Automatic Selection of Machine Learning Models for Compiler Heuristic Generation *

Paul Lokuciejewski[1], Marco Stolpe[2], Katharina Morik[2], Peter Marwedel[1]

[1] Computer Science 12 (Embedded Systems Group)
[2] Computer Science 8 (Artificial Intelligence Group)
TU Dortmund University
D-44221 Dortmund, Germany
FirstName.LastName@tu-dortmund.de

Abstract. Machine learning has shown its capabilities for an automatic generation of heuristics used by optimizing compilers. The advantages of these heuristics are that they can be easily adopted to a new environment and in some cases outperform hand-crafted compiler optimizations. However, this approach shifts the effort from manual heuristic tuning to the model selection problem of machine learning – i. e., selecting learning algorithms and their respective parameters – which is a tedious task in its own right.
In this paper, we tackle the model selection problem in a systematic way. As our experiments show, the right choice of a learning algorithm and its parameters can significantly affect the quality of the generated heuristics. We present a generic framework integrating machine learning into a compiler to enable an automatic search for the best learning algorithm. To find good settings for the learner parameters within the large search space, optimizations based on evolutionary algorithms are applied. In contrast to the majority of other approaches aiming at a reduction of the average-case execution time (ACET), our goal is the minimization of the worst-case execution time (WCET) which is a key parameter for embedded systems acting as real-time systems. A careful case study on the heuristic generation for the well-known optimization *loop invariant code motion* shows the challenges and benefits of our methods.

1 Introduction

Optimizing compilers transform a program written in a source language into a semantically equivalent program in a target language. The generated code should exhibit a high performance. Since finding optimal solutions to compiler optimizations is provably hard, compiler writers are forced to use heuristics as approximate solutions. The development of heuristics for compiler optimizations is a tedious task requiring both a high amount of expertise and an extensive trial-and-error tuning. The reasons are twofold. First, heuristics often use simplified architecture models of complex systems, which do not sufficiently capture all relevant architectural features. Second, compiler optimizations are typically executed within a sequence of interfering optimizations. Since the mutual interactions are hardly predictable, compiler writers develop heuristics based on conservative assumptions. Such heuristics avoid negative effects but also prevent the exploration of the optimization potential.

* The research leading to these results has received funding from the European Community's Artist Design Network of Excellence and from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement n° 216008.

Machine learning (ML) techniques have recently raised considerable research interest in the compiler community since they can help to automatically find good optimization heuristics. Given a set of characteristics (called *static features*) about the code to be optimized, machine learning tools automatically learn a mapping from these features to heuristic parameters. For today's rapidly evolving processor market, these machine learning based (MLB) heuristics offer two advantages. First, they often outperform hand-crafted heuristics [21]. Second, they can be automatically adopted to new environments.

A central questions in the heuristic generation is that of *model selection* which covers the choice of the learning algorithm, its parameters, and the features. Over the last decades, a vast spectrum of different machine learning algorithms was developed. The learner selection for the generation of high-performance heuristics is not trivial and becomes even more complicated since most learners are equipped with numerous parameters considerably affecting the learner's behavior. The consequence is that typically one or two learners are applied using standard parameter settings [18, 15, 6]. However, this approach does not exploit the full learners' potential and possibly misses optimization opportunities.

In this paper, we systematically explore the performance of different learners and their parameters for compiler heuristic generation. We use the open-source machine learning tool RapidMiner [17]. It includes not only a large number of learning algorithms, evaluation procedures, and feature transformation operators, but also operators for self-optimization regarding, e. g., parameter settings of learning algorithms. Since the true quality of learners depends on the quality of their predictions [12], the considered learners are *directly* involved in the model selection run on real-life benchmarks. Using this approach allows to find the best learner with the highest performance increase for a particular optimization.

While machine learning was studied in the past in the context of ACET minimization, this work focuses on embedded systems acting as hard real-time systems. Besides efficiency requirements, these systems are characterized by their critical timing constraints which are expressed by the worst-case execution time. Especially for safety-critical application domains, such as automotive and avionics, the satisfaction of the WCET must be guaranteed to avoid system failure. Thus, we concentrate on an automatic generation of MLB heuristics that promise the highest WCET improvement. The main contributions of this paper are as follows:

1. For the first time, we address the well-known problem of selecting an appropriate learning algorithm for the generation of optimization heuristics [20] in a systematic way.
2. We evaluate six popular learning algorithms. The study indicates that different learners and their parameter settings significantly affect the program performance.
3. Since the search space for the learners is typically too large for an extensive search, we apply a parameter optimization based on evolutionary algorithms.
4. To demonstrate the efficiency of our approach, MLB heuristics for the well-known optimization *loop invariant code motion* are generated. In contrast to previous works, our work aims at a WCET minimization.
5. Due to the integration of a machine learning tool into the novel WCET-aware compiler framework, the exploitation of a vast range of learning techniques is established. Also, the framework can be easily adopted to generate heuristics for other compiler optimizations.

6. The presented concepts can be easily adapted to ACET optimizations. Therefore, our work can be seen as a general contribution to compiler research independent of the considered objective.

The rest of this paper is organized as follows: Section 2 gives a survey of the related work. In Section 3, an overview of the current state of machine learning employed within a compiler is provided and problems arising from the selection of learners are discussed. To overcome these problems, we propose a new methodology for an automatic selection of parametric learners in Section 4. The optimization loop invariant code motion is introduced in Section 5. A description of our experimental environment and results achieved on real-world benchmarks are given in Sections 6 and 7, respectively. Finally, Section 8 summarizes this paper and gives directions for future work.

2 Related Work

ACET Minimization by Machine Learning: The application of machine learning techniques in compiler design was mainly studied in the context of the ACET minimization. Vaswani [23] uses empirical regression models to characterize interactions between optimizations in the GCC compiler. The search for good compiler optimization sequences, also called iterative compilation, has been thoroughly studied in the past. Kulkarni [11] uses genetic algorithms to avoid an exhaustive search. In [3], a characterization of the search space is used to find good compilation sequences more efficiently. Leather [13] applies fixed sampling plans while Cavazos [5] exploits performance counters to accelerate the search. In contrast, Agakov [2] reduces the number of evaluations using machine learning approaches by focusing on promising areas of the search space.

MLB Heuristic Selection for ACET Minimization: A vast application field of machine learning in compilers is the automatic generation of optimization heuristics, known in literature as *heuristic selection*. Monsifrot [18] used a supervised classification to generate heuristics for *loop unrolling* which decide whether unrolling should be performed. This approach was extended by Stephenson [15] to find MLB heuristics that predict the best unrolling factor for a given loop. Machine learning techniques (e. g., reinforcement learning) were also studied in the context of instruction scheduling [6]. In [12], a grammar-based mechanism using genetic programming is presented that automatically extracts features for machine learned heuristics.

WCET Reduction: Typically, compiler optimizations aim at an automatic reduction of the ACET. With the growing importance of embedded systems acting as real-time systems, the worst-case execution time must be considered as a crucial objective. WCET-aware compilation is a novel research area with an increasing academic and industrial interest. Approaches in this domain rely on *feedback data*, the WCET, which is provided by a static analyzer. A sophisticated WCET analyzer, also used in this work, is *aiT* [1]. Most approaches to WCET minimization operate on assembly level and exploit memory hierarchies. For example, the authors of [4] presented an algorithm for static locking of I-caches based on a genetic algorithm while compile-time cache analysis combined with static data cache locking was presented in [24]. Other approaches exploit fast scratchpad memories (SPM) for WCET minimization. Greedy algorithms for a WCET-aware SPM allocation of data are presented in [7], while optimal approaches based on an ILP formulation are explored for data and program code in [22] and [8], respectively.

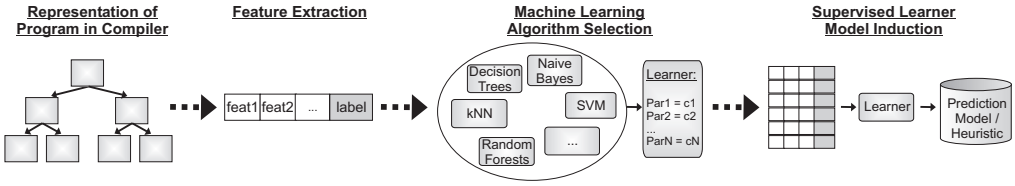

Fig. 1. Overview of Machine Learning Based Compiler Heuristic Generation

WCET Minimization by Machine Learning: The potential of machine learning for WCET minimization is sparsely explored within today's literature with only a few publications. Zhao [26] used a genetic algorithm for the search of standard low-level optimization sequences that aim at an effective reduction of the program WCET. In [14], supervised learning was used to infer heuristics for *function inlining*. The latter paper is most related to our current work since the objective of finding MLB heuristics for WCET reduction is pursued. However, there are also several significant differences. Most importantly, in [14] just a single supervised learner with its standard parameters was considered. Moreover, contrary to our compiler framework providing a seamless integration of a machine learning tool, the related paper used a compiler that was completely decoupled from the ML tool. Thus, the generated inlining heuristics had to be integrated into the compiler by hand. In addition, the optimizations are performed at different abstraction levels of the code. Function inlining was considered at the source code level whereas we consider *loop invariant code motion* as an optimization performed at assembly level.

3 Machine Learning in Compilers

In this section, an overview of supervised machine learning techniques in the compiler design is provided. Section 3.1 summarizes the common approach of incorporating machine learning techniques into a compiler and describes the workflow required to automatically generate a heuristic. A shortcoming of this workflow is the model selection problem which will be discussed in Section 3.2.

3.1 Current Workflow for Heuristic Generation

The heuristic generation begins with the obvious decision for which compiler optimization an improved heuristic should be generated. An overview of incorporating machine learning techniques into a compiler framework is depicted in Figure 1. For a **representation of the program** by internal compiler data structures, such as high- or low-level intermediate representations or abstract syntax trees, the developer has to decide which features best characterize the parts of the program to be optimized. The features must be transformed into a proper vector representation serving as input for the ML tool. This process is called **feature extraction**. In addition, for each feature vector a label representing the desired output, e. g., *YES/NO*, has to be determined. This phase transforms a set of benchmarks to become the *training set*. Next, a **selection of a learning algorithm and its parameters** is required. The machine learning community has developed a large portfolio of different learners over the last decades. Moreover, many learners have several user-defined parameters, leading to models with different performance. Due to the large number of possible combinations, the selection

of the appropriate learning algorithm is not straightforward. Finally, the chosen classifier (learner) induces a **prediction model** representing a heuristic which can be used to predict if/how the considered optimization should be performed for unseen data.

3.2 Problem Specification: Model Selection

A key aspect of the framework shown in Figure 1 is the problem of selecting a learning algorithm and parameters such that the induced model performs best in terms of the considered objective, e. g., the WCET. Due to the complex structure of learning algorithms and the non-trivial impact of their parameters, the performance of the induced model cannot be predicted statically with sufficient precision. Rather, a heuristic must be generated and its performance must be evaluated on a set of benchmarks [12].

As a consequence, the current state for the MLB heuristic generation can be seen as a *trial-and-error* approach. The compiler writer chooses a learner, induces a prediction model and evaluates the impact of the generated heuristic on benchmarks. If the heuristic did not yield the expected performance results, the compiler writer either tunes the learner parameters or even selects another learner and repeats the evaluation hoping for better results. Obviously, repeatedly evaluating different learners manually is time-consuming, error-prone, and it is often not clear if further tuning pays off. In literature [18, 15, 6, 12], typically one or two learners are employed without a detailed reasoning why exactly these algorithms including their parameters were chosen.

The exploitation of machine learning for heuristic generation is attractive since it relieves the compiler writer from the tedious task of developing heuristics manually and it also enables an easy and efficient adoption to changes in the compiler framework or the underlying system. However, the effort is now shifted from the manual tuning of heuristics to the model selection problem of learning. Here, we propose a new framework which systematically evaluates models induced by different learners and parameter settings through integration of a compiler and an ML tool.

4 Automatic Model Selection

In this section, we describe our methodology for an automatic selection of the best model. In Section 4.1, we summarize the key characteristics of the machine learning algorithms that we consider for comparison. In Section 4.2, performance evaluation of supervised learners is discussed. As will be described in Section 4.3, this evaluation can be exploited for evolutionary parameter optimization and the final model selection. For a detailed discussion of the learning methods, see standard literature [10].

4.1 Learning Algorithms

In this paper, we consider popular learning algorithms which have been successfully applied in the past for various applications and that rely on different principles.

Decision Trees partition the examples into axes-parallel rectangles by recursively splitting the training set into sub-trees. Frequently, information gain, based on the entropy (impurity) of a node, is used as splitting criterion. Additionally, there might be stopping criteria like the maximal depth of a tree and thresholds for the minimal number of examples in a node to split it further (minimal size for split), the minimal number of examples in a leaf (minimal leaf size), the minimal information gain for

splitting (minimal gain), and the number of alternative nodes considered (prepruning alternatives). Furthermore, a confidence level for post tree pruning can be specified.

Random Forests consist of several unpruned decision trees which are constructed from different bootstrap samples. The algorithm uses a randomly chosen subset of features to find the best split for each node and it is robust against overfitting. Like decision trees, random forests can still classify new examples very fast by majority voting over the predictions made by each tree in the forest. Only two parameters have to be optimized: the number of trees in the forest and the number of considered features for node splitting.

Linear Support Vector Machines (SVM) find a hyperplane which separates the examples such that those with the label $y = +1$ are in the positive half and those with the label $y = -1$ are in the negative half of the instance space. The hyperplane is determined by $\beta \cdot x + \beta_0$. The learning task is to estimate β and β_0, such that the error is minimal (i. e., the instances are placed on the correct side of the hyperplane) and that the learned model is of minimal complexity (i. e., the distance between the closest instance to the hyperplane is maximal). Those examples which are closest to the hyperplane are called support vectors. In order to allow some misclassified instances, the soft margin SVM offers a parameter C which gives a weight to the error as opposed to the complexity. Internal optimization compares all examples pairwise using a kernel function. For the linear SVM, the kernel function is the dot product $x_i \cdot x_j$.

SVMs with RBF kernel operates on not linearly separable data by including another kernel function into the SVM. The radial basis function (RBF) covers areas of instances by a Gaussian distribution: $K_{RBF}(x_i, x_j) = exp(\gamma(x_i, x_j)^2)$. Hence, the parameter of the Gaussian's width, γ, is decisive: for a low γ, almost every example is covered by its own RBF region, for a large γ, interesting regions cover a set of examples.

k-Nearest Neighbor (kNN) stores all examples and classifies a new input by looking at k most similar examples. The majority class of these k examples becomes the predicted class. If k is too small, the error is reduced, but the prediction becomes biased, e. g., by outliers. If k is too large, the error might also become large. Thus, the setting of the appropriate k is crucial for the learner's performance.

Naive Bayes predicts for an example x the class y such that the likelihood $P(y|x)$ is maximal. According to Bayes' theorem, it suffices to maximize the probability $P(X|y_i)P(y_i)$, since the a priori probability of the labels in Y (e. g., $P(y_i = YES)$ or $P(y_i = NO)$) are the same for all training examples. Implicitly, Naive Bayes assumes the independence of all example's features. Due to its simple calculation, Naive Bayes is a very fast algorithm and has typically no parameters for configuration.

4.2 Performance Evaluation

There are different metrics for performance evaluation of learners. Which metric to choose, depends on the requirements imposed by the exploiting system. The standard performance measurement of learning algorithms is accuracy. It is calculated on the basis of the test set. Examples $x_{n+1}, x_{n+2}, ..., x_{n+m}$ are handed over to the learned function f, delivering $\hat{y} = f(x)$. Then, the known true value y is compared to the predicted $\hat{y}$. Drawing training and test set under the same distribution D leads to an estimate of the true performance of the learner indicating, e. g., how often $\hat{y} = y$. The estimation is determined by generating a set of examples and splitting it into training and test set. This is done in *cross validation*: N-fold cross validation randomly partitions an example set into N sets, uses $N-1$ sets for training and the remaining set

for testing. The estimated performance of a learner is the average of the measurements of the N training and test cycles.

In case of compiler optimizations, program run time is crucial. For embedded systems acting as real-time systems, the main goal is to find a learner that yields highest WCET reduction. The WCET of a program is the longest execution time that can ever occur. Since the input to the program leading to the worst-case behavior is often not known and an exhaustive testing of all inputs is not feasible, measurements are not suitable for a WCET determination. To obtain the WCET, formal methods are used instead. The control flow graph of the program is statically analyzed taking addition information from the user, like loop iteration counts, into account. To cover all possible input data, abstraction from concrete values is used. Thus, the determination of the actual WCET is lifted to the derivation of an upper bound on the execution time of the program. In this paper, we use the term WCET as a synonym for a safe WCET estimation of the actual WCET.

The performance evaluation of a learner based on the accuracy is not appropriate since it does not allow to draw conclusions about the program's WCET.

Example 1: *Assume that a learning model acting as a compiler heuristic has to take three optimization decisions. The costs (impact on program's WCET in cycles) for the correct prediction/misprediction of the decisions are:* $Cost_A = 1/-1$, $Cost_B = 1/-1$, *and* $Cost_C = 10/-10$. *Predicting A and B correctly, but not C, results in an accuracy of* $66,\overline{6}\%$ *and a negative impact on the WCET of* $(1 + 1 - 10 =)\ -8$ *cycles, while predicting just C correctly yields a worse accuracy of* $33,\overline{3}\%$ *but a positive impact on the WCET of* $(-1 - 1 + 10 =)\ 8$ *cycles.*

Due to this missing correlation between the accuracy and the program (worst-case) execution time, learners should be evaluated by directly measuring the program performance but not their accuracy.

Moreover, the classical N-fold cross validation has to be applied in a modified fashion for the performance evaluation of learners used as optimization heuristics. For each of the N benchmarks of the example set, all examples belonging to one benchmark are excluded (test set), a ML model using the remaining examples (training set) is learned and this model is finally applied by a compiler to evaluate its impact on the WCET of the excluded benchmark. In more detail, the compiler computes the WCET $WCET_{MLB}$ for this benchmark using the new MLB heuristic and compares this value against a reference value $WCET_{ref}$. If $\Delta WCET_n < 1$, with $\Delta WCET_n = WCET_{MLB}/WCET_{ref}$, then the MLB heuristic was successful. The final performance is determined by performing the cross validation N times and computing the average relative WCET: $performance = \sum_{i=1}^{N} \Delta WCET_n/N$. Using this *benchmark-wise* cross validation is a common approach to estimate the *generalization* ability of a learning algorithm, i. e., by applying the models to *unseen* benchmarks it can be inferred how well new examples will perform using this model.

4.3 Parameter Optimization

Exhaustively searching over all combinations of user-defineable classifier parameters is not feasible. We therefore apply an evolutionary strategy [16]. Our approach is depicted in Figure 2. Each individual pn in a population of size (*pop_size*) represents a combination of parameter values, e. g., C and γ in case of the *SVM with RBF kernel*. In the beginning, the parameters of each individual are initialized randomly. To create a new

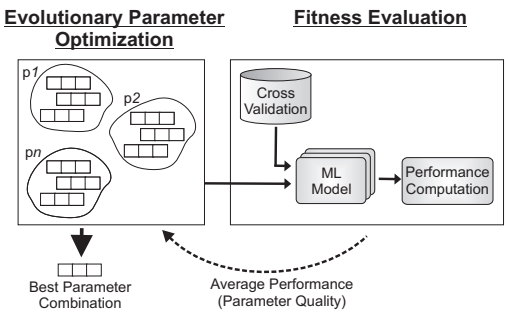

Fig. 2. Evolutionary Parameter Optimization

generation, a fraction of the individuals repeatedly takes part in a tournament selection which chooses fittest individuals (as parents) as long as *pop_size* individuals are selected. In a crossover step, individuals mate with a specified probability (*crossover_prob*). They produce children that contain the exchanged parameter values of their parents. These children are added to the current population. Then, all individuals are cloned and the clones are mutated by adding values from a Gaussian distribution to all parameters. The fitness of each individual is evaluated by a cross-validation which is based e. g., on the accuracy or, as in our case, on the reduction of the program's WCET. More accurately, for each individual a machine learning model is induced based on $N-1$ benchmarks and evaluated for the left-out benchmark. This performance computation is repeated N times and its average value represents the quality of a parameter combination. The whole process maintains the best individuals (*elitist selection*) and terminates if either a specified maximum number of generations (*max_gen*) is reached or there was no improvement over *imp* generations.

5 Case Study: Loop Invariant Code Motion

Loop invariant code motion (LICM) is a well-known ACET optimization. It recognizes computations within a loop that produce the same result each time the loop is executed. These computations are called *loop invariant code* and can be moved outside the loop body without changing the program semantics [19].

Definition 1. *An instruction i is said to be* ***loop invariant*** *iff: (a) its operands are constants, or (b) all instructions that define the operands of instruction i are outside the loop, or (c) all instructions that define the operands of instruction i are themselves loop invariant.*

LICM can be applied at the source code level to expressions, or at the assembly level, in particular to addressing computations that access elements of arrays. The positive effects are a reduced execution frequency of the moved loop-invariant code. Another positive effect of the optimization is that it might shorten the live ranges of variables leading to a decreased register pressure.

Besides these positive effects on the code, LICM may also degrade performance. This is mainly due to two reasons. First, the newly created variables to store the loop-invariant results outside the loop increase the register pressure in the loops since their live range spans across the entire loop nest. This might possibly lead to additional register spill code. This is an issue especially relevant for embedded systems with a

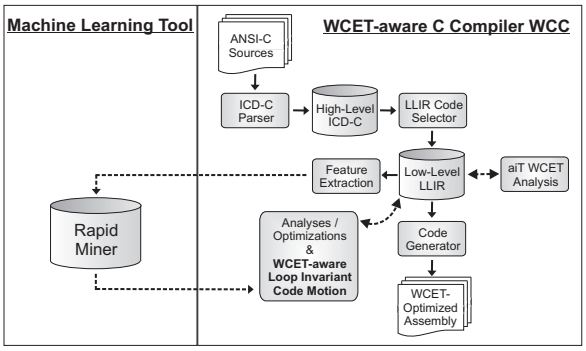

Fig. 3. Overview of Compiler Framework

small register file. For example, the TriCore processor, that is also used in this work, has 8 data and 8 address registers serving as general purpose registers. The remaining 8 data and address registers have special purposes, like storage of function arguments or return addresses, and are thus only partially exploited by the register allocation. Second, moving the loop invariant code might lengthen other paths of the control flow graph when the invariants are moved from a less executed to a more frequently executed path, e. g., moving instructions above a loop's *zero-trip test*.

Issues like the impact on the register pressure emphasize the dilemma compiler writers are faced with during the development of good heuristics. Performing loop invariant code motion has conflicting goals and it can not be easily predicted if this transformation is beneficial. LICM heuristics are also missing in standard compiler literature [19]. In addition, most compilers do not model the complex interactions between different parts of the code and the loop invariants, but perform LICM whenever invariants are found without using any heuristics that might avoid the adverse effects.

We tackle the difficult task of finding heuristics for loop invariant code motion using machine learning. The goal is to find a heuristic that exploits the positive effects of LICM on the one hand and prohibits the transformation for adverse situations on the other hand. In contrast to related works dealing with optimizations for which different heuristics are well-studied, e. g., loop unrolling, we have no hints which strategies for the LICM heuristic might be promising.

6 Experimental Environment

To demonstrate the practical use of our approach, experiments on a large number of different benchmarks were conducted. The 39 benchmarks come from the test suites DSPstone, MediaBench, MiBench, MRTC WCET Benchmark Suite, UTDSP and NetBench. On the one hand, the benchmarks are used to construct the data set for machine learning (cf. Section 6.2), which serves as training data for the LICM heuristic generation. On the other hand, they are used in the cross validation phase to evaluate the performance of the heuristic for WCET minimization. The training set based on these benchmarks comprises 3491 examples and its construction took about 50 hours on two Intel Xeon 2.13GHz quad cores. However, please note that the data set construction has to be performed once off-line.

All experiments were performed in the WCET-aware C compiler WCC [9] for the Infineon TriCore TC1796 processor. The framework including the integration of the

machine learning tool RapidMiner is depicted in Figure 3. The compiler shown on the right-hand side of the figure is provided with C source files. After parsing the C code, it is translated into the high-level intermediate representation *ICD-C*. At this level, standard compiler analyses and source code ACET optimizations (not shown in the figure) can be applied. Next, the code selector translates the code into the low-level intermediate representation *LLIR*. At this abstraction level, again different analyses and optimizations are available. In total, the compiler features 43 different optimizations which are activated in the highest optimization level $O3$. Loop invariant code motion, which is a low-level optimization within WCC, is performed as one of the last optimizations in the optimization chain. Since it is executed before register allocation, LICM operates on low-level code which does not contain physical registers but temporary variables (aka. *virtual registers*). For benchmarking, $O3$ is enabled, thus our WCET-aware LICM operates on highly optimized code.

The key feature of the WCC compiler is the tight integration of the static WCET analyzer aiT into the compiler backend. This way, WCET timing data is available in the compiler backend and can be exploited for analyses and optimizations.

6.1 Available Features

The presented compiler framework for the automatic selection of machine learning models is generic, i. e., it can be exploited to generate heuristics for a large number of low-level optimizations without any major adaption. To enable this option, a large set of features extracted from the compiler must be provided. These features must be chosen such that they cover a wide range of various characteristics of the program. Our *feature extractor* (cf. Figure 3) generates 73 features in total which describe characteristics of single instructions, basic blocks, loops, or functions depending on which low-level construct is passed to the feature extractor. The features can be classified as follows (given some examples):

1. **Structural features**: Type of instruction (arithmetic, load/store, jumps, floating point, etc.), size of given construct, number of block successors/predecessors, number of operands in given construct
2. **Liveness analysis related**: Liveness information (*live-in* and *live-out*) of instruction, number of *defs* and *uses* in instructions/blocks, information about register live times (for register pressure estimation)
3. **Loop features**: Loop nest levels, loop iteration counts
4. **Misc**: Length of critical path in loop, outcome of static branch prediction for jump instruction

This set of features is variable, i. e., depending on the application all features or just a subset can be used. The feature extractor was designed in a flexible way such that new features can be easily added. For learning algorithms that can only handle numerical values, nominal features are first transformed into discrete numerical values and then normalized by a linear transformation into $[0, 1]$.

6.2 Construction of Training Set for WCET-aware LICM

For the loop invariant code motion, 39 benchmarks are involved in the training set construction. We used WCC's feature analyzer with the full set of all 73 features. Each example of the training set was created by analyzing each loop invariant instruction i_{inv} separately. To do so, corresponding features for i_{inv} as well as for the basic block b_{pred}, to which i_{inv} is moved, were extracted. The label was determined by estimating

Table 1. Learner-specific parameters, the explored value ranges by the evolutionary search, and the best found parameter combinations yielding highest WCET reduction.

Parameter	Range	Best	Parameter	Range	Best
Decision Trees			**SVM with RBF kernel**		
max. depth	[1;20]	16	C	[0;10,000]	2405.15
min. split size	[4;100]	19	γ	[0;74]	30.08
min. leaf size	[2;100]	31	**Linear SVM**		
min. gain	[0;0.03]	0.014	C	[0;10,000]	616.11
prepr. altern.	[3;10]	4	**kNN**		
confidence	[0.1;0.5]	0.476	k	[3;100]	11
Random Forests			**Naive Bayes**		
no of trees	[1;100]	7	*no configurable parameters*		
features	[1;73]	39			

the WCET of *region reg* before and after LICM. The region *reg* is defined either as the loop to which b_{pred} belongs to or, if i_{inv} was moved completely outside a loop, *reg* represents the function were i_{inv} is located. Using outer loops for *reg* instead of the entire function makes the label extraction more reliable since it captures the effects of LICM more precisely. A decreased WCET after LICM means that the transformation is beneficial (label *YES*) for i_{inv} in its current context (b_{pred}). If the WCET does not change, also the label *YES* is used to perform such code motion which possibly enable optimization potential for subsequent LICM candidates. If a WCET increase due to adverse LICM effects was identified, the feature vector is labeled with *NO* to indicate that the code motion should be avoided for similar cases. For the next example, i_{inv} is kept in its new position and the next loop invariant instruction is considered.

6.3 Evolutionary Parameter Optimization for WCET-aware LICM

After the construction of the training set for the loop invariant code motion using exclusively the WCC compiler, the evolutionary parameter optimization for the selection of the best ML model requires a communication with RapidMiner.

The parameter optimization is performed for each of the six considered machine learners to find the model yielding the highest WCET reduction. The evolutionary algorithm generates different valid parameter combinations which are employed for the fitness evaluation. For our experiments we used the following parameters for the evolutionary algorithm: population size pop_size=20, number of generations max_gen=5, tournament selection performed on 30% of population size with a crossover probability $crossover\_prob$=90%, and termination if no improvement for imp=2 generations was observed (cf. Section 4.3).

The fitness evaluation is based on the *benchmark-wise* cross validation (cf. Section 4.2). For a given combination of parameters determined by the evolutionary algorithm, a model based on the training set of benchmarks is learned and validated against the benchmark from the test set, i. e., WCC computes the WCET $WCET_{MLB}$ for this benchmarks using $O3$ and the LICM heuristic based on the current model. This step is repeated for each of the N=39 benchmarks. To determine the quality of the model, $WCET_{MLB}$ is compared against a reference value $WCET_{ref}$ representing the WCET for this benchmark using $O3$ and disabled LICM. Finally, the fitness value which represents the quality of a given parameter combination is computed by:

Table 2. Performance results for different parameter combinations as found by evolutionary search.

Learner	Best	Worst	Avg.	Acc.
Decision Tree	96.17%	99.78%	97.42%	63.16%
Random Forests	96.60%	98.96%	97.69%	60.43%
Linear SVM	98.24%	98.62%	98.34%	53.50%
SVM with RBF kernel	**95.36%**	98.80%	97.12%	57.78%
kNN	97.32%	98.94%	97.98%	**67.48%**
Naive Bayes	98.17%	98.17%	98.17%	54.31%

$fitness = \sum_{i=1}^{N} \Delta WCET_n / N$. Obviously, this is a minimization problem, with smaller $fitness$ being better.

The output of the evolutionary parameter search is the machine learning model using the detected parameter settings that led to the highest WCET reduction. Our framework automatically performs the parameter optimization for each considered learner to find the model that exhibits the overall best WCET improvement. This model (heuristic) is finally integrated into the compiler. For future use of the novel WCET-aware LICM, the WCC compiler performs a feature extraction and consults RapidMiner to retrieve a prediction whether the considered loop invariant instruction promises a WCET reduction. The communication between WCC and RapidMiner is established in an efficient way, thus the additional overhead is marginal.

7 Results

In a first phase, the machine learning model selection was performed to find the best learner. Table 1 gives an overview of the considered learners, their parameters, and the explored parameter values by the evolutionary search (column *Range*). Please note that Naive Bayes does not provide any parameters to be optimized. However, the algorithm was considered due to its popularity and its specific functionality.

Table 2 summarizes the results of the evolutionary parameter optimizations for the six considered learners. The results in the second, third, and fourth column represent the performance values, i. e., the averaged relative WCET results obtained during the benchmark-wise cross validation (cf. Section 6.3) when comparing the WCET using the MLB heuristic against the code compiled with $O3$ and without LICM. In more detail, the second column (*Best*) represents the highest improvement of the WCET observed during the evolutionary search of each learner. These values were achieved using the parameter combinations shown in the third column of Table 1. For example, 95.36% for *SVM with RBF kernel* means that the WCET was reduced on average by 4.64%. The third and fourth column (*Worst*, *Avg.*) of Table 2 depict the worst and average WCET reduction (over all runs) found by the evolutionary search. Finally, the last column (*Acc.*) describes the classification accuracy that was computed for the parameter combination that lead to the best WCET reduction shown in the second column. The bold numbers point out the best results observed for all learners.

Three main conclusions can be drawn from this table. **First**, it can be seen that the WCET improvements significantly vary between the learners. For the considered learners and their best parameters, the relative WCET for the 39 benchmarks varies for the best parameters between 95.36% for *SVM with RBF kernel* as best model and 98.24% for the *Linear SVM*. Thus, a comparison of various learners is required

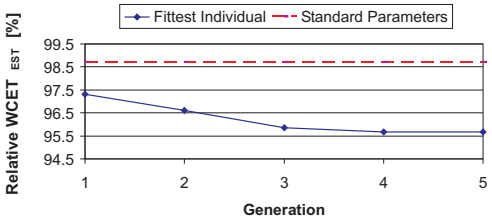

Fig. 4. Progress of Evolutionary Parameter Optimization

for the determination of the best model. Even though the difference of 2.9% might seem small, it should be taken into account that standard LICM achieves on average a WCET reduction of merely 0.6% (as will be shown later). Thus, the variation between the learners can be considered substantial and for other compiler optimizations with stronger effects on the program performance even considerably larger differences can be expected. Note also that the variance of 2.9% can not be referred to noise since statically computed WCET estimations are deterministic as the analysis always assumes the same worst-case run-time environment. **Second**, a comparison between the second and third column in Table 2 emphasizes the importance of a parameter optimization. For example, the choice of parameter settings for the learner *Decision Tree* generates LICM heuristics for which the relative WCET ranges between 96.17% and 99.78%, i. e., selecting inappropriate parameters may *waste* up to 3.61% on average of the optimization potential w. r. t. the WCET reduction. **Third**, a comparison between the WCET performance in the second column and the accuracy in the last column indicates that there is no direct correlation between these two performance metrics (cf. Section 4.2). For example, the highest accuracy of 67.48% was achieved for the *kNN* learner, while the average WCET reduction of 2.68% is poor compared to the other learners. Thus, finding the best model can be only accomplished when the model is directly evaluated against the considered objectives, in our case the WCET.

Figure 4 depicts the progress of the evolutionary parameter optimizations over 5 generations for the best learner (*SVM with RBF kernel*). The plot depicts the fittest individual (parameter combination) in each generation. As can be seen, the performance of the fittest individual is successively improved in the first four generations before no better parameters can be found in the last generation. This monotonically decreasing curve suggests that the evolutionary parameter optimization is the right choice for the search of good parameter settings in a large space. Also, a comparison between the performance of 98.71% for the standard SVM parameter settings ($C = 0$, $\gamma = 1$) and the performance of 95.36% for the best parameter combination found by the evolutionary search emphasizes the benefits of this approach. In order to evaluate the effectiveness of our machine learning based LICM heuristic, we measured the impact of our MLB heuristics for LICM on the WCET estimates ($WCET_{EST}$) of the considered 39 benchmarks. Figure 5 shows a comparison between the standard ACET LICM (*Standard-LICM*) and our optimization (*MLB WCET-LICM*) using the best heuristic generated by the *SVM with RBF kernel* learner. The reference mark of 100% corresponds to the WCET estimates for *O*3 with disabled LICM. Due to the challenges for the manual generation of an appropriate heuristic (cf. Section 5), the standard approach for LICM in many compilers is the application of the code transformation whenever possible. The light bars representing the MLB-LICM show WCET estimates computed during the benchmark-wise cross validation. By learning a model

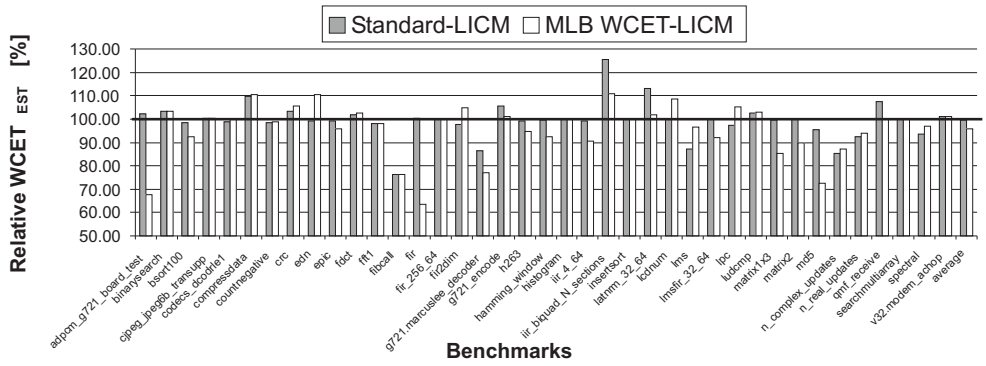

Fig. 5. Relative WCET Estimates for Standard and MLB LICM

and validating it on the excluded benchmark, the light bars indicate how good the heuristic performs on *unseen* data. As can be seen in the figure, in most cases the new MLB-LICM outperforms the standard LICM optimization, with up to 36.98% for the *fir* benchmark from the MRTC WCET Benchmark Suite. On average, the standard LICM achieves a WCET reduction of merely 0.56%, while our MLB-LICM reduces the WCET by 4.34%, as already shown in Table 2.

Most of the time for the evolutionary search was consumed by the WCET analyses. For one run of the benchmark-wise cross validation, i. e., inducing 39 models and using them for the WCET estimation of each benchmark in the test set, about 50 minutes on a single Intel Xeon 2.13GHz core of a system with 8GB RAM were required. Depending on the development of the evolutionary search, the maximal run time of 146 hours was observed for the evaluation of the learner *Random Forests*.

8 Conclusions and Future Work

Recent work has shown that machine learning can be exploited for the automatic generation of high-performance and easily adaptable compiler optimization heuristics. A central questions in this domain is that of model selection, i. e., which learners and their respective parameters should be used. This paper is the first one to address this well-known problem in a systematic way. We explore the potential of six popular learning algorithms using an evolutionary parameter optimization. In a case study, we exploit our novel compiler framework for the generation of heuristics for loop invariant code motion aiming at a WCET reduction. In contrast to standard LICM yielding an average WCET reduction of 0.56% on 39 real-life benchmarks, our new heuristics achieve a WCET reduction of 4.34% on average.

In the future, we intend to integrate further learning algorithms into our framework to explore their potential. Also, these algorithms require further evaluation to figure out why some learners work well or why not. Another important issue for future work is the integration of further compiler optimizations, e. g., register allocation, to study the generality of our methodology. The investigation of optimizations with bigger pay-offs can possibly better highlight the potential of our system. Moreover, we want to tackle another important issue of the model selection problem, the *feature selection*, which finds promising features from the set of extracted features. In [25] it has been shown that using an appropriate representation for training is beneficial for every learner and that particular learners show different preferences for the representation of features.

References

1. AbsInt Angewandte Informatik GmbH: Worst-Case Execution Time Analyzer aiT for TriCore. `http://www.absint.com/ait` (2009)
2. Agakov, F., Bonilla, E., Cavazos, J., et al.: Using Machine Learning to Focus Iterative Optimization. In: Proc. of CGO. New York, USA (2006)
3. Almagor, L., Cooper, K.D., Grosul, A., et al.: Finding Effective Compilation Sequences. In: Proc. of the LCTES. Ottawa, Canada (2004)
4. Campoy, A., Puaut, I., Ivars, A., Mataix, J.: Cache Contents Selection for Statically-Locked Instruction Caches: An Algorithm Comparison. In: Proc. of ECRTS (2005)
5. Cavazos, J., Fursin, G., Agakov, F., et al.: Rapidly Selecting Good Compiler Optimizations using Performance Counters. In: Proc. of CGO. San Jose, USA (2007)
6. Cavazos, J., Moss, J.E.B.: Inducing Heuristics to Decide Whether to Schedule. SIGPLAN Not. 39(6) (2004)
7. Deverge, J.F., Puaut, I.: WCET-Directed Dynamic Scratchpad Memory Allocation of Data. In: Proc. of ECRTS. Pisa, Italy (2007)
8. Falk, H., Kleinsorge, J.C.: Optimal Static WCET-aware Scratchpad Allocation of Program Code. In: Proc. of DAC. San Francisco, USA (2009)
9. Falk, H., Lokuciejewski, P., Theiling, H.: Design of a WCET-Aware C Compiler. In: Proc. of ESTIMEDIA (2006)
10. Hastie, T., Tibshirani, R., Friedman, R.: The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer Series in Statistics, Springer, Berlin (2008)
11. Kulkarni, P.A., Hines, S.R., Whalley, D.B., et al.: Fast and Efficient Searches for Effective Optimization-phase Sequences. ACM TACO 2(2) (2005)
12. Leather, H., Bonilla, E., O'Boyle, M.: Automatic Feature Generation for Machine Learning Based Optimizing Compilation. In: Proc. of CGO. Seattle, USA (2009)
13. Leather, H., O'Boyle, M., Worton, B.: Raced Profiles: Efficient Selection of Competing Compiler Optimizations. In: Proc. of LCTES. Dublin, Ireland (2009)
14. Lokuciejewski, P., Gedikli, F., Marwedel, P., Morik, K.: Automatic WCET Reduction by Machine Learning Based Heuristics for Function Inlining. In: Proc. of SMART. Paphos, Cyprus (2009)
15. Mark Stephenson and Saman Amarasinghe: Predicting Unroll Factors Using Supervised Classification. In: Proc. of CGO. San Jose, USA (2005)
16. Mierswa, I.: Non-Convex and Multi-Objective Optimization in Data Mining. Ph.D. thesis, Technische Universität Dortmund (2008)
17. Mierswa, I., Wurst, M., Klinkenberg, R., Scholz, M., Euler, T.: YALE: Rapid Prototyping for Complex Data Mining Tasks. In: Proc. of KDD. Philadelphia, USA (2006)
18. Monsifrot, A., Bodin, F., Quiniou, R.: A Machine Learning Approach to Automatic Production of Compiler Heuristics. In: Proc. of AIMSA. Varna, Bulgaria (2002)
19. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers Inc., San Francisco, USA (1997)
20. Srikant, Y.N., Shankar, P.: The Compiler Design Handbook: Optimizations and Machine Code Generation, Second Edition. CRC Press, Inc., Boca Raton, FL, USA (2007)
21. Stephenson, M., Amarasinghe, S., Martin, M., O'Reilly, U.M.: Meta Optimization: Improving Compiler Heuristics with Machine Learning. SIGPLAN Not. 38(5) (2003)
22. Suhendra, V., Mitra, T., Roychoudhury, A., Chen, T.: WCET Centric Data Allocation to Scratchpad Memory. In: Proc. of RTSS. Miami, USA (2005)
23. Vaswani, K., Thazhuthaveetil, M.J., Srikant, et al.: Microarchitecture Sensitive Empirical Models for Compiler Optimizations. In: Proc. of CGO. San Jose, USA (2007)
24. Vera, X., Lisper, B., Xue, J.: Data Cache Locking for Higher Program Predictability. In: Proc. of SIGMETRICS (2003)
25. Wurst, M., Morik, K.: Distributed Feature Extraction in a P2P Setting - A Case Study. Future Generation Computer Systems, Special Issue on Data Mining 23(1) (2007)
26. Zhao, W., Kulkarni, P., Whalley, D., et al.: Tuning the WCET of Embedded Applications. In: Proc. of RTAS. Toronto, Canada (2004)