Intelligent Prediction of Execution Times

Dirk Tetzlaff
Technische Universität Berlin,
Chair Software Engineering for Embedded Systems,
dirk.tetzlaff@tu-berlin.de

Sabine Glesner
Technische Universität Berlin,
Chair Software Engineering for Embedded Systems,
sabine.glesner@tu-berlin.de

***Abstract*—It is a major challenge in software engineering to statically analyze in advance the expectable run-time behavior of applications. The most needed information is the expected execution time of a function to determine its computational cost. In this paper, we present a sophisticated approach that solves this problem by utilizing *Machine Learning* (*ML*) techniques based on regression modeling to automatically derive precise predictions for this information. This enables to focus optimization efforts on the parts that are relevant for the resulting performance and to predict execution times of functions on different processing elements of a heterogeneous architecture. Among others, our approach eliminates the need for manual annotations of run-time information, which automates and facilitates the development of complex software, thus improving the software engineering process. For our experiments that demonstrate the accuracy of our approach, we have used a considerable number of programs from various benchmark suites which encompass different real-world application domains. This shows on the one hand the general applicability and on the other hand the high scalability of our ML techniques.**

I. Introduction

As a matter of fact, the steadily increasing demand for higher performance rules all application domains. Often, program execution spends most of the time in a small fraction of code, a characteristic known as the "90-10 rule" – 90% of the execution time comes from 10% of the code. These code fractions, called *hot spots* of a program, are regions where optimization would be most beneficial to improve the run-time performance of programs. Hence, one has to predict the computational costs of functions to decide whether they are hot spots. Moreover, the cost must be *statically* predicted in advance at compile time without having information about the run-time behavior to apply hot spot optimizations as used in compilers. Since pure static analyses must consider all cases of possible run-time behavior, they may drastically over-approximate considered behavior and hence their predictions are imprecise. To overcome this, profiling and manual annotations of forecast values were proposed – both having certain drawbacks. Regarding profiling, its result must be fed back to an additionally needed compilation step after executions of the application that deliver the run-time behavior. Since profiling executes operations for analyzing and storing necessary run-time values to derive needed information, it may lead to a substantial execution time overhead. In general, it works well only when the actual run-time tendencies of a program match those collected during profiling [1]. That is, profiling is strongly dependent on the input data set of that profiling run. Regarding manual annotations, it is difficult and error-prone or often an impossible burden for the programmer since not all run-time information is knowable. Consequently, it is necessary to automatically incorporate knowledge of related dynamic run-time behavior into the compiler but without the need for manual annotations or profiling to preserve an automatic, continuous and efficient compilation flow. Furthermore, the technique should be highly scalable to facilitate the integration in industrial-strength compiler environments used for compilation of real-world applications.

We tackle the problem of incorporating statically unknown or imprecise information about the execution time of functions into the compilation flow with the use of ML techniques, especially regression modeling based on pure static code features. In a one-time-only *training phase* per architecture, which is decoupled from compilation, we automatically establish an ML model that relates static code features to dynamic run-time behavior. Then, we automatically generate from this model an executable heuristic that a compiler can use as predictor function for execution times of functions. On the long term, we aim at mapping concurrent applications to heterogeneous parallel architectures. To that end, we use the predicted execution times to improve the allocation to the *Processing Elements* (*PEs*) of the target architecture. Additionally, we use the learned iteration counts of loops and recursion frequencies of functions from our previous work as estimate for communication overheads if communication arises within loops or recursive functions. Integrating a beforehand machine learned heuristic into the compiler that holds observations from several profiling runs and relates them to static code features, as we do, eliminates the need for profiling at compile time and has the advantage to focus optimizations on more realistic behavior, not on one single behavior. To determine the accuracy of our approach, we evaluate experiments for learning the execution time in detail. This demonstrates that we are able to precisely predict the considered run-time behavior. Using 197 programs of 25 benchmark suites from different real-world application domains for our experiments shows the general applicability of our approach to a wide range of programs with different behavior.

In summary this paper shows that, as is the case in many other domains, programs can be successfully represented by static code features. These features can then be used to gauge their similarity and thus the applicability of previously learned off-line knowledge. This solves the problem of over-approximation, which is inherent to static program analyses. Hence, the optimization potential is increased which leads to a higher quality of delivered software. This paper is organized as follows: Our approach for machine learning based execution time prediction is given in Section II. We present its implementation and experimental results of our work in Section III. Then, we discuss related work in Section IV.

ISBN: 978-1-4673-5256-7/13/$31.00 ©2013 IEEE

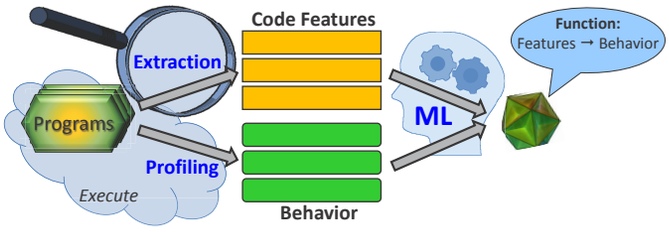

Fig. 1: Training phase

Finally, conclusions are drawn in Section V.

II. ML BASED EXECUTION TIME PREDICTION

We aim at statically predicting execution times of functions with *regression modeling*. This requires a one-time-only training phase per architecture to build the ML model. Since the training phase is decoupled from actual compilation, the compile time is not extended by training. Once trained, the model can be used from then on without modification to also predict execution times of unseen functions. In the training phase, we collect code features by means of static analysis as well as dynamic run-time behavior through several profiling runs from a comprehensive suite of programs. Both information is fed into our learning algorithm (see Fig. 1). The execution time of a function clearly depends on executed instructions of that function. Since we aim at predicting execution times of functions solely based on information that is available at compile time of applications, we cannot know which instructions are actually executed. The information we have is the *Intermediate Representation* (*IR*) obtainable from the source code, especially a *Control Flow Graph* (*CFG*) for each function. During code generation in the back end of the compiler, machine code to be emitted from IR is available. We consider features for learning execution times that represent these machine code instructions. Depending on the target architecture, the number of different machine code instructions that a processor can execute ranges from about ten to more than two hundred. If we chose to represent each different instruction by a different feature, we would generate a sparse feature vector in the latter case. We thus use equivalence classes of machine code instructions with respect to similar behavior to categorize instructions and we use these categories as features. The categorization is as follows (see Table I).

Tests (features 1 – 2): The feature *bitwise* represents instructions on bits, e. g., tests on bits at logical comparisons like conjunctions and disjunctions or operations on bits to perform logical negation. Instructions captured by the feature *comp* are relational operators that compare arithmetic values in integer or floating point representation.

Arithmetic operations (features 3 – 6): The feature *calc* represents calculations like address arithmetic or mathematical operations on integer data types. The feature *convert* captures conversions between integer data types such as sign extensions or zero extensions. Instructions represented by the feature *fcalc* are arithmetic operations with numbers in floating point representation. With the feature *fconvert*, we capture conversions from integer to floating point data types and vice versa.

TABLE I: Static code features for learning execution times

	No.	Name	Description
Tests	1	bitwise	operations on bits
	2	comp	comparison between data values
Arithmetic	3	calc	integer arithmetic calculations
	4	convert	conversion between integer data types
	5	fcalc	floating point arithmetic calculations
	6	fconvert	conversion between floating point and integer values
Control	7	docall	function calls
	8	param	push actual parameters to stack
	9	jump	jumps in the control flow
Copy	10	addr	load effective address
	11	load	load of values into registers
	12	move	move values between registers
	13	store	storage of values to memory

Control flow related (features 7 – 9): Function calls, which alter the control flow, are classified by the feature *docall*. For each function call, a different number of arguments must be given to the callee. We capture instructions that push data to the stack to pass parameters of function calls with the feature *param*. The feature *jump* represents jumps in the control flow, either conditional or unconditional.

Copy instructions (features 10 – 13): The feature *addr* captures instructions that calculate the effective address of objects and store it in a register. We represent instructions that load values from memory or from its spill location into registers with the feature *load*. The feature *move* represents instructions that move values between registers or between registers and spill locations. With the feature *store*, we captures instructions that store values from registers to memory.

Given the CFG of a function, we visit its *basic blocks* (*BBs*) and record categorized instructions within each BB in the feature vector. Since not all instructions are actually executed due to conditional jumps in the CFG, we do not simply count categorized instructions. Instead, we weight each occurrence with its probability of being executed that we compute according to the algorithm of Wu and Larus [2]. If another function is called by the function for which a prediction of its execution time is required, the executed instructions of this other function also contribute to the execution time. We therefore take account of all function calls. When we know the instructions within the function body of a callee (because this function is defined in the application under compilation), we do not increase the feature *docall*. Instead, we add the feature vector of the function to the one of the current function. Again, we weight the feature vector of the function with the execution probability of the function call before summing the features to the ones of the current function in order to regard that not all function calls are actually executed. Calls to functions where we cannot derive its executed instructions (e. g., functions defined in included libraries or recursive function invocations) are captured with the feature *docall*.

We assume that there exists a linear relationship between the amount of (classified) machine code instructions to be executed by the function and its execution time. Therefore, we consider *linear* regression modeling, for which several learning algorithms exist. We have evaluated different basic learning algorithms for regression modeling, namely *decision trees*, *k*-

ISBN: 978-1-4673-5256-7/13/$31.00 ©2013 IEEE

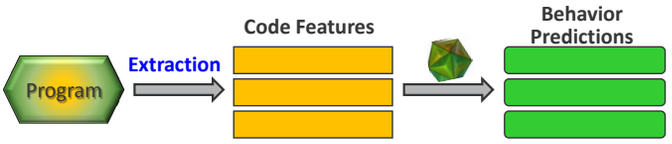

Fig. 2: ML-based behavior prediction

Nearest Neighbor (*k-NN*), *Ordinary Least Squares Estimation* (*OLS*), the *Iterated Reweighted Least Squares* (*IWLS*) method for robust linear modeling, and *Support Vector Machine* (*SVM*) (see [3] as reference). Additionally, we have adapted the *Predicting Query Run-time 2* (*PQR2*) technique of Matsunaga and Fortes [4] that is based on a decision tree (we discuss the original algorithm in Section IV). We exclusively use static code features of applications as input and we also have employed *Naïve Bayes* and *Random Forests* [5] as possible classifiers at inner nodes. Furthermore, instead of using only k-NN classifiers with $k = 1$ and $k = 3$ at inner nodes, we use a ten range grid search for the best k between $k = 1$ and $k = 10$. Our evaluation of all algorithms (see Section III) shows that it is advantageous to apply our adaption of the PQR2 algorithm for statically predicting execution times.

Once trained, the learned model can be used to predict execution times of new observations, solely based on their static features (see Fig. 2). We automatically generate an executable heuristics from this model. This provides the compiler with intelligence about the dynamic program behavior and can be used for a precise cost estimation. The idea of the heuristics is to focus on typical program behavior instead of considering all possibilities, as it is done by most static analyses.

III. Experimental Results

We have implemented our learning technique within the *R Project*, which is a collection of statistical functions that we have already successfully used in previous work. For implementing the compilation steps, we use the modular, industrial strength *Compiler Development System* (*CoSy*) of ACE [6]. We have implemented the automatic extraction of code features of functions and the static branch prediction algorithm proposed by Wu and Larus [2] for determining execution probabilities and frequencies. To determine the actual execution time for profiling during the training phase, we have developed and implemented an instrumentation algorithm within CoSy.

For our experiments, we have used 197 programs of 25 benchmark suites from different domains. Table II lists the benchmark suites with the number of its used programs (column #) and their descriptions. Similar programs included in different benchmark suites are taken only from one suite and are omitted from the other. We have analyzed in total 23037 functions, which are contained within our benchmark collection, and we have measured their execution time in nanoseconds. We have observed execution times that range from 49 nanoseconds to more that 120 minutes, with an average of 9 seconds. In Table III the observed execution times are shown in the first row, partitioned by their magnitude. As can be seen, most of the functions finish execution below one millisecond, but also a considerable number of functions

TABLE II: Serial benchmark suites

No.	Suite	#	Description
1	BioBench	1	Bioinformatics applications
2	bitbench	4	Bit stream benchmarks
3	cBench	18	Broad spectrum of applications
4	CSiBE	12	Code size benchmarks
5	FreeBench	3	Synthetic CPU-benchmarks
6	GCbench	1	Artificial garbage collection benchmark
7	Heaplayers	3	Memory-intensive benchmarks
8	LLCbench	2	Low-level characterization benchmarks
9	LLVM	37	LLVM testing infrastructure
10	McCat	3	Benchmarks for the McCat compiler
11	MediaBench	3	Multimedia applications
12	MediaBench II	7	Multimedia applications
13	MiBench	12	Embedded benchmark suite
14	NPB	8	Serial version derived V-2.4
15	Prolangs-C	6	Rutgers PROLANGS benchmarks
16	Ptrdist	1	Pointer-intensive benchmarks
17	Shoot	15	Revived version of Bagley's computer language shootout benchmarks
18	SPEC CPU95	5	Industry-standardized, compute-intensive benchmarks
19	SPEC CPU2000	3	
20	SPEC CPU2006	1	
21	Splash2	8	Serial version of the SPLASH-2 suite
22	SWEET WCET	29	Worst case execution time benchmarks
23	Trimaran	6	From Trimaran Consortium
24	UnixBench	1	Revised BYTE UNIX benchmark suite
25	Versabench	8	Benchmarks for flexible architectures
	Sum	197	

TABLE III: Execution times and mean absolute errors

	$\leq 1\mu s$	$\leq 1ms$	$\leq 1s$	$\leq 1min$	other
actual time	6610	13002	2266	978	181
dTree	0	0	387	22462	188
naïve	0	0	41	22831	165
k-NN	10529	7905	2789	1493	321
rlm	0	4558	15735	1518	226
lm	0	0	103	22605	329
SVM	0	0	21473	1341	223
adapted PQR2	10159	8396	2861	1351	270

take minutes to execute. For our experiments, we have applied the learned predictors to unseen data for evaluation (i. e., the benchmarks that the predictors where trained with and the benchmarks used for evaluation are disjoint). In particular, we have performed *leave-one-program-out cross-validation* for assessing how our results will generalize to an independent data set. That is, we remove all functions of a certain program from the training set, train the predictors with the remaining set, and predict the removed functions with the established predictors. This cross-validation is iteratively performed such that each program is left out once, which results in 197 experiments. As precision measure, we have used the average deviation between predictions and correct time (*mean absolute error*). Additionally, the *correlation* between predictions and correct times indicates whether a relationship was actually learned.

The results demonstrate very well the accuracy of our approach. Fig. 3a shows the mean and median absolute errors for the learning algorithms with their standard deviations as indicated by bars. The decision tree learning technique is la-

ISBN: 978-1-4673-5256-7/13/$31.00 ©2013 IEEE

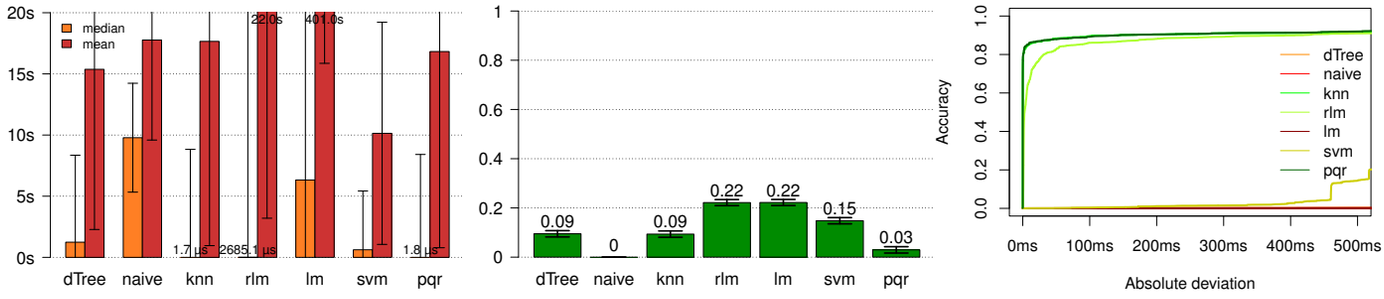

Fig. 3: Error metrics: (a) mean and median absolute errors, (b) correlation, and (c) regression error characteristic curves

beled with *dTree*, k-NN with *knn*, the IWLS method with *rlm*, OLS with *lm*, SVM with *svm*, and our adaption of the PQR2 algorithm with *pqr*. The label naïve denotes the prediction by the average. As can be seen, the mean errors are skewed compared to the median errors due to the great range of actual execution times. The SVM learning technique yields with about 10.1 seconds the lowest mean absolute error, followed by decision trees with 15.4 seconds and our adaption of the PQR2 algorithm with 16.8 seconds. In contrast, the median errors for the best three learning techniques are in orders of magnitude smaller. The k-NN technique yields the lowest median error with 1.7 microseconds, followed by our adaption of the PQR2 algorithm with 1.8 microseconds and IWLS with about 2.7 milliseconds. Fig. 3c shows the *regression error characteristic curves* [7] of the absolute deviation for our investigated ML techniques. This shows that our adapted PQR2 technique predicts more than 85% of the observations with an absolute deviation less than 250 milliseconds, which indicates that the mean error is skewed due to some large outliers in the predictions. The mean absolute errors partitioned by their magnitude that are shown in Table III also prove that the mean errors are skewed. Solely our adaption of the PQR2 algorithm and the SVM learning technique are able to predict execution times of nearly half of the functions with mean absolute errors below one microsecond. Furthermore, our approach has the most errors below one millisecond compared to all other techniques. Hence, the median error is more meaningful to asses the quality for learning execution times based on our collected observations. The correlation between predictions and correct times is shown in Fig. 3b. All correlations except the zero correlation of the naïve predictor are statistically significant at the probability level $p = 0.001$. That is, we have 99.9% evidence a relationship exists. For a more fine-grained view on the errors, we show the observed actual execution time of each function color coded in Fig. 4a. The 23037 functions are sorted by time in ascending order and the box is filled column-wise (i. e., the figure has no x- nor y-dimension). For the detailed view on mean absolute errors of leave-one-program-out cross-validation with different ML techniques, we have sorted the functions accordingly. The prediction errors for each function with SVM are shown in Fig. 4b, with k-NN in Fig. 4c, and with our adaption of the PQR2 algorithm in Fig. 4d. As can be seen, the two learning techniques that have a lower mean absolute error than our adaption of the PQR2 algorithm perform worse than

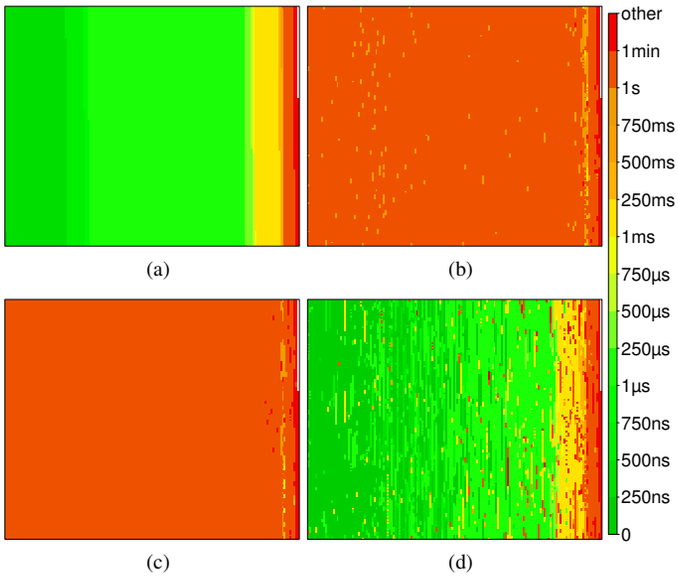

Fig. 4: Execution time (a) observed values – vs. prediction errors (b) with SVM, (c) with decision trees, (d) with our adaption of PQR2

our approach. Furthermore, our approach mainly yields large prediction errors at functions with large execution times. This demonstrates that our predictions are precise relative to the actual execution time.

Additionally to the experiments above, we have established ML models for different PEs to statically determine which PE will execute an application most efficiently. Again, we then have applied the learned predictors to unseen data for evaluation. In particular, we have trained the models with about two thirds of the functions and have applied the predictors to one third, namely to 7500 functions, which were sampled randomly. We have considered four different Intel® CPUs as PEs: Core i5-2400 @3.10GHz, Core 2 Duo E6850 @3.00GHz, Core 2 Duo E7500 @2.93GHz, and Xeon E5430 @2.66GHz. We have profiled the programs of our benchmark collection on each PE. In Table IV, we show a summary of observed execution times. As can be seen, the Core i5-2400 CPU performs best on average. Interestingly, this is not the case

ISBN: 978-1-4673-5256-7/13/$31.00 ©2013 IEEE

TABLE IV: Execution times in nanoseconds on different PEs and results for predicting the best PE

	Xeon E5430	i5-2400	2 Duo E7500	2 Duo E6850
Min.	3.170e+02	2.480e+02	2.900e+02	3.110e+02
1st Qu.	5.060e+02	3.840e+02	5.610e+02	5.680e+02
Median	2.897e+03	2.158e+03	2.926e+03	3.124e+03
Mean	1.439e+08	1.104e+08	1.323e+08	1.904e+08
3rd Qu.	1.136e+04	7.092e+03	1.016e+04	1.243e+04
Max.	1.101e+11	8.300e+10	1.003e+11	1.570e+11
Best PE	197	6977	95	231
# Predicted	600	5697	334	869
Exact predicted	18	5253	3	16

for all functions. The row *Best PE* in this table shows the number of functions that are executed most efficiently on the corresponding PE. Based on the observed execution times and the feature vector of each function, we establish four ML models with our adaption of the PQR2 algorithm. With these models, we obtain execution time predictions for each PE from a feature vector. Then, we determine the PE that corresponds to the lowest prediction. Using this scheme with the feature vector of the `main` function of an application hence predict the best performing PE for this application.

Again, our experimental results are very precise. For 5290 of the 7500 functions, we have correctly predicted the PE that executes a function most efficiently. Note that our prediction is not the trivial one, i. e., we have not always predicted the Core i5-2400 CPU as the best. In row *# Predicted* of Table IV, we show the number of predictions that rate the corresponding PE as the best. The row *Exact predicted* presents the number of these predictions that are correct (which sum up to 5290). That is, we are able to exactly predict the PE that executes a function most efficiently for 70.5% of the functions. For 644 of the functions (i. e., 8.6%), we have predicted the second best performing PE to be the best, for 813 (i. e., 10.9%) the third best performing PE as the best, and for 753 (i. e., 10%) the fourth best as the PE that executes the functions most efficiently. Given a heterogeneous architecture, a run-time scheduler do not know in general on which PE an application will execute most efficiently and thus can allocate it everywhere. Hence, we can asses the resulting benefit when an application is allocated to the PE that our predictor rates as the best. The mean benefit of an application is the difference between the execution time of the `main` function on our predicted PE and the mean of its execution times on all PEs. Similarly, the maximal benefit our approach is the difference to the maximum of its execution times on all PEs. To determine this, we have trained the models with all but the `main` functions and have applied the learned predictors to the `main` functions. The average of the mean execution times of the `main` functions on all PEs are 66.7 seconds, the average of the maximal execution times are 87.6 seconds, and the average of the execution times on our predicted PEs are 52.7 seconds. Hence, using our approach yields a mean execution time reduction of 21% and a maximal execution time reduction of 40%. In conclusion, our approach is best suited to predict execution times based on static code features and, based on this, to predict the PE that executes an application most efficiently.

IV. Related Work

In this section, we discuss related approaches that aim at predicting execution times. The approach of Iverson et al. [8] predicts the execution time as a function of a vector of parameters that holds input and performance data. Input data is the argument an application is invoked with. Performance data refers to varying execution times of the same application on several target architectures. That is, predicting execution times for new applications with variant arguments requires to adapt the parameter vector, which is a major drawback because an expert has to do this manually. Giusto et al. [9] have proposed a method for estimating execution times, which can be used to speed up simulation time. Their method first translates actually executed instructions of a program into a set of 25 *virtual instructions*. Then, an estimate for the execution time in terms of processor cycles is given by a predictor equation $Cycles = K + \sum_i P_i * N_i$ where N_i is equal to the number of executed virtual instructions of type i, P_i is the cycle count of instruction i, and K is the intercept. For solving the equation, P_i can be determined using linear regression and expert knowledge that the authors called "users intuition". This intuition must be used to remove certain instructions i from the predictor equation to obtain better results. Close to the work discussed above, the method of Bontempi and Kruijtzer [10] is also based on regression modeling with a set of virtual instructions and the number of times each instruction is executed as features. Their method extends linear to nonlinear regression techniques, called *lazy learning*, and additionally includes architectural parameters in the feature vector to predict execution cycles of applications. Oyamada et al. [11] propose to use *neural networks* for estimating execution cycles. They classify instructions into five classes that capture possible behavior: integer calculations, floating-point calculations, backward and forward branches in the CFG, and memory accesses. As with the approaches before, their method requires the number of times of executions of these classified instructions to be known. In contrast, our technique does not require to profile each application since our learned predictor for the execution time is statically applicable.

The proposed method of Gupta et al. [12] also predicts execution times in the form of time ranges with a prediction model based on *classification trees*, which they called *Predicting Query Run-time* (*PQR*) trees. Predictions are made for queries on enterprise data warehouses to improve its workload management. As features to their learning algorithm, they compute some derived attributes of the query plan and multiply them by the number of current queries running on the grid. Different from original classification trees where the classes (or time ranges) are known prior to the creation of the tree, at every node of the PQR tree a classifier is determined that predicts the two subranges of the current range for the children. As the PQR tree grows, the classes are automatically obtained by a combination of a classifier and two subranges at every node that gives the highest accuracy, i. e., the greatest fraction of correctly classified queries. As possible classifiers for the nodes of the PQR tree, the authors have used k-NN classifiers with $k = 1$ and $k = 3$, and *classification trees* with six different model parameters. Matsunaga and Fortes [4] have extended the PQR tree approach of Gupta et al. [12] to the regression problem by allowing the leaves of the tree to select the best regression method from a pool of basic regression

algorithms. The pool used in their experiments comprises *linear regression* and SVM models. For each application, they have used different features holding properties about the target architecture and arguments given to the application at run-time as input to their learning algorithm. Hence, their approach is application dependent and must be configured by an application expert to allow for prediction with new applications. As we have described in Section II, we have adapted the PQR2 approach of Matsunaga and Fortes [4] to our work.

In summary, there is no ML approach that predicts the execution time solely based on static code features. There exist recent approaches for *worst-case execution time* (*WCET*) analysis based on *Abstract Interpretation* (*AI*), such as [13], [14], [15], [16], [17]. However, AI might slow down the analysis such that it becomes impractical. In particular, this can be observed for the analysis of program loops with high iteration counts for which each loop iteration is separately simulated [18]. Recently, a WCET analysis based on AI and *Integer Linear Programming* (*ILP*) is proposed by Chattopadhyay et al. [19]. However, additionally to the drawbacks of AI, ILP may suffer from the explosion in the number of generated ILP constraints. Their analysis takes at most about 300 seconds, with an average of 20 to 30 seconds over all programs, which is unacceptable for compilation of real-world concurrent applications.

V. Conclusion

In this paper, we have presented a sophisticated ML approach for predicting execution times of functions. To our knowledge, it is the only ML approach that predicts execution times solely based on static code features. We have improved the PQR2 algorithm originally developed by Matsunaga and Fortes [4] and we have adapted this technique to our static setting. This relates static code features of functions to their expectable execution time. Because each program is profiled multiple times with different input data during training, we collect realistic behavior. The needed one-time-only training phase is decoupled from actual compilation of programs. Hence, the compile time is not increased by the learning technique. From our established ML model, we automatically generate an executable heuristics that can be used as predictor function for a precise cost estimation. Since the heuristics is automatically generated and incorporated into the compiler without the need for user intervention, our approach preserves an automatic and continuous compilation flow.

Our experimental results demonstrate that we are able to predict execution times of more than 85% of the functions with an absolute error below 250 milliseconds. Our adaption of the PQR2 algorithm is the technique with the most errors below one millisecond compared to other learning techniques. Compared to the techniques that also yield errors below one millisecond, our approach has the lowest mean absolute error. Furthermore, we are able to exactly predict the PE of a heterogeneous architecture that executes a function most efficiently for more than 70% of the functions. Using our approach thus yields a mean execution time reduction of 21% for the investigated benchmarks. In conclusion, our approach eliminates the need for profiling at compile time or manual annotations during software engineering, nevertheless providing the compiler with intelligence about the dynamic program behavior.

References

[1] M. D. Smith, "Overcoming the challenges to feedback-directed optimization (keynote talk)," in *Proc. of DYNAMO*. ACM, 2000, pp. 1–11.

[2] Y. Wu and J. R. Larus, "Static branch frequency and program profile analysis," in *MICRO 27*. ACM Press, 1994, pp. 1–11.

[3] E. Alpaydin, *Introduction to Machine Learning*, 2nd ed., ser. Adaptive Computation and Machine Learning. The MIT Press, 2010.

[4] A. Matsunaga and J. Fortes, "On the use of machine learning to predict the time and resources consumed by applications," in *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, ser. CCGRID'10. Washington, DC, USA: IEEE Computer Society, May 2010, pp. 495–504.

[5] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[6] ACE, *Associated Compiler Experts bv., Amsterdam, The Netherlands*, 2013, `http://www.ace.nl`.

[7] J. Bi and K. P. Bennett, "Regression error characteristic curves," in *Proceedings of the 20th International Conference on Machine Learning (ICML)*, T. Fawcett and N. Mishra, Eds., 2003, pp. 43–50.

[8] M. A. Iverson, F. Özgüner, and L. C. Potter, "Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment," *IEEE Transactions on Computers*, vol. 48, no. 12, pp. 1374–1379, December 1999.

[9] P. Giusto, G. Martin, and E. Harcourt, "Reliable estimation of execution time of embedded software," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE'01. Piscataway, NJ, USA: IEEE Press, 2001, pp. 580–589.

[10] G. Bontempi and W. Kruijtzer, "A data analysis method for software performance prediction," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE'02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 971–976.

[11] M. S. Oyamada, F. Zschornack, and F. R. Wagner, "Accurate software performance estimation using domain classification and neural networks," in *Proceedings of the 17th Symposium on Integrated Circuits and System Design*, ser. SBCCI'04. New York, NY, USA: ACM, September 2004, pp. 175–180.

[12] C. Gupta, A. Mehta, and U. Dayal, "PQR: Predicting query execution times for autonomous workload management," in *Proceedings of the 2008 International Conference on Autonomic Computing*, ser. ICAC'08. Washington, DC, USA: IEEE Computer Society, June 2008, pp. 13–22.

[13] M. Lv, W. Yi, N. Guan, and G. Yu, "Combining abstract interpretation with model checking for timing analysis of multicore software," in *Proceedings of RTSS'10*. IEEE, 2010, pp. 339–349.

[14] A. Gustavsson, J. Gustafsson, and B. Lisper, "Toward static timing analysis of parallel software," in *WCET'12*, ser. OASIcs, T. Vardanega, Ed., vol. 23, 2012, pp. 38–47.

[15] S. Metzlaff and T. Ungerer, "Impact of instruction cache and different instruction scratchpads on the WCET estimate," in *Proc. of HPCC-ICESS*, 2012, pp. 1442–1449.

[16] V. Rodrigues, B. Åkesson, S. P. M. de Sousa, and M. Florido, "A declarative compositional timing analysis for multicores using the latency-rate abstraction," LIACC, Faculty of Computer Science, University of Porto, Technical report CISTER-TR-130108, 2013.

[17] S. Chattopadhyay and A. Roychoudhury, "Scalable and precise refinement of cache timing analysis via path-sensitive verification," *Real-Time Systems*, pp. 1–46, 2013.

[18] P. Lokuciejewski, D. Cordes, H. Falk, and P. Marwedel, "A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models," in *Proc. of CGO'09*. Seattle, Washington, USA: IEEE Computer Society, March 2009, pp. 136–146.

[19] S. Chattopadhyay, C. L. Kee, A. Roychoudhury, T. Kelter, P. Marwedel, and H. Falk, "A unified WCET analysis framework for multi-core platforms," in *Proceedings of the IEEE 18th Real-Time and Embedded Technology and Applications Symposium*, ser. RTAS'12. IEEE, April 2012, pp. 99–108.

ISBN: 978-1-4673-5256-7/13/$31.00 ©2013 IEEE