Improving FPGA-Based Logic Emulation Systems through Machine Learning

ANTHONY AGNESINA and SUNG KYU LIM, Georgia Institute of Technology
ETIENNE LEPERCQ and JOSE ESCOBEDO DEL CID, Synopsys Inc.

We present 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 place and route 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 to perform re-partitioning of difficult partitions. Tested on a large-scale industry system on chip design, our recommendation flow provides additional 15% compile time savings for the entire system on chip. To exploit our ML model inside the time-critical multi-FPGA partitioning step, we implement it in an optimized multi-threaded representation.

CCS Concepts: • **Hardware → Simulation and emulation**; • **Computing methodologies → Machine learning approaches**;

Additional Key Words and Phrases: Field programmable gate array, SoC verification, emulation flow optimization with machine learning

ACM Reference format:
Anthony Agnesina, Sung Kyu Lim, Etienne Lepercq, and Jose Escobedo Del Cid. 2020. Improving FPGA-Based Logic Emulation Systems through Machine Learning. *ACM Trans. Des. Autom. Electron. Syst.* 25, 5, Article 46 (July 2020), 20 pages.
https://doi.org/10.1145/3399595

1 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.

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.
Authors' addresses: A. Agnesina and S. K. Lim, Georgia Institute of Technology, North Avenue NW, Atlanta, GA 30332; emails: agnesina@gatech.edu; limsk@ece.gatech.edu; E. Lepercq and J. Escobedo Del Cid, Synopsys Inc., 690 East Middlefield Road, Mountain View, CA 94043; emails: {elepercq, jpablo}@synopsys.com.

1084-4309/2020/07-ART46 $15.00
https://doi.org/10.1145/3399595

To achieve successful mapping of large ASIC designs found, for instance, in the automotive, 5G, networking, artificial intelligence, and datacenter segments, an emulator integrates many hundreds of FPGAs. Commercial FPGAs can provide larger capacity and faster runtime performance (up to 5 MHz) compared with custom FPGAs or special-purpose custom logic processor-based architectures. However, these FPGAs are not suited to the very high pin-to-gate ratio requirements of logic emulation systems [18]. 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 [7]. 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 [8] (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 (DSE) of CAD tool parameters. Grewal et al. [6] show that it is possible to predict the best quality-of-results (QoR) placement flow among a reduced set of candidate flows, using supervised learning of previous P&R runs. The approach proposed by Mametjanov et al. [13] based on selective sampling allows finding parameter configurations for improved timing and power consumption. They also show they can predict relatively accurately performance and power across the entire design parameter space. Xu et al. [19] and Yu et al. [22] use a state-of-the-art tool auto-tuner [1] to find tool parameter settings in the FPGA and high-level synthesis compilation flows that optimize QoR. It is based on the multi-armed bandit problem to organize and control a set of classical optimization techniques to explore design space efficiently. Kapre et al. [9] use Bayesian learning to generate good-quality FPGA CAD tool parameter configurations. Yanghua et al. [21] perform feature selection to reduce the number of parameters to consider, and instead of predicting the best design points, Meng et al. [15] eliminate the non-optimal design points by regression. Outside of DSE, examples of the use of ML to predict properties of a given netlist include estimations of several post-implementation metrics [4] or routing congestion after placement [23] or during placement [16, 20]. ML has also been used to improve CAD algorithms by incorporating reinforcement learning and support vector machines to perform P&R in 3D-FPGAs [14].

Compared with the mentioned related work, we do not focus in this work on power or performance metrics. In our case, designs are set to compile at low frequencies, and timing closure is rarely an issue. As the purpose of the framework is to obtain an emulated FPGA version of an ASIC design made for debug and validation purposes, working functionality dominates over speed. Moreover, our goal is not DSE. Our approach revolves around predicting compile time given a fixed netlist and parameter settings. Based on compile time prediction, we propose several ways to enhance our emulation framework. The recommendation part devised here is unique to our work and arises from the multi-FPGA nature of our problem where utilization balancing between FPGAs is possible. Other approaches focus on optimizing a unique FPGA, and none of the presented studies target important issues related to compile time, nor have they 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 article are as follows:

- We build a complete ML data pipeline framework, allowing for the extraction of numerous predictors.

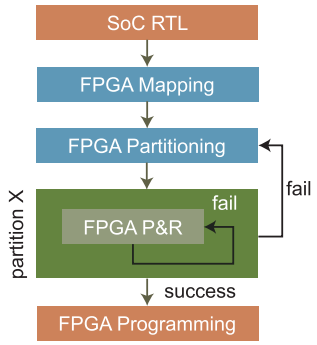

Fig. 1. Our multi-FPGA emulation scheme with FPGA re-compilation. Rare cases include re-partitioning.

- 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.

2 ML 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 multi-level hierarchical approach and a similar algorithm to hMetis [10]. As simultaneous objectives need to be optimized during partitioning (hop counts, cut-sizes, maximum FPGA utilization, etc.), it is possible that the required partitioning quality cannot be met without user input. In case of designs that are very large for the number of available FPGAs inside the emulator, re-partitioning may be necessary to successfully map on these highly utilized FPGAs. In this situation, P&R success is very sensitive to the quality of the partitioning.

After partitioning, a system route step determines how the signals between FPGAs will be routed using cables and connectors. Preliminary steps to prepare the SoC design for FPGA implementation are also necessary, including RTL changes and addition of debug instrumentation.

2.1 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 (≡ 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 running tasks for this FPGA are aborted, and the pending tasks are cancelled. Here, each job

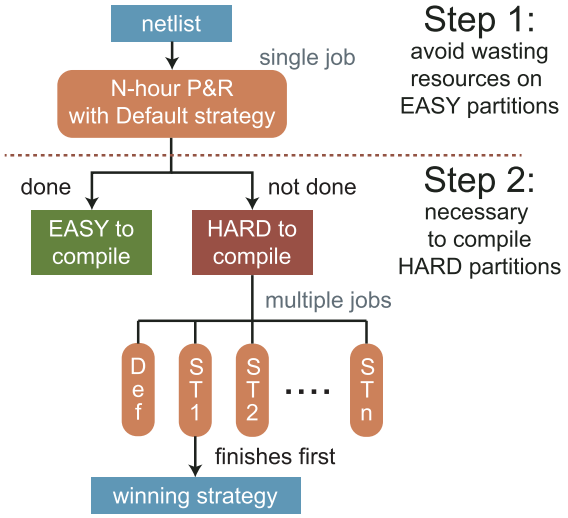

Fig. 2. Our two-step FPGA P&R flow. EASY netlist finishes in step 1, whereas HARD continues with the default run, along with new jobs added, into step 2.

implements a different strategy, such as a combination of Vivado parameters like area optimization, QoR placement and routing parameters, or physical optimization.

In a traditional flow, the strategies selected are not design related but mostly are 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 the 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 article, 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 of 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 ML framework.

2.2 Our Commercial Database

After every P&R run, we perform regular expression pattern matching on the host workstation logs to extract the features of interest. To build the database—a distributed NoSQL Apache Cassandra database with a size of a few gigabytes—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 (particularly compile time and winning strategy).

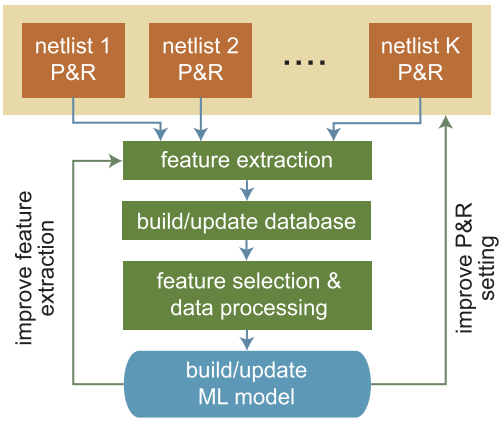

Fig. 3. Our ML framework. We update our database and ML models upon new compilation data being added.

Table 1. 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	186 min
Range	[800, 1.9M]	[8.7K, 3.6M]	[100, 160K]	[25 m, 1 day]
Complete SoC				
-	# LUTs	# FFs	# DSPs	# Partitions
Mean	24M	19M	4.5K	40
Range	[1M, 312M]	[500K, 198M]	[0, 82K]	[4, 377]

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 offline setting, by re-fitting them on the entire database. To account for the frequent changes in the compiler (partitioning strategy, time-division-multiplexing (TDM) ratios, etc.), 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 4) and trigger preemptive re-partitioning with balancing (see Section 6).

Table 1 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. If a large SoC is partitioned onto K FPGAs, it will account for K entries of the database. The value of K can go up to 400 for very large SoCs. The FPGAs used in our emulation system are Xilinx Virtex-7 2000T and Xilinx Virtex-UltraScale 440 FPGAs with 1.9M and 5.5M available logic gates, respectively.

An alternative use of statistical methods can be found in the database collection in itself, by helping us discard invaluable entries and improve feature extraction. In fact, we started storing the EDIF netlists compressed in gzip format (each up to approximately 130 MB), not just the aforementioned features, to let us explore later, if needed, the benefits of analyzing them. However, even efficient hashing schemes still result in very large disk storage requirements. According to our updating rate, storing the complete zipped compilation logs and netlists results in tens of terabytes

of additional storage per year. Thus, discarding some netlists has become necessary. By discarding the EDIF netlists of easy-to-compile (= EASY) designs that are predicted very accurately as such by our ML models (whose further netlist analysis is then non-essential), we are able to save 86% of the storage space. Moreover, statistical analysis is also beneficial in avoiding unnecessary feature extraction steps. Indeed, we were able to identify features that require intensive computations to be extracted, but are highly correlated to the simpler and more common features we describe in the next section. In particular, we now avoid computing and collecting metrics related to groups of tangled logics (GTLs) [5] (size, cut size, rent exponent, minimal GTL-SD score, etc.). These metrics were originally believed to be important in determining the complexity of a netlist, but data analysis proved that they are non-essential. This is important, as their extraction is long (a lot of effort is spent in performing multiple searches starting with different seeds to generate a large population of linear orderings and candidate GTLs) and difficult (GTLs can be very varied in size and the GTL-SD signal is often noisy).

2.3 Feature Selection and Data Processing

Our database currently has around 1M FPGA compilation entries of industry leaders' designs and is growing. Among them, we restrict ourselves to those designs with more than 20% of the filling rate. In addition, we reject the features that correspond to post-P&R knowledge (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, recursive feature elimination, and autoencoders. However, they all result in a decrease of predictability performance. After feature selection, the data is processed to impute missing values, and 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 a mean of zero and unit variance. Skewed numerical features are also transformed by Box-Cox power transformation. Note that scaling is only required for models such as neural networks but not tree-based algorithms.

3 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 next.

P1: EASY vs. HARD Netlist Classification	
Input	Netlist, target FPGA, default P&R strategy, wall time
Output	Predict if the P&R session using the Default strategy will finish (= the netlist fits into the FPGA) 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.

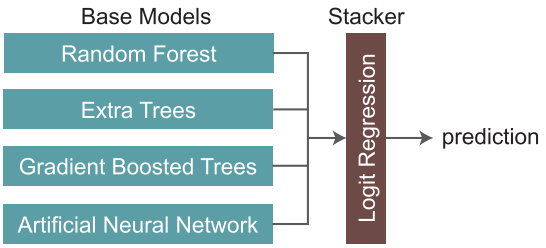

Fig. 4. Our model stacking strategy.

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 to an EASY design.

3.1 Model Construction and Experiment Settings

We use a powerful method used in ML called *stacking*, which is shown in Figure 4. The predictions from the first-level base models are fed to a second-level meta-model stacker that 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 of these models have strengths and weaknesses—for example, tree-based methods are good with noisy, high-dimensionality data, and pretty insensitive to over-fitting. 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 1M 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. Given that some partitions belong to the same SoC, it is most likely that the model "sees" all SoC designs during training. However, this is not important, as our goal is to predict accurately each partition individually. Although some SoC designs might be similar, the partitions of a given SoC are usually quite diverse (one partition will contain either memory blocks, or host interface logic, or debug instrumentation, etc.). We train and tune the hyper-parameters of the base models on the training set using stratified fivefold cross validation and Bayesian search optimization. The Stacker is then trained by fourfold 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 over-fitting) and tune the Stacker manually. The total training time of the stacked model is around 2 hours. The model is stored serialized in compressed gzip format to reach the size of 5 GB.

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 the 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.

3.2 Results and Analysis

3.2.1 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

Table 2. Confusion Matrix with Our Baseline Feature Set

		Predicted Class	
		EASY	HARD
Actual Class	EASY	9,163 (98%)	230
	HARD	260	1,039 (80%)

Table 3. Baseline vs. Modified Feature 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

Note: We either remove CPU info or add SLR info in our modified sets.

in Section 2.3 (= our baseline). The second one excludes the information related to the host machine, which may not be easy to collect—for example, 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 region (SLR) and super long line utilization of the FPGA devices. Before calling the Xilinx tool to perform P&R, the features are extracted or estimated at runtime by traversing the graph representation of the netlist. For instance, the number of CTRL sets is estimated knowing that the Xilinx tool will ultimately optimize them based on a few rules that we consider. To estimate SLR utilization, we perform a quick, multiple-way h-Metis partitioning similar to that of the Xilinx engine. Feature collection is done in parallel per bundle of 8 to 16 FPGAs, and takes a few seconds without SLR estimation and up to 12 minutes with it.

We show in Table 2 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 3 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 of 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 of the considered metrics compared with the base models (+4% accuracy, +6% F1-score, +3% AUC, and −0.07 log-loss).

3.2.2 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 been shown to lead to inconsistencies. We thus decide to use the Shapley values [12] 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 some of the top features is shown in Table 4. Our main observations are as follows:

- The information on the host machine, such as free memory and cache space, includes high-impact features, which is expected, as a heavily loaded machine is, by experience, slower. This showcases the reality of our framework where engineers battle for computing resources. However, in a perfect scenario without contention, we still believe that the host

Table 4. Feature Importance Ranking Based on Their Impact on Output Prediction

Rank	Feature	Imp.	Rank	Feature	Imp.
1	# LUTs	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	# FFs	0.065	10	# Muxcy	0.036

machine information affects compile time. EDA tools are memory intensive and built to rely heavily on the parallelization of their inner algorithms. It is most often the case that multi-core CPUs will place and route faster than single-core processors.

- 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 actually LUT6, which are 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 Virtex-UltraScale. Interested in further understanding the large importance of the FPGA family, we also compute the feature importance on the different sets of data relative to the two FPGA families. We find that the number of clock wires is two times more important in the Virtex-7 than in the Virtex-UltraScale. We believe that this can be explained by the integration of better leaf clocking resources in the ultrascale device (i.e., "ASIC-like" clocking). Because the selected features are not unique to Xilinx FPGAs, we believe that our framework can be applied directly to other brands of FPGAs, such as Intel/Altera. Moreover, because of the similarity in internal FPGA architectures, we also speculate that our built models could be re-trained online with new compilation results on these different FPGAs and still achieve relatively good accuracy.
- 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.

If we look at the feature importance on the feature set including the SLR information (not reported here), we find that the ranking in importance of SLR utilization is $3 - 2 - 1 - 0$. Indeed, as SLR crossings can induce non-negligible delays, the SLR partitioner tries to fill SLR0 first before filling the contiguous one (SLR1 and so on), meaning that a design with SLR3 utilization is a very large design (i.e., potentially HARD by experience).

3.3 Application to Wall-Time Optimization

Earlier, the EASY/HARD labels were 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

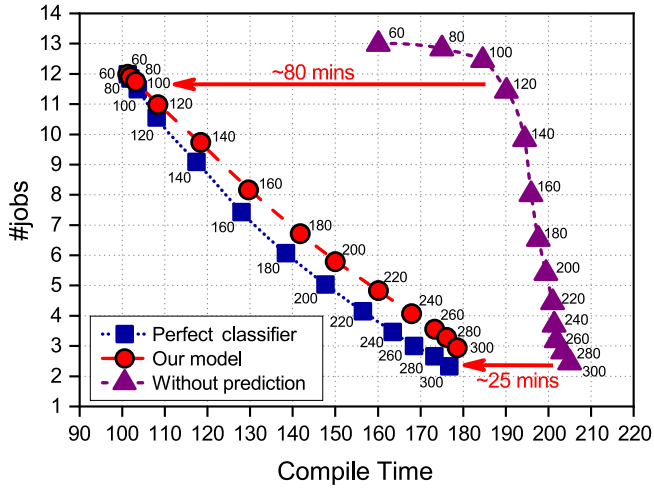

Fig. 5. Compile time improvement using our ML model. The horizontal axis represents the average compile time per partition. The numbers on the points indicate the wall time used. Our saving ranges from 25 to 80 minutes depending on wall time used.

launching additional strategies. To do so, we first find an upper bound of the compile time gain ratio $\widehat{\alpha_{CT}}$, which is defined as

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

where $ALL\,(\mathbf{CT})$ is the compile time when all strategies are tried concurrently (which then corresponds to the compile time of the fastest strategy), and $DEF\,(\mathbf{CT})$ 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 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 100 minutes. However, this would also yield approximately 11.5 jobs per

Table 5. Description of the Default Strategy and the Top 3 Advanced Strategies with the 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 re-assignments).
Strategy-3	Makes delays more pessimistic for long distance and higher fanout nets with the intent to shorten their overall wirelength.

design, which is a too high of a hardware cost. Reasonably, keeping our original wall time of 300 minutes (5 hours) still yields a reasonable compile time gain of 25 minutes per design for less than one job launched.

4 PREDICTING WINNING STRATEGY

As shown in the previous section, our ML model can 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.

4.1 Model Construction

We use the stacking and training/validation/testing methodologies presented in Section 3.1 but modify the settings from binary to multi-class classification with a one-vs.-rest (OVR) approach. We fit one classifier per strategy (≡ per class). Then, for each classifier, the strategy is fitted against all of the other strategies. Because the training sets are highly imbalanced with OVR, we follow the work of Lee et al. [11], which modifies the target values so that the positive strategy has target +1 and the negative class (i.e., the remaining strategies) has target $-1/(\#strategies - 1)$. Because we have four base models and 21 strategies, the input of our meta-model is 64 wide, which is large. To help with dimensionality, we use as the meta-model a regularized version of the multinomial logistic regression.

Table 5 describes the objectives of the Default strategy, as well as those of the three strategies with the highest success rates (percentage of times it is a winner, excluding Default). The strategy *Flow_RuntimeOptimized* is not one of them, as it has in fact a very low success rate. Despite targeting a faster runtime, this strategy often fails to compile on difficult netlists.

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

Table 6. Job Minimization with Our Strategy Predictor
(Wall Time = 300 minutes)

	# 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%

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 2.1, these strategies are not design related but rather are decided by user experience.

4.2 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 $\mathcal{P}_i = \mathbb{P}(\text{design}_i) = (p_{L_0}, \ldots, p_{L_{20}})$ given at the output of our model. There are 21 contending thresholds, one per strategy L_k. The probabilities obtained yield a sense of confidence level. Deciding how to use these values is up to the user. In Table 2 and Section 3.3, 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 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

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

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

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

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

As this function that we ought to minimize 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:

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

During training, we solve Equation (3) for each model and each 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 $\textbf{card}(CL) \approx 1.8$ 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 the original 5-hour wall time. The new average #jobs is shown in Table 6

and compared with the other flows. We observe that our strategy predictor combined with our EASY/HARD predictor provides the 33% jobs savings mentioned in the abstract.

5 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.

5.1 Model Construction

The same model stacking and training/validation/testing methodology presented earlier 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. An MAE value of 18 minutes shows that on average the prediction is very accurate. However, a root mean square error of 37 minutes shows that it also exhibits large variations of correctness.

5.2 Application to Job Scheduling

Using the built ML model presented previously, we predict firsthand 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—that is, 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* [3], with time complexity of $O(n \log n)$ and whose description is as follows:

ML-based SLACK heuristic

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

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.
5. Create a job ordering by filling a list with consecutive jobs in the sorted tuples.
6. Apply list scheduling to this job ordering, and obtain a FIFO schedule per machine.
7. Return makespan computed using the actual matching CT_j.

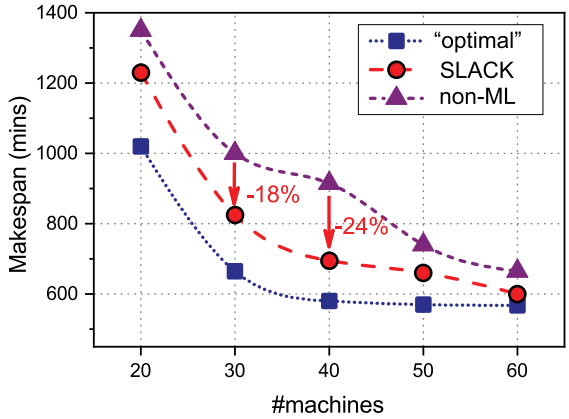

Fig. 6. Makespan improvement of our runtime regression-based SLACK scheduler. The "optimal" conducts SLACK scheduling using known, not ML predicted, compile time. The "non-ML" method assigns the largest netlist (in terms of # LUT) to the first available machine.

This scheduling is performed offline, and no job balancing between machines is done at runtime. Although this may be suboptimal at runtime, the list scheduling still results in a FIFO schedule per machine incurring zero downtime and under-utilization. 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 5,000 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 200-minute savings on a 900-minute makespan, as mentioned in the abstract. Cumulated over a 7-day week, this leads to savings of more than 1 and a half days.

6 ML-BASED DESIGN RECOMMENDATIONS

Partitioning quality can tremendously influence the P&R runtime and success rate. A poor partitioning can result in a large number of HARD partitions. If even one partition remains unroutable, the emulation flow shown in Figure 1 must be re-started 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.

6.1 Construction of Recommendations

Baehrens et al. [2] show how individual decisions can be explained using class probability gradients. Motivated by their approach, we rather propose to construct recommendations based 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 (approximately two to three features to change together at most), and second, to avoid under-populating the FPGAs too much, which cannot be done when constrained to a fixed number of partitions.

The main components of our algorithm are the following:

- 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:

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

  where

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

  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.

Compared with other approaches, such as LIME [17], our method provides a definite value to change rather than just a direction of change. In addition, in LIME, data points are sampled from a fixed distribution that ignores the correlation between features. This can lead to unlikely data points that can then be used to learn local explanation models.

The description of our algorithm, which runs in less than 5 minutes, 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: $x_{\text{recom}} \leftarrow x_0$;

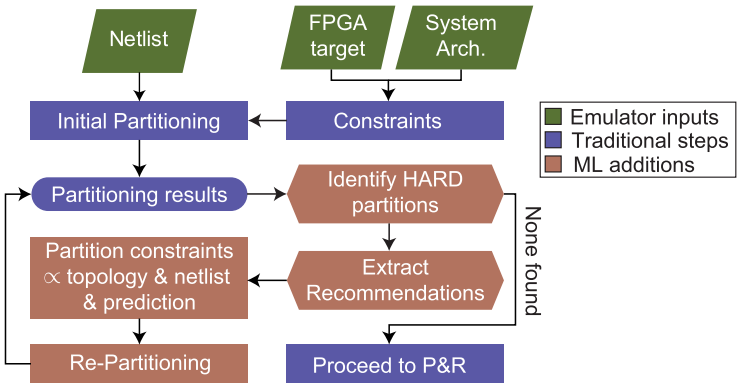

Fig. 7. Our recommendation flow. The model built in Section 3 is used to identify with high confidence the HARD partitions. Then, the algorithm VECTOR generates the recommendations used to define new hierarchical mapping constraints.

4. Sampling boundary: $\delta \leftarrow \alpha$;
5. **while** $(\mathbb{P}(x_{\text{recom}}) \geq \epsilon)$
6. **for** (each s in S)
7. $Q(s) = \text{SAMPLE}\ (F, x_{\text{recom}}(s), \delta)$;
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 $f = \underset{s \in S}{\text{argmax}}\, V(s)$;
10. **if** (f empty)
11. increase δ;
12. **else**
13. update $x_{\text{recom}} \leftarrow x(f) : x \in Q(f) \triangleq V(f)$;
14. **endwhile**
15. return x_{recom};

6.2 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 3. 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, and 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 a high level of stability in the results. For example, if a recommendation shows that 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, TDM ratios, etc.). This trade will most likely make the receiving EASY partition "harder." As even minor FPGA changes can affect the P&R, the

Table 7. Compile Time (CT) Improvements in Minutes Using Our Recommendation Flow

	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

Note: We use a commercial SoC design partitioned to 14 netlists. LUTs and FFs are re-partitioned across HARD-0, HARD-1, and EASY-0.

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 7 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 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. However, the EASY partition degrades reasonably. Overall, the compilation time of the complete design reduces by 15% as mentioned in the abstract, with savings of 326minutes. Note that the re-partitioning step only takes approximately 45minutes. Thus, our recommendations-augmented partitioning flow provides more FPGA-P&R-friendly partitions, resulting in overall faster P&R steps.

7 RUNTIME INFERENCE

We seek to use our EASY vs. HARD prediction during the initial partitioning step shown in Figure 7 to assess partitioning quality very early in the flow. This step iteratively improves a starting random partitioning by moving modules around, until a sufficiently good solution is found, where the quality is usually scored in terms of total cut size. This process is repeated multiple times to obtain a set of potential candidates. A candidate partitioning is a bundle of partitions of size at most approximately 400. To predict the quality of a candidate during that step, the predictions on all partitions of that candidate must be obtained extremely fast in a few milliseconds. However, our generated trained Python models are too slow for that particular task. Thus, we decide to transform our ML models into a more efficient C++ representation.

Because our methodology stacks many different base models together, migrating all of the models from Python to C++ is a considerable effort. Therefore, as a first approach, we choose to transform random forest first, as it is the most accurate stand-alone model. Using this model alone causes a drop in accuracy of 4% compared with stacking.

From the sklearn random forest Python model, we extract the decision rules of each decision tree, which we store in a text file. This file is then parsed to fill in our C++ object as described in Figure 8. The chosen C++ random forest representation is a standard template library vector of "flattened" trees, where *flattened* refers to their layout in memory. The vector representation is coherent with the algorithm of a random forest, which is an ensemble of decision trees where the

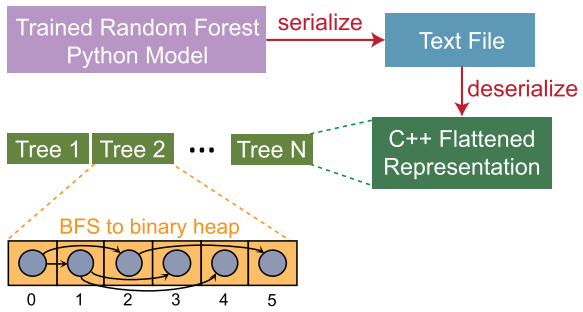

Fig. 8. Our Python model transformation pipeline for high-speed inference.

Table 8. Improvements in Runtime of Prediction by Model Conversion and Multi-Threading (#threads = 10)

Model	Prediction Time per Partition (*us*)	Speed-Up vs. Python
Python	500	—
C++ flattened	20	25
Parallelized forest	11	45
Parallelized bundles	2.2	230

final prediction is the average of the prediction of the trees. Each flattened tree is itself a vector of "flat" nodes, whose structure is as follows:

```
struct FlatNode {
  int   feature_index;
  float threshold;
  int   left_child_index;
  int   right_child_index;
};
```

For a given set of `features`, the decision path consists of moving from one node to the next by jumping in the vector from the node index to `left_child_index` or `right_child_index`, depending on the value of the predicate `features[feature_index] < threshold`. Because of the flattened data layout, this path most likely corresponds to memory locations available in the same cache line. The nodes indexes in the vector are set using breadth-first search (BFS) of the trees as in binary heaps. This initial implementation is improved with the addition of multi-threading performed in two ways: we either parallelize for each partition the calculation of the forest's prediction among the trees or parallelize among the partitions of a bundle. The second solution was found largely superior as shown in Table 8 and scales almost linearly with the number of threads. The runtime of the multi-threaded forest computation is crippled by the large discrepancy in the depths of the different trees composing the forest. All in all, our flattened tree C++ representation provides fast inference of the partitioning quality in the largest emulation systems, with 400 partitions predicted in a few milliseconds.

8 CONCLUSION

Our ML 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. Our high-speed random forest implementation allows the use of the model in the time-critical initial partitioning step. Considering the "chaos" present in EDA tools, with unpredictable behaviors of P&R heuristics in complex advanced nodes, these results are encouraging in demonstrating the benefits of ML in FPGA compilation.

REFERENCES

[1] Jason Ansel and Shoaib Kamil. 2014. OpenTuner: An extensible framework for program autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT'14)*.

[2] David Baehrens, Timon Schroeter, Stefan Harmeling, Motoaki Kawanabe, Katja Hansen, and Klaus-Robert Muller. 2010. How to explain individual classification decisions. *Journal of Machine Learning Research* 11 (Aug. 2010), 1803–1831.

[3] Federico Della Croce and Rosario Scatamacchia. 2018. Longest processing time rule for identical parallel machines revisited. arXiv:1801.05489.

[4] Steve Dai, Yuan Zhou, Hang Zhang, Ecenur Ustun, Evangeline F. Y. Young, and Zhiru Zhang. 2018. Fast and accurate estimation of quality of results in high-level synthesis with machine learning. In *Proceedings of the 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'18)*.

[5] Tanuj Jindal, Charles J. Alpert, Jiang Hu, Zhuo Li, Gi Joon Nam, and Charles B. Winn. 2010. Detecting tangled logic structures in VLSI netlists. In *Proceedings of the 47th Design Automation Conference (DAC'10)*. ACM, New York, NY.

[6] Gary Grewal, Shawki Areibi, Matthew Westrik, Ziad Abuowaimer, and Betty Zhao. 2017. Automatic flow selection and quality-of-result estimation for FPGA placement. In *Proceedings of the 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW'17)*.

[7] William N. N. Hung and Richard Sun. 2018. Challenges in large FPGA-based logic emulation systems. In *Proceedings of the 2018 International Symposium on Physical Design (ISPD'18)*. ACM, New York, NY.

[8] Andrew B. Kahng. 2018. Machine learning applications in physical design: Recent results and directions. In *Proceedings of the 2018 International Symposium on Physical Design (ISPD'18)*. ACM, New York, NY.

[9] Nachiket Kapre, Bibin Chandrashekaran, Harnhua Ng, and Kirvy Teo. 2015. Driving timing convergence of FPGA designs through machine learning and cloud computing. In *Proceedings of the 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*.

[10] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. 1999. Multilevel hypergraph partitioning: Applications in VLSI domain. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 7, 1 (March 1999), 69–79.

[11] Yoonkyung Lee, Yi Lin, and Grace Wahba. 2004. Multicategory support vector machines. *Journal of the American Statistical Association* 99, 465 (2004), 67–81.

[12] Scott Lundberg and Su-In Lee2017. A unified approach to interpreting model predictions. arXiv:1705.07874.

[13] Azamat Mametjanov, Prasanna Balaprakash, Chekuri Choudary, Paul D. Hovland, Stefan M. Wild, and Gerald Sabin. 2015. Autotuning FPGA design parameters for performance and power. In *Proceedings of the 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*.

[14] Rajkumar Manimegalai, E. Siva Soumya, Velusamy Muralidharan, Balaraman Ravindran, Veezhinathan Kamakoti, and Davinder Bhatia. 2005. Placement and routing for 3D-FPGAs using reinforcement learning and support vector machines. In *Proceedings of the IEEE International Conference on VLSI Design*.

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

[16] Chak-Wa Pui, Gengjie Chen, Yuzhe Ma, Evangeline F. Y. Young, and Bei Yu. 2017. Clock-aware ultrascale FPGA placement with machine learning routability prediction. In *Proceedings of the 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'17)*.

[17] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "Why should I trust you?" Explaining the predictions of any classifier. arXiv:1602.04938.

[18] Russell Tessier. 2008. Multi-FPGA systems: Logic emulation. In *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann, 637–669.

[19] Chang Xu, Gai Liu, Ritchie Zhao, Stephen Yang, Guojie Luo, and Zhiru Zhang. 2017. A parallel bandit-based approach for autotuning FPGA compilation. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'17)*.

[20] Dani Maarouf, Abeer Alhyari, Ziad Abuowaimer, Timothy Martin, Andrew Gunter, Gary Grewal, Shawki Areibi, and Anthony Vannelli. 2018. Machine-learning based congestion estimation for modern FPGAs. In *Proceedings of the 2018 28th International Conference on Field Programmable Logic and Applications (FPL'18)*. 427–4277.

[21] Que Yanghua, Harnhua Ng, and Nachiket Kapre. 2016. Boosting convergence of timing closure using feature selection in a learning-driven approach. In *Proceedings of the 2016 26th International Conference on Field Programmable Logic and Applications (FPL'16)*.

[22] Cody Hao Yu, Peng Wei, Max Grossman, Peng Zhang, Vivek Sarker, and Jason Cong. 2018. S2FA: An accelerator automation framework for heterogeneous computing in datacenters. In *Proceedings of the 55th Annual Design Automation Conference (DAC'18)*. Article 153.

[23] Jieru Zhao, Tingyuan Liang, Sharad Sinha, and Wei Zhang. 2019. Machine learning based routing congestion prediction in FPGA high-level synthesis. In *Proceedings of the 2019 Design, Automation, and Test in Europe Conference and Exhibition (DATE'19)*.

Received June 2019; revised December 2019; accepted May 2020