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2017.

[8] S. M. Lundberg and S.-I. Lee, "A Unified Approach to Interpreting Model Predictions," in *Advances in Neural Information Processing Systems 30*, 2017.

[9] Y. Lee, Y. Lin, and G. Wahba, "Multicategory Support Vector Machines," *Journal of the American Statistical Association*, 2004.

[10] F. Della Croce and R. Scatamacchia, "The Longest Processing Time rule for identical parallel machines revisited," *Journal of Scheduling*, 2018.

[11] D. Baehrens *et al.*, "How to Explain Individual Classification Decisions," *J. Mach. Learn. Res.*, Aug. 2010.

[12] M. T. Ribeiro, S. Singh, and C. Guestrin, ""Why Should I Trust You?": Explaining the Predictions of Any Classifier," *CoRR*, 2016.