MLComp: A Methodology for Machine Learning-based Performance Estimation and Adaptive Selection of Pareto-Optimal Compiler Optimization Sequences

Alessio Colucci[1,*], Dávid Juhász[2,*], Martin Mosbeck[2], Alberto Marchisio[1], Semeen Rehman[2], Manfred Kreutzer[3], Günther Nadbath[3], Axel Jantsch[2], Muhammad Shafique[1]
[1]*Institute of Computer Engineering, Technische Universität Wien (TUWien), Vienna, Austria*
[2]*TU Wien, Christian Doppler Laboratory for Embedded Machine Learning, Vienna, Austria*
[3]*ABIX GmbH, Vienna, Austria*
Email: {alessio.colucci,david.juhasz,martin.mosbeck,alberto.marchisio,semeen.rehman,axel.jantsch, muhammad.shafique}@tuwien.ac.at,{mkreutzer,gnadbath}@a-bix.com

***Abstract*—Embedded systems have proliferated in various consumer and industrial applications with the evolution of Cyber-Physical Systems and the Internet of Things. These systems are subjected to stringent constraints so that embedded software must be optimized for multiple objectives simultaneously, namely reduced energy consumption, execution time, and code size. Compilers offer optimization phases to improve these metrics. However, proper selection and ordering of them depends on multiple factors and typically requires expert knowledge. State-of-the-art optimizers facilitate different platforms and applications case by case, and they are limited by optimizing one metric at a time, as well as requiring a time-consuming adaptation for different targets through dynamic profiling.**

To address these problems, we propose the novel *MLComp* methodology, in which optimization phases are sequenced by a Reinforcement Learning-based policy. Training of the policy is supported by Machine Learning-based analytical models for quick performance estimation, thereby drastically reducing the time spent for dynamic profiling. In our framework, different Machine Learning models are *automatically* tested to choose the best-fitting one. The trained Performance Estimator model is leveraged to efficiently devise Reinforcement Learning-based multi-objective policies for creating quasi-optimal phase sequences.

Compared to state-of-the-art estimation models, our Performance Estimator model achieves lower relative error ($< 2\%$) with up to 50$\times$ faster training time over multiple platforms and application domains. Our Phase Selection Policy improves execution time and energy consumption of a given code by up to 12% and 6%, respectively. The Performance Estimator and the Phase Selection Policy can be trained efficiently for any target platform and application domain.

I. Introduction

The number and complexity of embedded systems are constantly growing [1, 2]. Recent years saw an advent of the Internet of Things (IoT) and Cyber-Physical Systems (CPSs), and their subsequent applications [3, 4, 5]. These systems are tightly resource-constrained, requiring latency-limited real-time operation with a very low power budget. Software running on them must be optimized and tailored to the specific hardware.

Major optimizing compilers, like LLVM [6] and GCC [7], provide an ever-increasing number of optimization phases to improve operational characteristics of embedded software. The phases are applied during compilation in sequence. Their optimal selection and ordering depend on the program to be compiled and on the target platform, as well as on the final optimization objective. The value of the objective function must be estimated at compile-time to tune phase sequencing.

Standard phase selection and ordering policies in optimizing compilers [6, 7] are fixed algorithms that have been tuned for the average case and do not exactly fit to actual use-cases. State-of-the-art approaches for choosing the optimization phases are based on Machine Learning (ML), e.g. Supervised Learning (SL) and Reinforcement Learning (RL) [8, 9, 10, 11], and other adaptive mechanisms [12, 13]. Some approaches ignore phase ordering and deal with phase selection only [9, 14], while the order is important for the quality of the generated code [15]. Most of the works optimize programs for one specific metric only, like execution time [8, 10, 12, 14] or energy consumption [13]. *The state-of-the-art solutions are typically not generic, i.e., they provide good results only in a limited environment and for one specific metric at a time.* Moreover, these methods gather the required metrics by profiling execution, which is super-expensive in time, and should be replaced by a fast-yet-accurate estimation method to reduce the total adaptation time.

TABLE I: Comparison of ML-based state-of-the-art phase selection policies.

Solution	Technique	Metrics			Phase Ordering	Dynamic Features
		Execution Time	*Energy Consumption*	*Code Size*		
COBAYN [14, 22]	SL	X			No	Profiling
Milepost GCC [9]	SL	X		X	No	Profiling
MiCOMP [8]	SL	X			Static	Profiling
[10]	RL	X			Dynamic	Profiling
[23]	SL	X			Dynamic	Profiling
MLComp (PSS)	***RL***	***X***	***X***	***X***	***Dynamic***	***Prediction***

TABLE II: Comparison of state-of-the-art performance estimators.

Solution	Automation	Machine Learning	Metrics				Data Gathering	Accuracy
			Execution Time	*Energy Consumption*	*# Executed Instructions*	*Average Power*		
[16]	Limited	Basic				X	Profiling	∼5%
[17]	Limited	Basic	X			X	Profiling	∼5%
[18]	Limited	Basic				X	Simulation	∼3%
[19]	No	No				X	Simulation	∼2%
[20]	Limited	No	X			X	Profiling	∼7%
[21]	No	No				X	Simulation	<5%
MLComp (PE)	***Full***	***Advanced***	***X***	***X***	***X***	***X***	***Profiling***	***<2%***

State-of-the-art estimation models can be distinguished as ML-based [16, 17, 18] and formal [19, 20, 21] ones. ML-based models tend to use accurate sensors and interfaces to estimate the power, and hence require external modifications. However, these methods focus only on a single metric and employ only a small selection of models with at most 5% relative error. Formal models estimate the power using formulas and accurate simulation of switching activity that guarantees high accuracy. However, they require deep knowledge of the internal details of the target platform.

Our work aims at overcoming specific limitations of state-of-the-art solutions in compiler phase sequencing and performance estimation. Tables I and II highlight the shortcomings of major

*These authors contributed equally

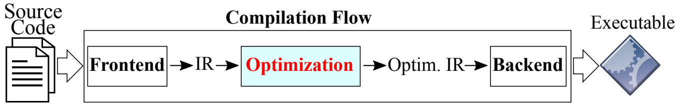

Fig. 1: A compiler converts source code into an executable by passing it through optimizations in Intermediate Representation.

state-of-the-art with respect to different properties/features, and show comprehensive and superior coverage of our *MLComp* methodology. Our phase sequence selector possesses the following key attributes:

- It utilizes *RL*, which has been given little attention in the literature for obtaining optimal phase sequences.
- It optimizes programs for multiple objectives in contrast to typical single-objective optimizations.
- It supports fast adaptation for different application domains.

The latter is enabled by fast performance estimation in the adaptation phase, which requires proper performance modeling of target platforms. To adapt to the different target platforms efficiently, potential models need to be evaluated and the best-fitting one selected. This process, which is usually done by manual analysis and design [21], is automated in our solution.

Our efforts are focused on the following *scientific challenges*, which, to the best of our knowledge, have not been addressed in the literature before:

- automatic evaluation of different ML-based performance models to support adaptation to different platforms;
- efficient training of adaptive phase sequence selection policies for multi-objective optimization of programs.

Novel Contributions: To address the above challenges, we propose a novel methodology *MLComp* that employs:

- **adaptive analytical models** for estimating energy consumption and execution time, which are trained on features of target applications executing on a given target platform; and
- **adaptive phase sequence selection policies**, which can be trained for quasi-Pareto-optimal code size, energy consumption, and execution time of target applications running on a target platform.

The paper makes the following additional *novel contributions*:

- **testing environment** to collect static and dynamic features of target applications on a target platform;
- **framework to adapt performance models** by analyzing and modeling energy consumption and execution time based on the collected code features; and
- **training framework** for RL-based adaptation of phase selection policies with respect to estimated dynamic features, and utilizing trained policies in LLVM.

After presenting a brief overview of the required background knowledge in Section II, we explain our novel *MLComp* methodology in Section III. Experimental setup is explained in Section IV and followed by our results in Section V. Conclusion is drawn in Section VI.

II. Background

A compiler converts a given software code implemented in a high-level programming language into machine executable code. The compilation flow is divided into 3 main parts:

1) the *front end* transforms source code into the compiler's Intermediate Representation (IR);
2) the *middle end* performs analyses and transformations in IR to ensure quality and prepares for code generation;
3) the *back end* generates a target-specific executable from IR.

Optimizing transformations are performed in each stage. Optimizations in the front end are specific to the programming language, while those in the back end tune hardware-specific low-level details. We work with IR-level optimizations in the middle end as depicted in Fig. 1. Those optimizations are general and applied independently to the source language and target platform. However, they can affect the performance in different ways depending on the target, which calls for their adaptive selection and ordering.

Standard ML methods are available to solve different kinds of algorithmic and modeling problems. SL [24] is used to learn a model representation that can fit input data to output predictions. The model is trained in iterative passes. In each pass, the model predicts output for the given input data, and the results are used to update the model weights to return predictions closer to the correct ones. RL [25], on the other hand, uses a reward-based system to learn the operations which should be done from the current state of the system itself. The reward reflects how an operation contributes to reaching the objective.

Programs are represented by their characterising features for ML approaches. The main types of program features are:

- *source code features* that characterize application code in a programming language and IR during compilation;
- *graph-based features* that provide information about data and control dependencies during compilation;
- *dynamic features* that describe operational aspects in architecture-dependent or architecture-independent ways.

Architecture-dependent dynamic features such as execution time and energy consumption are objectives for compiler optimization, while static features describe programs during compilation [26, 27, 28]. Dynamic features are time-expensive to determine because they require direct execution of compiled programs. This problem can be circumvented through model-based prediction. The performance of ML models is improved by scaling and filtering the input features [11, 29], an example of which is Principal Component Analysis (PCA) [30, 31].

III. The MLComp Methodology

The flow of our methodology is depicted in Fig. 2 and discussed hereafter. The concept is based on two models:

1) *Performance Estimator (PE)* is a fast and efficient way of estimating dynamic features for a given application domain, which is represented by a set of *target applications*, and for a given *target platform*. It allows for accurate prediction adapted to the given domain faster than standard estimation methods.
2) *Phase Sequence Selector (PSS)* is based on a policy for selection and ordering of optimization phases, and supports RL-based adaptation for different *target applications* and *target platforms*. Deploying PSS reduces development cost and realizes faster time-to-market by relieving performance engineering from the details of phase selection and ordering.

A. Data Extraction

Data Extraction is the first step of the methodology. We collect a training dataset for the PE model using the flow depicted in

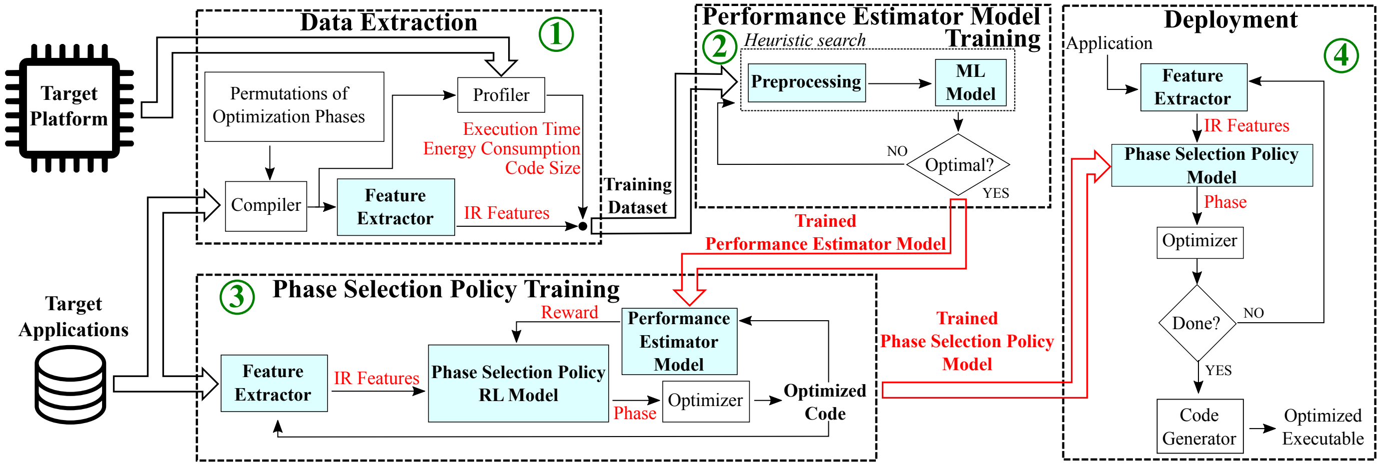

Fig. 2: The *MLComp* methodology trains and utilizes Performance Estimator and Phase Sequence Selector in four steps.

box ① in Fig. 2. Data is extracted for a target platform from a set of target applications by exploring different permutations of optimization phases. Permutations increase the number of data points as differently optimized variants of programs. For each combination of permutations and applications, the corresponding optimized code is compiled and its features are collected. We extract IR features that are similar to those of Milepost GCC [9], such as IR instruction counts, data and control dependencies, loop hierarchies, and call chains. Our tool also extracts platform-specific instruction counts from generated code for PE training. Dynamic features such as execution time, energy consumption, and code size are obtained via profiling the compiled code. All features are collected in a dataset that is used for training the PE model. The size of the dataset depends on the specific set of optimizations and the target applications.

B. Performance Estimator (PE)

Algorithm 1 Model search for fitting the Performance Estimator model

1: **procedure** ModelSearch($input$, $accuracy_{thr}$, $list_{models}$)
2: ▷ We initialize $accuracy_{best}$ to the worst case value, which is $-\infty$
3: ▷ Higher $accuracy$ is better
4: ▷ $model_{best}$ is initialized to a dummy one, returning a random value
5: init $accuracy_{best}$, $model_{best}$
6: ▷ split $input$ into $training$ and $testing$ data
7: $training$, $testing$ ← split($input$)
8: ▷ We cycle through all the models
9: **for** $model$ in $list_{models}$ **do**
10: ▷ After training the $model$, we test it and check its accuracy
11: train($model$, $training$)
12: $accuracy$ ← test($model$, $testing$)
13: **if** $accuracy > accuracy_{best}$ **then**
14: $accuracy_{best}$ ← $accuracy$
15: $model_{best}$ ← $model$
16: **end if**
17: **if** $accuracy_{best} > accuracy_{thr}$ **then**
18: **break for loop**
19: **end if**
20: **end for**
21: **return** $model_{best}$, $accuracy_{best}$
22: **end procedure**

The next step is the *Performance Estimator Model Training* in box ② in Fig. 2. We search for the preprocessing method and ML model that fits the best to the profiling data based on the code features. The list of methods and models to search is given as input. The search process is detailed in Algorithm 1. Tables III and IV present the preprocessing methods and ML models used by our PE modeling in this paper. The set of output metrics is completely customizable. As a training dataset is collected for one target platform, the PE model is to be trained for each target platform separately to achieve high accuracy. The trained PE model is used in later steps to predict a program's dynamic features from its IR features.

C. Phase Selection Policy

Algorithm 2 Training the Phase Selection Policy

1: **procedure** TrainPolicy($programs$, $num_episodes$, $batch_size$, $learning_rate$)
2: ▷ Initialize $policy$ to a random one
3: init $policy$
4: ▷ Perform training episodes in batches
5: $episode_count \leftarrow 0$
6: **while** $episode_count < num_episodes$ **do**
7: ▷ Run episodes and then update $policy$
8: $list_{episodes}$ ← init($batch_size$, $programs$)
9: $list_{results}$ ← run($list_{episodes}$, $policy$)
10: $policy$ ← optimize($policy$, $learning_rate$, $list_{results}$)
11: $episode_count \leftarrow episode_count + batch_size$
12: **end while**
13: **return** $policy$
14: **end procedure**

The trained PE model is used for the *Phase Selection Policy Training* in box ③ in Fig. 2. We use RL to train the policy that selects the best optimization phase to apply to a program characterized by its IR features, and thereby enables an efficient phase sequence to be created iteratively. The training is done in batches of episodes as listed in Algorithm 2. The policy is optimized using the REINFORCE *policy gradient method* [32, 33]. The training algorithm creates a phase sequence for a randomly selected target application in each episode with the current policy as depicted in box ③ in Fig. 2. The reward in each iteration of an episode reflects how well the last phase changed the dynamic features. Furthermore, the reward guides the training to Pareto-optimal outcomes by penalizing any degradation of the dynamic features. Accumulating rewards over an episode gives a discounted reward, which indicates the overall fitness of the policy for creating a Pareto-optimal phase sequence with respect to final dynamic features. At the end of each batch, the policy is updated according to the episodes' discounted rewards and corresponding phase sequences. The policy is trained with a given

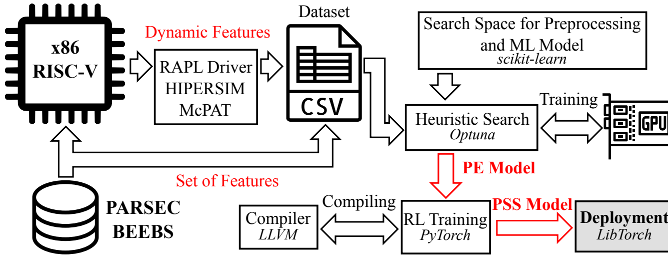

Fig. 3: Detailed toolflow for *MLComp* experimental setup. The process is repeated for each pair of target platform and applications, and it can be adapted for other pairs with minimal code changes.

TABLE III: Preprocessing algorithms evaluated during PE training. The list can easily be expanded, as our framework is customizable with different libraries.

Preprocessing Algorithms		
PCA	Kernel PCA	NCA
Mean-Std Scaling	Min-Max Scaling	Max-Abs Scaling
Robust Scaling	Power Transformer	Quantile Transformer

PE and a set of programs that represent a target platform and an application domain, respectively. The training time is reduced compared to other methods by using PE for fast estimation of dynamic features. Phase Selection Policy is the model used in the PSS.

D. Phase Sequence Selection (PSS)

The last step of *MLComp* is *Deployment* in box ④ in Fig. 2, which is the PSS utilizing a trained Phase Selection Policy. We apply the PSS model to drive a compiler's optimizer by selecting phases one after the other. The policy predicts how probable it is that a phase improves dynamic features of the program and accordingly the phase with the highest probability is applied. In case the selected phase did not change the program, which might happen because of the uncertainty of the selection, the best predicted phase remains the same for the next iteration. PSS overcomes that situation by applying the second best, the third best, and so on until a predefined limit, which is "Max. inactive subsequence length" in Table V. Phase selection ends when that limit is reached or when the total number of applied phases reaches a threshold.

Note, PSS does not require a PE model because the policy learns the platform-specific knowledge. Decoupling the PE and PSS models allows their separate training so that a platform vendor might provide a trained PE model for application developers, who can train a custom PSS model with a set of representative applications.

Although the PSS model is trained to reach the Pareto-front by selecting locally optimal phases, we observe that both the PE and PSS models have *approximation uncertainties* and true Pareto-optimality can not be guaranteed. The accuracy of PSS might be quantified by applying *probabilistic dominance* [34], which requires an in-depth empirical evaluation and statistical analysis beyond the scope of this paper. Our evaluation in Section V still shows quasi-Pareto-optimality of the results.

IV. Experimental Setup

A detailed toolflow for our experimental setup is shown in Fig. 3. We worked with two different target platforms: profiling for an

TABLE IV: ML models evaluated during PE training. The list can easily be expanded, as our framework is customizable with different libraries.

Machine Learning Regression Models		
Ridge	Kernel Ridge	Bayesian Ridge
Linear	SGD	Passive-Aggressive
ARD	Huber	Theil-Sen
LARS	Lasso	Lasso-LARS
Support Vector	Nu-Support Vector	Linear Support Vector
ElasticNet	Orthogonal Matching Pursuit	Multi-Layer Perceptron
Decision Tree	Extra Tree	Random Forest

TABLE V: Parameters of PSS training.

Parameter	Value	Parameter	Value
Number of layers	3	Size of inner layer	16
Number of episodes	512	Batch size	6
Max. phase sequence length	128	Learning Rate	0.1
Max. inactive subsequence length	8		

x86 target is done on an Intel Core i7 system using the RAPL [35] interface to measure power consumption, and dynamic features for a *RISC-V target* are obtained by accurate simulation with the industrial-grade simulator HIPERSIM [36] integrated with the open-source McPAT [37]. Programs are compiled with LLVM [6] version 9.0.0, which is able to target both platforms. The size of the collected dataset depends on the target applications and the set of optimization phases chosen at compile time: in this evaluation, we used between 200 and 600 data points for both the PARSEC benchmark [38] on *x86 target* and the BEEBS benchmark [39] on *RISC-V target*.

The training of the PE model is implemented in Python using Optuna [40], scikit-learn [41], and pandas [42]. It covers preprocessing methods and ML models listed in Tables III and IV, respectively. The set of output metrics has been chosen to analyze different patterns and distributions. Even though power consumption has a slight correlation with execution time, as in the number of cycles and the number of instructions [13], increasing the complexity of the system reduces the correlation of these metrics. Therefore, each of them is important for learning the dynamic behaviour of the system.

PSS training is implemented also in Python using PyTorch [43] for realizing the model. The 63 code features that our static analysis obtains are preprocessed by PCA with Maximum Likelihood Estimation (MLE) [31] before being passed to Deep RL [25]. The PSS model is trained with the parameters listed in Table V, applying optimization phases shown in Table VI. The

TABLE VI: LLVM optimization phases used for PSS evaluation. The list can easily be expanded. These phases are from optimization levels `-O3` and `-Oz`.

Optimization Phases		
adce	aggressive-instcombine	alignment-from-assumptions
argpromotion	bdce	called-value-propagation
callsite-splitting	constmerge	correlated-propagation
deadargelim	div-rem-pairs	dse
early-cse	early-cse-memssa	elim-avail-extern
float2int	globaldce	globalopt
globals-aa	gvn	indvars
inline	instcombine	instsimplify
ipsccp	jump-threading	licm
loop-deletion	loop-distribute	loop-idiom
loop-load-elim	loop-rotate	loop-sink
loop-unroll	loop-unswitch	loop-vectorize
lower-expect	mem2reg	memcpyopt
mldst-motion	prune-eh	reassociate
sccp	simplifycfg	slp-vectorizer
speculative-execution	sroa	tailcallelim

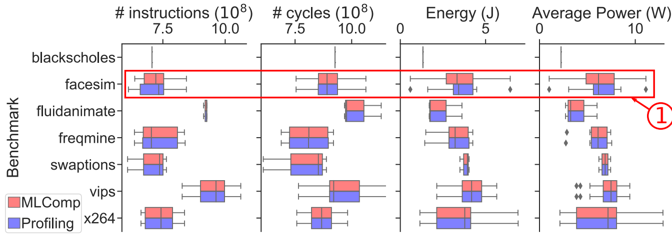

Fig. 4: Comparison between profiling data and prediction of a trained PE model for PARSEC benchmark applications on x86 platform shows very similar distributions, which supports the efficacy and accuracy of our model.

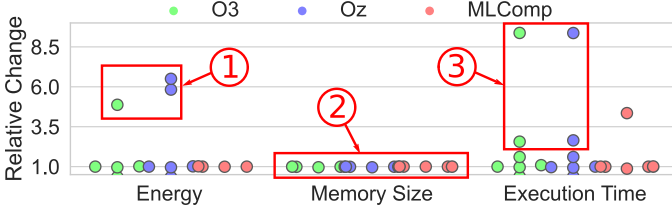

Fig. 5: PSS validation results for PARSEC applications on x86 platform. Values are relative to those of unoptimized code, the lower is the better.

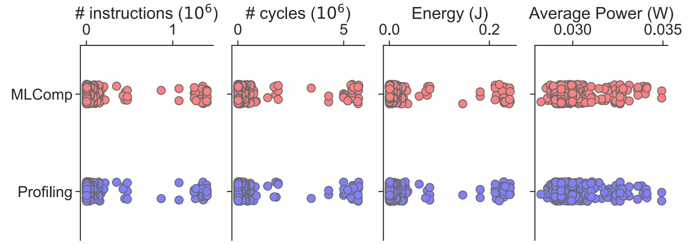

Fig. 6: Comparison between profiling data and prediction of a trained PE model for BEEBS benchmark applications on RISC-V platform.

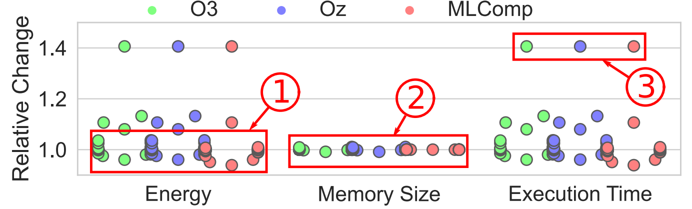

Fig. 7: PSS validation results for BEEBS applications on RISC-V platform. Values are relative to those of unoptimized code, the lower is the better.

trained model is stored in TorchScript format, to be loaded into and utilized by our custom LLVM optimization with LibTorch, the PyTorch C++ API.

Note that PE and PSS are independent of the target platform and the used application set. The necessary adaptations when changing the target platform are limited to adjusting target-specific compiler flags and utilizing a tool with support for gathering dynamic features inside the *Data Extraction* block. Furthermore, any application can be used with our training frameworks as long as it supports a build method using LLVM and allows controlling optimization phases via parameters or environment variables.

V. Evaluation

To evaluate our *MLComp* methodology, we trained and tested both PE and PSS models on different target platforms with different benchmarks as target applications.

A. PARSEC Benchmark Evaluation on x86 Platform

Here we focus our analysis on the PARSEC benchmark [38], running on an x86 platform. First, we gather the required dataset by profiling the execution of programs from the benchmark compiled with different optimization phases. Then, we use our framework to train different ML models and select the best one; the results are shown in Fig. 4. As we can see, the exact distributions and the ones generated by our PE model are almost identical for all the 4 metrics. Note that the *blackscholes* benchmark has a very tight distribution, while all the others have wider distributions. Referring to ①, we can see that the only visible difference resides in the *facesim* benchmark. However, there is always a high fidelity, as the error between the correct and the predicted distributions always has the same bias. This property is important for the training of the PSS model, giving the correct positive/negative reward to the current choice, even if a limited prediction error is present.

After validating the PE model, we used it to train the corresponding PSS model. In Fig. 5, we can see the result of the validation executed after the training. Specifically, distributions are pretty similar across standard state-of-the-art optimizations and *MLComp*. However, in some cases, as shown by ① and ③, some standard phase usage can increase both the energy consumption and the execution time between 8x and 10x, respectively, while *MLComp* shows slight improvements. Regarding memory size, as pointed by ②, there are minimal gains, which could be related to the benchmarks being synthetic applications.

B. BEEBS Benchmark Evaluation on RISC-V Platform

We performed a similar evaluation for BEEBS [39] on the RISC-V platform. In this case, the number of benchmarks is much higher compared to PARSEC, and since the PE results are similar to those with PARSEC, we show only an overview of the distribution points in Fig. 6.

This PE model was then used to train a PSS model, obtaining the results shown in Fig. 7. Here, at pointer ①, we can see that our *MLComp* performs better on average than standard state-of-the-art policies: reducing energy while also optimizing other objectives. Also with BEEBS, we can see that the memory size does not improve or worsen much, as pointed by ②. In addition, *MLComp* results in similar patterns of execution time and energy consumption. Focusing on pointer ③, we can see how our *MLComp* obtain more balanced results compared to standard state-of-the-art policies.

C. Discussion and Key Takeaways

Our PE model has a maximum percentage error *smaller than 2%* across all four metrics, while that of the comparable state of the art is in the range of 2%-7% on a single metric [16, 17, 18, 19, 20, 21]. Moreover, the efficient setup for data extraction and the heuristic search of models help us to reach higher accuracy with less time spent for acquiring data and training the model. Profiling the applications and training the models took *only 2 days*, compared to 15, 30 or 108 days [21, 19].

Drawing a straight comparison is more difficult for our PSS model, as related techniques optimize a single objective only. The state-of-the-art results oscillate between 5% and 30% improvement in execution time [8, 23], which makes our results fall in their average with up to 12% improvement in that metric.

However, our PSS model considers additional objectives and reaches up to 6% reduction in energy consumption while not increasing code size. There is actually a slight 0.1% improvement in the latter.

We can summarize the following key observations:

- The PE model realizes fast estimation with high accuracy, as it is capable of reproducing the profiled distributions.
- The PSS model performs better than standard optimizations on average and also provides quasi-optimal results for multiple objectives.
- Our *MLComp* methodology is fully automated and is usable with different target platforms and applications, enabling for fast estimation and optimization without manual analysis and modeling required.

VI. Conclusion

We propose the *MLComp* methodology to overcome limitations of current solutions in compiler optimization phase sequencing and performance modeling. State-of-the-art optimizers can be applied for different target platforms and applications case by case, but their adaptation is expensive and they typically optimize one metric only. *MLComp* supports adaptive selection of Pareto-optimal phase sequences with respect to execution time, power consumption, and code size by a Phase Sequence Selector (PSS) with an RL-based policy. Fast adaptation of the policy for different target platforms and application domains is enabled by an ML-based Performance Estimator (PE) model, which provides fast-yet-accurate prediction of dynamic program features. The PE model is trained for a target platform by automatically selecting the most suitable data preprocessing method and ML model for accurate prediction. This is a novel contribution in performance modeling as current solutions require manual analysis and modeling. Experiments with LLVM on the x86 and RISC-V platforms show that our methodology is efficiently reusable with different target platforms and applications. The PE model realizes fast estimation with very high accuracy, and the PSS model performs better than state-of-the-art optimizations with multiple objectives.

Acknowledgment

The presented work has been conducted in the "Cost Efficient Smart System Software Synthesis - COGUTS II (Code Generation for Ultra-Thin Systems)" project, funded and supported by the Austrian Research Promotion Agency (FFG) under grant agreement 872663, and affiliated under the umbrella of the EU Eureka R&D&I ITEA3 "COMPACT" Cluster programme.

References

[1] L. Atzori et al. "The Internet of Things: A Survey". In: *Comput. Networks* 54.15 (2010), pp. 2787–2805.

[2] F. Samie et al. "IoT Technologies for Embedded Computing: A Survey". In: *Proc. Elev. IEEE/ACM/IFIP Int. Conf. Hardware/Software Codesign Syst. Synth.* New York, New York, USA: ACM Press, 2016, 8:1–8:10.

[3] S. Liu et al. "Computer Architectures for Autonomous Driving". In: *Computer (Long. Beach. Calif).* 50.8 (2017), pp. 18–25.

[4] M. Yaqoob et al. "Control of Robotic Arm Manipulator with Haptic Feedback Using Programmable System on Chip". In: *2014 Int. Conf. Robot. Emerg. Allied Technol. Eng.* IEEE, 2014, pp. 300–305.

[5] B. Massot et al. "A Wearable, Low-Power, Health-Monitoring Instrumentation Based on a Programmable System-on-chipTM". In: *2009 Annu. Int. Conf. IEEE Eng. Med. Biol. Soc.* IEEE, 2009, pp. 4852–4855.

[6] C. Lattner et al. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Int. Symp. Code Gener. Optim. 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[7] R. M. Stallman et al. *Using the GNU Compiler Collection*. 2020, p. 1004.

[8] A. H. Ashouri et al. "MiCOMP: Mitigating the Compiler Phase-Ordering Problem Using Optimization Sub-Sequences and Machine Learning". In: *ACM Trans. Archit. Code Optim.* 14.3 (2017), pp. 1–28.

[9] G. Fursin et al. "Milepost GCC: Machine Learning Enabled Self-Tuning Compiler". In: *Int. J. Parallel Program.* 39.3 (2011), pp. 296–327.

[10] S. Kulkarni et al. "Mitigating the Compiler Optimization Phase-Ordering Problem Using Machine Learning". In: *ACM SIGPLAN Not.* 47.10 (2012), pp. 147–162.

[11] A. H. Ashouri et al. "A Survey on Compiler Autotuning Using Machine Learning". In: *ACM Comput. Surv.* 51.5 (Sept. 18, 2018), 96:1–96:42.

[12] C. Blackmore et al. "Automatically Tuning the GCC Compiler to Optimize the Performance of Applications Running on Embedded Systems". In: *CoRR* abs/1703.0 (2017), pp. 1–10. arXiv: 1703.08228.

[13] J. Pallister et al. "Identifying Compiler Options to Minimize Energy Consumption for Embedded Platforms". In: *Comput. J.* 58.1 (2015), pp. 95–109. arXiv: 1303.6485.

[14] A. H. Ashouri et al. "COBAYN: Compiler Autotuning Framework Using Bayesian Networks". In: *ACM Trans. Archit. Code Optim.* 13.2 (2016), 21:1–21:25.

[15] L. Almagor et al. "Finding Effective Compilation Sequences". In: *ACM SIGPLAN Not.* 39.7 (2004), pp. 231–239.

[16] T. Diop et al. "Power Modeling for Heterogeneous Processors". In: *Proceedings of Workshop on General Purpose Processing Using GPUs*. GPGPU-7. New York, NY, USA: Association for Computing Machinery, Mar. 1, 2014, pp. 90–98.

[17] P. Balaprakash et al. "AutoMOMML: Automatic Multi-Objective Modeling with Machine Learning". In: *High Performance Computing*. Ed. by J. M. Kunkel et al. Vol. 9697. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 219–239.

[18] B. Li et al. "Accurate and Efficient Processor Performance Prediction via Regression Tree Based Modeling". In: *J. Syst. Archit.* 55.10-12 (Oct. 1, 2009), pp. 457–467.

[19] A. Bona et al. "Energy Estimation and Optimization of Embedded VLIW Processors Based on Instruction Clustering". In: *Proceedings of the 39th Annual Design Automation Conference*. DAC '02. New York, NY, USA: Association for Computing Machinery, June 10, 2002, pp. 886–891.

[20] S. Van den Steen et al. "Analytical Processor Performance and Power Modeling Using Micro-Architecture Independent Characteristics". In: *IEEE Trans. Comput.* (2016), pp. 1–1.

[21] J. Laurent et al. "Functional Level Power Analysis: An Efficient Approach for Modeling the Power Consumption of Complex Processors". In: *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*. DATE '04. USA: IEEE Computer Society, Feb. 16, 2004, p. 10666.

[22] A. H. Ashouri et al. "A Bayesian Network Approach for Compiler Auto-Tuning for Embedded Processors". In: *2014 IEEE 12th Symp. Embed. Syst. Real-Time Multimed.* IEEE, 2014, pp. 90–97.

[23] A. H. Ashouri et al. "Predictive Modeling Methodology for Compiler Phase-Ordering". In: *Proc. 7th Work. Parallel Program. Run-Time Manag. Tech. Many-Core Archit. 5th Work. Des. Tools Archit. Multicore Embed. Comput. Platforms*. New York, New York, USA: ACM Press, 2016, pp. 7–12.

[24] O. Simeone. "A Brief Introduction to Machine Learning for Engineers". In: *Found. Trends® Signal Process.* 12.3-4 (2018), pp. 200–431.

[25] V. François-Lavet et al. "An Introduction to Deep Reinforcement Learning". In: *Found. Trends® Mach. Learn.* 11.3-4 (2018), pp. 219–354.

[26] F. E. Allen. "Control Flow Analysis". In: *ACM SIGPLAN Not.* 5.7 (1970), pp. 1–19.

[27] P. B. Schneck. "A Survey of Compiler Optimization Techniques". In: *Proc. ACM Annu. Conf.* New York, New York, USA: ACM Press, 1973, pp. 106–113.

[28] J. Ferrante et al. "The Program Dependence Graph and Its Use in Optimization". In: *ACM Trans. Program. Lang. Syst.* 9.3 (1987), pp. 319–349.

[29] Z. Wang et al. "Machine Learning in Compiler Optimization". In: *Proc. IEEE* 106.11 (2018), pp. 1879–1901. arXiv: 1805.03441.

[30] W. K. Härdle et al. "Principal Components Analysis". In: *Applied Multivariate Statistical Analysis*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 319–358.

[31] T. P. Minka. "Automatic Choice of Dimensionality for PCA". In: *Proceedings of the 13th International Conference on Neural Information Processing Systems*. NIPS'00. Denver, CO: MIT Press, 2000, pp. 577–583.

[32] R. J. Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Mach. Learn.* 8.3 (May 1992), pp. 229–256.

[33] R. S. Sutton et al. "Policy Gradient Methods for Reinforcement Learning with Function Approximation". In: *Proc. 12th Int. Conf. Neural Inf. Process. Syst.* MIT Press, 1999, pp. 1057–1063.

[34] F. Khosravi et al. "Efficient Computation of Probabilistic Dominance in Robust Multi-Objective Optimization". In: *CoRR* abs/1910.0 (2019), pp. 1–30. arXiv: 1910.08413.

[35] H. David et al. "RAPL: Memory Power Estimation and Capping". In: *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*. ISLPED '10. New York, NY, USA: Association for Computing Machinery, Aug. 18, 2010, pp. 189–194.

[36] ABIX GmbH. *HIPERSIM*. URL: https://a-bix.com/about.html.

[37] S. Li et al. "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures". In: *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 42. New York, New York: Association for Computing Machinery, 2009, pp. 469–480.

[38] C. Bienia et al. "The PARSEC benchmark suite: Characterization and architectural implications". In: *Proc. Int. Conf. Parallel Archit. Compil. Tech.* January (2008), pp. 72–81.

[39] J. Pallister et al. "BEEBS: Open Benchmarks for Energy Measurements on Embedded Platforms". In: *arXiv* abs/1308.5174 (2013), pp. 1–12.

[40] T. Akiba et al. "Optuna: A next-Generation Hyperparameter Optimization Framework". In: *Proceedings of the 25rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2019.

[41] F. Pedregosa et al. "Scikit-Learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[42] J. Reback et al. *Pandas-Dev/Pandas: Pandas 1.1.2*. Version v1.1.2. Zenodo, Sept. 8, 2020.

[43] A. Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035.