Machine Learning Approach for Loop Unrolling Factor Prediction in High Level Synthesis

Georgios Zacharopoulos, Andrea Barbon, Giovanni Ansaloni and Laura Pozzi
Faculty of Informatics
Università della Svizzera italiana
Lugano, Switzerland
Email: {georgios.zacharopoulos},{andrea.barbon},{giovanni.ansaloni},{laura.pozzi}@usi.ch

***Abstract*—High Level Synthesis development flows rely on user-defined directives to optimize the hardware implementation of digital circuits. Nevertheless, the most beneficial directive values are hard to predict, and exhaustive explorations are infeasible even for moderately complex designs. Focusing on the Loop Unrolling directive, we herein address this challenge by proposing a novel Machine Learning methodology, able to jointly forecast the optimal loop unrolling factors for all the loops in a target application. We showcase that our method results in a better prediction score (up to 63%) and a reduced convergence time compared to other state-of-the-art approaches. Our method achieves 90% of the speedup that can be obtained (with a perfect a-priori knowledge of optimal loop unrolling factors) when synthesizing the computational hotspots of each considered benchmark as hardware accelerators.**

***Index Terms*—Loop Unrolling; Machine Learning; High Level Synthesis; Hardware/Software Co-design; Customizable Processors; ASIPs.**

I. Introduction

High Level Synthesis (HLS) frameworks allow designers to describe hardware circuits at a higher abstraction with respect to traditional Register Transfer Level (RTL) methodologies. HLS tools, such as LegUp [1], ROCCC [2] and Vivado HLS [3] decouple the description of the intended hardware functionality (usually expressed in C or C++) and its implementation, which is governed by a set of optimization directives.

HLS provides an effective implementation pathway for heterogeneous systems combining processors and accelerators, because parts of an application can be easily re-targeted from software to hardware, often without any source code modifications. On the other hand, the performance of the generated hardware is challenging to control from a designer perspective, due to two compounding factors. First, hardware synthesis is a time-consuming process, limiting in practice the amount of possible implementations that can be evaluated. Second, the effect of assigning different values to directives is difficult to foresee, due to low-level application characteristics. To cope with these issues, simulation tools such as Aladdin [4] have been developed so as to avoid a complete hardware synthesis, while rapidly estimating the performance and cost (area) of HLS-defined designs. The developers of Aladdin report approximations of the area and latency of designs within few percentage points with respect to full synthesis flows (both in terms of area and latency) while requiring 500X less computation time.

Nonetheless, even when employing estimation tools, an exhaustive evaluation of all directives settings for each candidate accelerator in a heterogeneous system is still unfeasible beyond simple cases. Addressing this challenge, we herein propose a machine learning framework that is able to infer the proper implementation of an HLS design based on its characteristics, automatically derived from a source code analysis pass, that we developed within the LLVM compiler framework [5].

We focus on hardware loop unrolling, an HLS directive that targets loops whose trip count can be statically defined, a common case for applications targeting heterogeneous systems. Loop unrolling instantiates multiple copies of the logic implementing the functionality defined in a loop body, tangibly impacting the performance of accelerators [6] [7]. This directive should nonetheless be applied judiciously, because it entails a high area cost for the duplicated logic; in addition, its ensuing benefits can be hampered by loop-carried dependencies and frequent memory accesses.

In this context, our framework is able to assign, given a target cost-performance trade-off, a loop unrolling factor for each loop in an HLS-defined accelerator with high precision, while waiving the need for a design space exploration. Following the common practice in machine learning methodologies, we rely on a training phase to establish the link between loop features and target metrics (performance and area). After training, hardware unrolling factors are then inferred for previously-unseen loops, according to their Control-Data Flow Graph characteristics (e.g.: the critical path) and their performed operations (e.g.: the amount of loads and stores).

Our contribution is two-fold:

- We illustrate a novel methodology, based on Random Forest machine learning classification [8], which is able to accurately predict unrolling factors for loops in HLS designs.
- We introduce an automated framework, integrated in the LLVM compiler, that a) automatically extracts relevant loop information from the analysis of the source code Intermediate Representation, and b) leverages the derived features to assign directive values for loop unrolling factors.

Our method has higher predictive scores, and lower average errors, with respect to state of the art alternatives [9], while requiring less computational effort for training. Accelerators designed with our framework achieve comparable performance with respect to the ones derived from exhaustive explorations.

The paper proceeds as follows: Section II illustrates related efforts in the field. Sections III and IV describe the hardware unrolling prediction framework and comparatively evaluate its performance. Section V concludes the paper.

II. Related Work

Previous works have explored the applicability of machine learning to drive compiler optimizations. In software compilers, it has been employed by Agakov et al. [10] to speed up iterative compilation, by Monsifrot et al. [11] to produce compiler heuristics and by Kulkarni et al. [7] to select the order in which optimization passes should be performed. Stephenson et al. [9] have applied supervised classification algorithms, such as Nearest Neighbor (NN) classification and Support Vector Machines (SVM), in order to accurately predict unrolling factors. In all above-mentioned research works the authors targeted software compilation; in Section IV, we comparatively evaluate the performance of our framework with the methodology proposed by Stephenson et al., showcasing the benefit of our choice of loop features and classification strategy in the HLS scenario.

Similarly to us, Liu et al. [12] used a Random Forest classification model in the context of HLS, extending the Iterative Refinement framework proposed in [13] [14] [15] and [16]. They address a different problem with respect to us: that of retrieving the set of Pareto-optimal implementations of a given design by navigating its configuration space. A similar stance, addressing system-level design, is illustrated by Ozisikyilmaz et al. [17]. As opposed to these works, our aim is to perform a predictive assignment of synthesis directives, based on a training performed on a disjoint input set. This problem was also investigated by Kurra et al. [6]. Contrary to their methodology, we do not rely on a detailed estimation delay model of the loop body datapath and control logic.

III. Methodology

In this section, we first define the employed objective function and the performance metrics we considered to evaluate our framework performance. Then we introduce the LLVM analysis pass that we developed in order to automatically extract relevant loop features and the approach we followed to retrieve the area and run-time performance of HLS designs. Lastly, the supervised learning classifier method is detailed, which, during the training phase, gathers the data from the previous steps to produce a loop unrolling predictor, and, during the test phases, assigns loop unrolling factors based on loop features.

A. Objective Function

The design of an accelerator implementation involves a trade-off between its performance (execution latency) and its required resources (area). Depending on the implementation constraints, the relative relevance of these objectives may vary. We therefore consider a parametric **Impact (I)** function as follows:

$$I(L,A) = \alpha \cdot \frac{(L_1 - L)}{L_1} + (1-\alpha) \cdot \frac{(A_1 - A)}{A_1},\ 0 \leq \alpha \leq 1$$

Where L and A are the latency and area of the function being synthesized as an accelerator when a Loop Unrolling Factor value (LUF) is adopted for the target loop. L_1 and A_1 are instead the latency and area for the same accelerator with LUF equal to one (i.e., when the target loop is fully rolled).

We can now define the *optimal* LUF as the one that maximizes the Impact function above. Note that, when $LUF = 1$, then $I(L,A) = 0$, which corresponds to a baseline implementation. $I(L,A)$ may also be negative for sub-optimal LUF choices (where unrolling might increase area without decreasing latency), but will always be ≥ 0 for optimal unrolling factors.

For the experimental evaluation described in Section IV, we considered a set of seven commonly-used LUFs $S :< 1, 2, 4, 8, 16, 32,\ 64 >$. Our methodology is nonetheless not limited to these values, and can be applied to any set of candidate LUFs. If a loop trip count can not be exactly divided by an element in S, loop peeling is automatically applied by the employed HLS tool (described in Section III-C) to the last loop iterations. Unfeasible LUF choices, when a loop trip count is smaller that the desired unrolling factor, are skipped both during the training and the test phases.

The relevance of latency and area can be expressed by setting the α parameter. For the evaluation presented in Section IV we explored three different configurations: a) Optimize equally for latency and area ($\alpha = 0.5$). In this configuration we maintain a balance between decreasing the execution time and keeping low the usage of hardware resources. b) Optimize mostly for latency ($\alpha = 0.9$). Minimizing latency is favored by this approach, thus focusing on increasing the speedup of an application, and finally c) Optimize mostly for area ($\alpha = 0.1$). This setting aims at decreasing the area budget of the implementation, often at the cost of achieving a smaller speedup.

To evaluate the classification performance of a trained classifier, we adopted two different metrics. The *Prediction Score* states the percentage of optimal (according to $I(L,A)$)

TABLE I
Features extracted by LLVM LU Analysis Pass.

Features - X Vector
Critical Path
Loop Trip Count
Has Loop Carried Dependencies
Load Instructions
Store Instructions

TABLE II
Feature vectors selected by Stephenson et al. [9].

Features - X Vector 1	Features - X Vector 2
# Operands	*# Floating Point Operations*
Range Size	*Loop Nest Level*
Critical Path	*# Operands*
# Operations	*# Branches*
Loop Trip Count	*# Memory Operations*

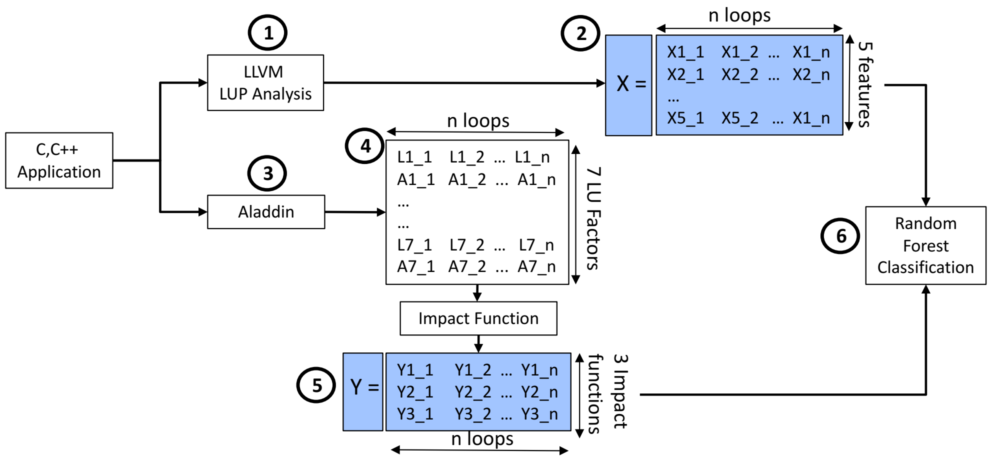


Fig. 1. Overview of the Loop Unrolling Prediction methodology.

LUFs that were correctly identified on the out-of-sample test set. The *Average Error* instead measures the average distance between the indexes in S of the correct and the predicted LUF.

B. LLVM-based Loop Features extraction

Loop features are automatically identified in our framework by an analysis pass (depicted as point 1 in Figure 1) that we developed within the LLVM compiler toolchain [5]. Features are retrieved starting from applications written in C or C++, operating on their Intermediate Representation, provided by the LLVM front-end passes. The -Oz compilation flag was used during compilation from the source code to the Intermediate Representation level, in order to avoid vectorization of the loops, which hinders a correct inspection of the loop features.

Our *LLVM Loop Unrolling Prediction Analysis Pass* iterates over functions of the applications and identifies loops. On each of them, it performs loop, scalar evolution and dependence analysis to retrieve their features, summarized in Table I: the critical path, the trip count, the presence of loop carried dependencies and the required memory accesses (load and stores).

Our choice of features is based on the factors that influence the cost and the achievable speedup of hardware-unrolled loops: a loop with a long critical path may be expensive to duplicate, while loop carried dependencies and memory accesses may force a serialization of execution irrespectively of the degree of unrolling. These considerations lead us to consider a markedly different feature list with respect to works focusing on software targets, such as the one of Stephenson et al. (Table II).

To gather the required feature values, we built upon existing methods (e.g. the getTripCount method belonging to the ScalarEvolution class reference), and implemented an LLVM analysis pass, whose pseudo-code is presented in Algorithm 1. The output of the algorithm is, for each loop, a feature vector X stating its characteristics, represented as point 2 in

Algorithm 1 LLVM Analysis Pass - Loop Unrolling Prediction Analysis

Input: Application written in C, C++
Output: X (Feature Vector)

```
 1: function RunOnFunction( )
 2:     for BB in Function do
 3:         if L=getLoopForBB() then
 4:             LoopUnrollingPredictionAnalysis(BB,L)
 5:
 6: function LoopUnrollingPredictionAnalysis(Basic Block BB, Loop L)
 7:     LI=getLoopInfoAnalysis()
 8:     SE=getScalarEvolutionAnalysis()
 9:     DA=getDependenceAnalysis()
10:     /* Gather Features for X Vector */
11:     x1=getCriticalPath(BB)
12:     x2=getTripCountForLoop(L)
13:     x3=getLoopCarriedDependencies(BB)
14:     x4=getNumberOfLoadInstructions(BB)
15:     x5=getNumberOfStoreInstructions(BB)
```

Figure 1. Feature vectors are used during training to tune the classifier (described in Section III-D), and during the test phase to predict high-impact loop unrolling factors.

C. Area and Latency Estimation

To establish a link between LUFs and performance/cost of implementations, area and latency values must be retrieved both for the loops in the training set (in order to optimize the classifier) and the ones in the test set (to measure its accuracy).

To compute them, we relied on Aladdin [4] (point 3 in Figure 1), a pre-RTL power-performance simulator for hardware accelerators. We simulated all functions contained in the considered benchmarks, adopting, for the contained loops, each feasible unrolling factor in the S set defined in Section III-A. Latency is reported by Aladdin in clock cycles, while

Algorithm 2 Random Forest Classification - Training and Test

Input: X and Y Vectors

Output: Trained Random Forest Classifier

```
1: for i in NumberOfTrainingSessions do
2:     X_train, X_test, Y_train, Y_test=train_test_split(X,Y)
3:     /* Training Phase */
4:     M=RandomForestLearningModel
5:     M.train(X_train, Y_train)
6:     /* Evaluation Phase */
7:     Pred=M.predict(X_test)
8:     Error=abs(Pred-Y_test)
9:     Score=M.score(X_test-Y_test)
```

area is expressed in μm^2 in a $45nm$ technology. The result is shown as point 4 in Figure 1.

We then computed the Impact (I) for the different α values, to retrieve the optimal loop unrolling factor for every loop of a function, which is the index of the LUF that maximizes I. The result is three vectors $\{Y1, Y2, Y3\}$ (point 5 in Figure 1) that contain the target values for the classification algorithm. The $Y1$ vector includes the optimal loop unrolling factor that balances the hardware implementation of the accelerators in terms of low latency and low area. Values in the $Y2$ vector favors low-latency implementations, applying more aggressive loop unrolling, whereas $Y3$ favors low-area ones.

D. Random Forest Classification

Supervised learning tasks, and in particular classification problems, require an appropriate selection both of the features and of the employed model. The rationale behind our choice of features is presented in Subsection III-B. Herein, we discuss the adopted classification strategy (point 6 in Figure 1) and the process we employed to validate it.

We used Random Forest as our supervised learning model, which has been shown by Liu et al. [12] to outperform alternatives such as Multilayer Neural Networks and Support Vector Machines classification in the context of HLS design space exploration. Random Forest algorithms follow a decision tree methodology, combining many weak classifiers to derive a strong one, allowing the generation of low-complexity and robust classifiers.

The algorithm employed, as presented in Algorithm 2, follows an approach similar to a *k-fold cross validation* strategy. The whole data set (X and Y vectors, see points 2 and 5 in Figure 1) is divided randomly between a training set and a test set, where the training set is equal to 80% of the whole data set and the remaining 20% is the test set. Then, the Random Forest model is used for the training process on the training set and out-of-sample predictions are carried out for each element of the test set. After all predictions on the test set have been computed, the prediction score and the average error (as defined in Section III-A) are computed for the current training session.

The process is repeated on different random partitions between training and test data, for 1000 times. As a last step, the mean value of the prediction score and average error over all iterations is computed, measuring the overall predictive performance of our approach. To aggregate the results obtained from different cross-validation rounds, we defined the predicted LUFs for each loop as the average across all performed out-of-sample predictions, rounded to the nearest value (index of the S set). The aggregated predictions were stored in a new Y vector, with one element per loop in the data set.

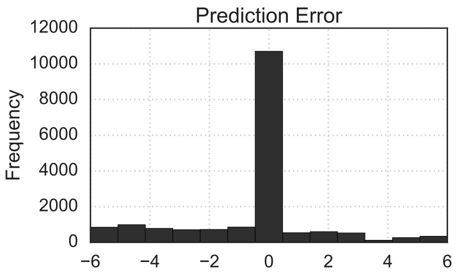


Fig. 2. Distribution of Loop Unrolling Factor Prediction Errors over 18.000 out-of-sample predictions.

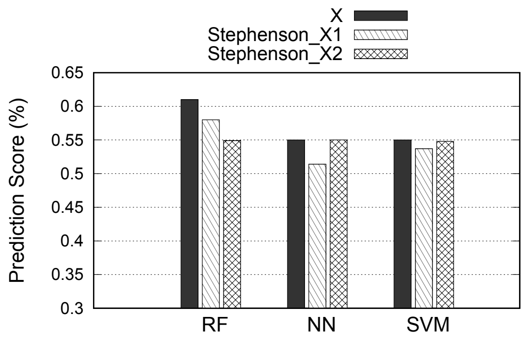
IV. Experimental Results

In order to comparatively evaluate our proposed methodology, combining Random Forest classification and LLVM-based loop features extraction, we investigated its performance on benchmarks of different complexity. Small and medium-sized ones are `adpcm`, an audio encoding kernel, `stencil`, an implementation of an iterative algorithm that updates array elements according to a given pattern, and `sha`, a secure hash encryption method used in the information security domain. `jpeg` and `mpeg2` are instead larger benchmarks, which perform image and video compression, respectively. Applications were drawn from the CHStone [18] and the Scalable Heterogeneous Computing (SHOC) benchmark suites [19]. In total, they comprise 87 different loops.

We implemented Random Forest classification using the Scikit-learn suite [20], that includes state-of-the-art implementations of Machine Learning models in python. Scikit-learn was also employed to re-implement the two methods proposed by Stephenson et al. [9], that we consider as baselines.

Giving an initial proof of concept for our strategy, Figure 2 reports the difference between the indexes of the predicted optimal (according to impact value) loop unrolling factors and the ones retrieved with an exhaustive exploration, considering 18.000 out-of-sample predictions on all the benchmark loops. Results are highly concentrated on zero, indicating a high rate of correct predictions.

In the following, we further investigate the performance of our frameworks from multiple viewpoints: the benefits of our choice for the classifier and the features are discussed in Section IV-A. Then, a comparison is drawn with an alternative implementation relying on Iterative Refinement, proposed, among other, by Palermo et al. [14] as an effective strategy when applied to HLS explorations. We conclude the Section by discussing the run-time of our methodology.



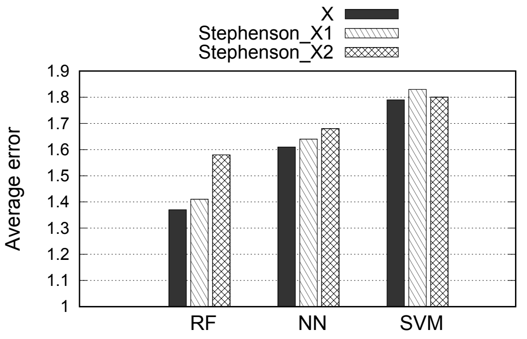
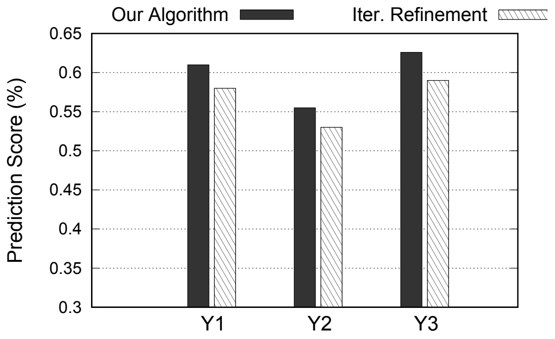


Fig. 3. Comparison of the Prediction Score (top) and Average Error (bottom) across Random Forest, Nearest Neighbor, Support Vector Machines models and the respective feature selection: our X vector, Stephenson et al. X1 and X2 vectors [9].



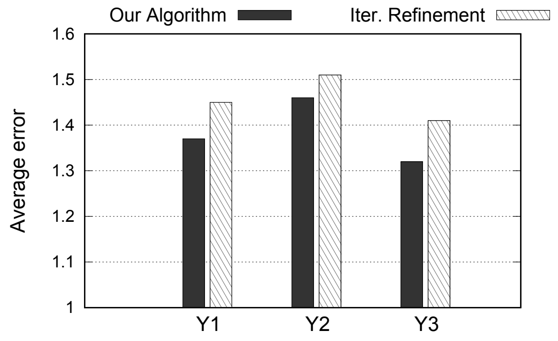


Fig. 4. Comparison of the Prediction Score (top) and Average Error (bottom) between our algorithm and Iterative Refinement [13] [14] [15] [16] across all three Y vectors (Y1: balances latency and area, Y2: favors speedup by minimizing latency and Y3: favors a low HW area usage).

A. Classification Models and Features

To evaluate our choice of features (X vector in Table I) and training model Random Forest (RF), we compare our approach against two state-of-the-art methodologies proposed by Stephenson et al. [9]. With respect to our work, the latter present different classification strategies: Support Vector Machines (SVM) and Nearest Neighbor (NN) and a different choice of investigated loop features, reported in Table II. The latter include the number of operands, the live range size, the number of floating point operations and the loop nest level.

Figure 3 shows, for a choice of $\alpha = 0.5$, the prediction score and the average error of the nine strategies resulting from the adoption of different feature vectors and classification strategies (ours and the ones of Stephenson et al.). Experimental results highlight that our framework (X feature vector and RF classification) outperform other choices, reaching a prediction score above 60% and an average error of less than 1.4. Similar results were obtained for impact functions favoring area or latency ($Y2$ and $Y3$ vectors).

To further test our method, we evaluated the performance on the cross product of the three models and the three feature vectors, which further emphasize that X and the Random Forest systematically outperform, in terms of average error as well as prediction score, other alternatives, presenting a positive marginal contribution to the global improvement achieved by our methodology.

B. Comparison with Iterative Refinement

In the second round of experiments, we compared our method against an Iterative Refinement approach, used in [13] [14] [15] [16]. Iterative Refinement uses part of the training data set to obtain a first version of the classifier, whose performance is then improved by using a second, disjoint set of input and outputs.

For this evaluation, we considered the three different settings of Y target vectors $\{Y1, Y2, Y3\}$, as described in Section III-A. The employed data, the features (X vector) and the training model (Random Forest) were the same both for our algorithm and the one using Iterative Refinement. For Iterative Refinement, we allocate 75% of the training set for the initial training phase, and the remaining 25% for the refinement phase.

The prediction score, as seen in Figure 4, ranges from 53% to 63% across the three Y vectors. Nevertheless, our methodology consistently outperforms the Iterative Refinement approach, while achieving the highest prediction score (63%) for the setting that favors low area resources ($Y3$). A similar observation can be made for the average error values, where our approach keeps a lower average error for all predictions, across all vectors, with the one related to the $Y3$ vector being the lowest (1.32).

Figure 5 reports the speedup, area and impact metrics of HLS designs optimized with our predictive method, comparing

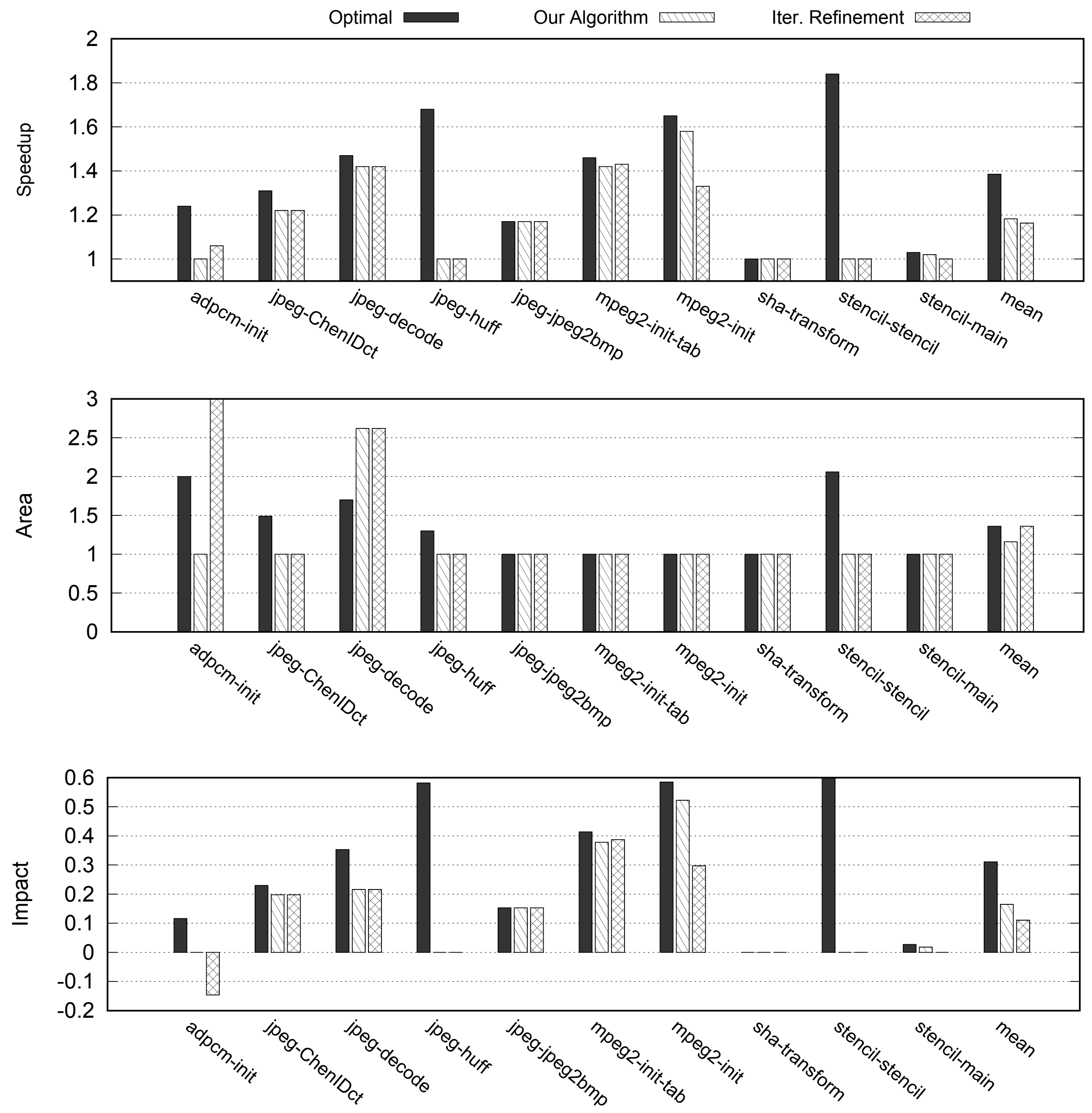
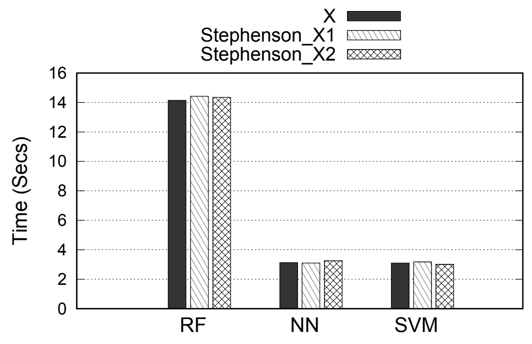


Fig. 5. Comparison of the Speedup, Area and Impact achieved for every function by our Algorithm and by an Iterative Refinement approach [13] [14] [15] [16], compared to the optimal solutions derived from exhaustive explorations. Speedup and Area numbers are normalized with respect to fully rolled configurations.

them to an Iterative Refinement approach and to results obtained from an exhaustive exploration. The graphs correspond to an impact function with $\alpha = 0.9$ (similar results were obtained for other α values). Two considerations can be drawn from the reported data: first, in most cases our methodology closely tracks the user-defined trade-off between area and latency. In this respect, `jpeg-huff` and `stencil-stencil` are outliers, since their loop structure is quite complex, making their optimization challenging to automate. Second, the impact achieved by our approach is equal or higher (by 66% in the case of `mpeg2`) than the impact attained by Iterative Refinement. On average, our methodology obtains 86% of the speedup achieved by the optimal factor retrieved with a costly exhaustive exploration (90% for $\alpha = 0.5$ and 92% for $\alpha = 0.1$).

C. Convergence Time Comparison

Besides retrieving high-quality LUF predictions, our framework also requires a lower computational effort with respect to other methods. In this regard, experimental evidence is shown



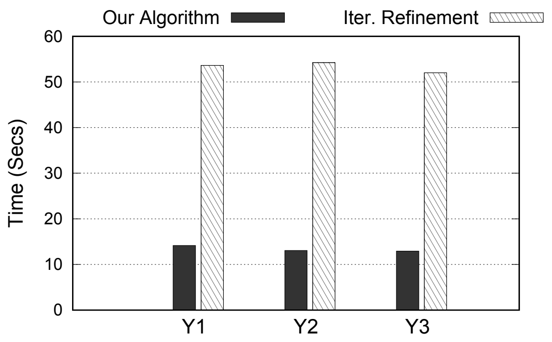


Fig. 6. Top: time required to converge for state-of-the-art Machine Learning models with our X feature vector and Stephenson et al. X1 and X2 vectors [9]. Bottom: convergence time of our algorithm and Iterative Refinement across all three Y vectors.

in Figure 6, reporting the time required for training and testing, comparing different classification strategies, feature vectors and impact functions.

As expected, the choice of employed features, as well as the relative relevance of area and latency, does not tangibly impact the computing time. On the other hand, the type of employed classifier has a noticeable effect, with Random Forest being slower to converge than NN and SVM. Nonetheless, only 14 seconds were required by our approach. The difference between the Iterative Refinement approach and our methodology, though, is even more significant, as the the former requires almost four times more than our methodology to converge.

It is worthwhile to mention that all these approaches require orders of magnitude less convergence time with respect to exhaustive explorations, whose runtime may range from minutes (for estimation tools like Aladdin [4]) to hours (for synthesis frameworks such as Vivado HLS [3]).

V. Conclusions

We have presented a novel methodology based on LLVM analysis and Random Forest classification that performs loop unrolling factor predictions for HLS designs. Our approach achieves better prediction score and less average error in comparison to state-of-the-art Machine Learning methods. Experimental evidence showcases that, by carrying out accurate predictions of loop unrolling factors, high performance accelerator implementations can be realized, while avoiding highly time-consuming exhaustive explorations.

References

[1] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "LegUp: An Open-source High-level Synthesis Tool for FPGA-based Processor/Accelerator Systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 2, pp. 1–27, Sep. 2013.

[2] J. R. Villarreal, A. Park, W. A. Najjar, and R. Halstead, "Designing modular hardware accelerators in C with ROCCC 2.0," in *Proceedings of the 18th IEEE Symposium on Field-Programmable Custom Computing Machines*, 2010, pp. 127–134.

[3] Xilinx, "Vivado high-level synthesis," www.xilinx.com/products/design-tools/vivado/integration/esl-design.html, Mar. 2017.

[4] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, "Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *Proceedings of the 41st Annual International Symposium on Computer Architecture*. IEEE, 2014, pp. 97–108.

[5] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2nd International Symposium on Code generation and optimization*, Palo Alto, California, Mar. 2004, p. 75.

[6] S. Kurra, N. K. Singh, and P. R. Panda, "The impact of loop unrolling on controller delay in high level synthesis," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*. EDA Consortium, 2007, pp. 391–396.

[7] S. Kulkarni and J. Cavazos, "Mitigating the compiler optimization phase-ordering problem using machine learning," *ACM SIGPLAN Notices*, vol. 47, no. 10, pp. 147–162, 2012.

[8] L. Breiman, "Random Forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, Oct. 2001.

[9] M. Stephenson and S. Amarasinghe, "Predicting unroll factors using supervised classification," in *Proceedings of the 3rd International Symposium on Code generation and optimization*. IEEE, 2005, pp. 123–134.

[10] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. O'Boyle, J. Thomson, M. Toussaint, and C. K. Williams, "Using machine learning to focus iterative optimization," in *Proceedings of the 4th International Symposium on Code generation and optimization*. IEEE Computer Society, 2006, pp. 295–305.

[11] A. Monsifrot, F. Bodin, and R. Quiniou, "A machine learning approach to automatic production of compiler heuristics," in *AIMSA*, vol. 2. Springer, 2002, pp. 41–50.

[12] H.-Y. Liu and L. P. Carloni, "On learning-based methods for design-space exploration with high-level synthesis," in *Proceedings of the 50th Design Automation Conference*. IEEE, 2013, pp. 1–7.

[13] G. Mariani, G. Palermo, V. Zaccaria, and C. Silvano, "Oscar: An optimization methodology exploiting spatial correlation in multicore design spaces," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 5, pp. 740–753, 2012.

[14] G. Palermo, C. Silvano, and V. Zaccaria, "ReSPIR: a Response Surface-Based Pareto Iterative Refinement for Application-Specific Design Space Exploration," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 12, pp. 1816–1829, Nov 2009.

[15] S. Xydis, G. Palermo, V. Zaccaria, and C. Silvano, "A meta-model assisted coprocessor synthesis framework for compiler/architecture parameters customization," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, 2013, pp. 659–664.

[16] M. Zuluaga, A. Krause, P. Milder, and M. Püschel, "Smart design space sampling to predict pareto-optimal solutions," in *ACM SIGPLAN Notices*, vol. 47, no. 5. ACM, 2012, pp. 119–128.

[17] B. Ozisikyilmaz, G. Memik, and A. Choudhary, "Efficient System Design Space Exploration Using Machine Learning Techniques," in *Proceedings of the 45th Design Automation Conference*. ACM, Jun. 2008, pp. 966–969.

[18] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii, "CHStone: A benchmark program suite for practical c-based high-level synthesis," in *Proceedings of the 2008 IEEE International Symposium on Circuits and Systems*. IEEE, 2008, pp. 1192–1195.

[19] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (shoc) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 2010, pp. 63–74.

[20] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.