Automatic WCET Reduction by Machine Learning Based Heuristics for Function Inlining *

Paul Lokuciejewski[1], Fatih Gedikli[1], Peter Marwedel[1], Katharina Morik[2]

[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. The application of machine learning techniques in compiler frameworks has become a challenging research area. Learning algorithms are exploited for an automatic generation of optimization heuristics which often outperform hand-crafted models. Moreover, these automatic approaches can effectively tune the compilers' heuristics after larger changes in the optimization sequence or they can be leveraged to tailor heuristics towards a particular architectural model. Previous works focussed on a reduction of the average-case performance.
In this paper, learning approaches are studied in the context of an automatic minimization of the worst-case execution time *(WCET)* which is the upper bound of the program's maximum execution time. We show that explicitly taking the new timing model into account allows the construction of compiler heuristics that effectively reduce the WCET. This is demonstrated for the well-known optimization function inlining. Our WCET-driven inlining heuristics based on a fast classifier called *random forests* outperform standard heuristics by up to 9.1% on average in terms of the WCET reduction. Moreover, we point out that our classifier is highly accurate with a prediction rate for inlining candidates of 84.0%.

1 Introduction

Today's embedded systems are characterized by both efficiency requirements and critical timing constraints. Average-case performance, power consumption and resource utilization are objectives describing the efficiency of a system. Timing constraints 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. Moreover, the precise knowledge of this key parameter is mandatory for various scheduling algorithms and is required for an efficient development of hardware platforms which have to meet real-time constraints.

To cope with the complex requirements imposed by modern embedded systems, software developers rely on a high-level language and an optimizing compiler. Typically, the goal of state-of-the-art compilers is the automatic reduction of the average-case execution time *(ACET)* or the energy dissipation. In contrast, a compiler-based reduction of the WCET is still a novel research area with a small number of published approaches. Similar to feedback-directed compilation relying on profiling information to optimize the average-case performance, WCET-driven compiler optimizations require the integration of a static WCET analyzer into a compiler framework. The analyzer provides worst-case timing information that allows modelling of the longest path in the control flow graph (CFG), the so-called worst-case execution path (*WCEP*). The

* The research leading to these results has been partially funded by the European Community's ARTIST2 Network of Excellence and by the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement n^{o} 216008.

objective of a WCET optimization is the minimization of this path which implicitly reduces the program's WCET.

The development of heuristics for compiler optimizations which efficiently reduce a given objective for a broad range of applications is a tedious task which requires both a high amount of expertise and an extensive trial-and-error tuning. The reasons are twofold. First, the advent of complex computer architectures complicates the development of heuristics. Optimizing compilers base their decisions on abstract architecture models. However, the construction of precise models is a time-consuming process, thus compiler writers are often constrained to employ inaccurate abstractions not able to capture all relevant architectural features. The result is a set of universal compiler heuristics that are not capable of exploiting a particular target and might even have a negative impact on the program to be optimized. Second, compiler optimizations are not performed separately but within a sequence of interfering optimizations. Since optimizations might have conflicting goals, disadvantageous interactions can be observed. However, formal models expressing interactions between optimizations are hardly known. Compiler writers are forced to either tune a heuristic for a given optimization sequence that must not be changed, or their heuristics are based on conservative assumptions which do not allow the exploration of the program's optimization potential.

A solution to this dilemma is the application of machine learning (ML) approaches to automatically generate heuristics. The key advantage of learning techniques is their capability to find relevant information in a high dimensional space, thus helping to understand and to control a complex system. Providing a description of the parts of the program to be optimized, so-called *static features*, a machine learning algorithm automatically learns a mapping from these features to heuristic parameters. Using the learning result enhances the flexibility of a compiler framework. Exchanging the target architecture or modifying the optimization sequence only requires an automatic re-learning to adjust heuristics. Thus, machine learning helps to reduce the efforts of developing compiler optimizations which is crucial in today's rapidly evolving market.

In this paper, we exploit machine learning techniques for an automatic WCET reduction. To our best knowledge, this is the first study applying supervised learning to construct heuristics for a WCET-driven compiler optimization. We consider the well-known optimization *function inlining* which can have strong effects on the WCET. An inappropriate decision to inline a function might degrade the worst-case performance of the program. We improve standard heuristics for inlining by supervised learning of a classifier that decides whether inlining of a particular function promises a WCET reduction and should thus be applied.

The main contributions of this paper are as follows:

1. In contrast to other works, we explicitly consider a detailed timing model based on the WCET which allows the extraction of features describing the program's temporal behavior.
2. Based on this data, we develop a novel high-level WCET-driven function inlining that is tailored towards an effective WCET minimization.
3. Our machine learning classifier is highly accurate since it is based on an advanced technique called *random forests* that deliver a collection of decision trees.

The rest of this paper is organized as follows: Section 2 gives a survey of the related work. The standard optimization function inlining is presented in Section 3. Section 4 introduces our WCET-driven function inlining as well as the machine learning techniques used for the automatic heuristic generation. A description of our experimental environment is given in Section 5, followed by results achieved on real-world benchmarks in Section 6. Finally, Section 7 summarizes this paper and gives directions for future work.

2 Related Work

Typically, compiler optimizations aim at the reduction of the ACET. With the growing importance of embedded systems, new optimization goals, e. g. minimization of energy dissipation or code size reduction, have moved into the focus of research. In contrast, an automatic compiler-based reduction of the WCET, which is a crucial objective in real-time systems, is still a novel research area. All the approaches rely on feedback data, the WCET, from a static analyzer. Thus, timing information is provided for the low-level intermediate representation (IR) of the program. In this work, we use the sophisticated WCET analyzer *aiT* [1].

To reduce the worst-case performance of a program, the longest path must be optimized. Most published works operate on a low-level IR and exploit memory hierarchies. [2] presents an algorithm for static locking of instruction caches based on a genetic algorithm. [3] combines compile-time cache analysis with static data cache locking. In [4], WCET-driven procedure positioning optimizations are presented. All these approaches rely on a static cache analysis which is an essential part of a WCET estimation for cache-based processors.

In [5], a genetic algorithm performing different low-level standard compiler optimizations to the program under test is applied. The objective is to find an effective optimization sequence that yields the largest reduction in the program's WCET. Zhao's paper is most related to our work since it also combines approaches from the domain of machine learning to reduce the WCET. However, the main difference is that we use supervised learning while [5] exploits an evolutionary algorithm. Furthermore, the optimizations are applied at different abstraction levels of the code.

A compiler guided trade-off between WCET and code size for an ARM7 processor was studied by [6]. They use a simplified timing analyzer to obtain WCET information employed in their code generator to produce code that exploits this trade-off and uses the two instruction sets (16-and 32-bit instructions) for different program sections.

Contrary to these low-level optimizations, WCET-driven high-level optimizations require an additional intermediate step which transforms timing information into the high-level IR and makes them accessible to the optimizations. In literature, this step is called *Back-Annotation* and was utilized in [7] where we exploit the high-level optimization *procedure cloning* to improve the precision of the WCET estimation. Also, the WCET-driven function inlining presented in this paper is an optimization operating on the program's high-level IR. All of our optimizations are integrated into the WCET-driven C compiler *WCC* which is discussed in [8].

The application of machine learning techniques for the generation of compiler heuristics has been studied for the reduction of the average-case performance. [9] uses supervised learning to generate heuristics for *loop unrolling*. Since the transformation might have a negative impact on the program execution, their generated heuristic serves as a classifier which decides if a loop should be unrolled. Loop unrolling is also considered by [10]. The authors employ supervised learning techniques to predict an appropriate unroll factor. The utilization of genetic algorithms for a generation of compiler heuristics for the optimizations hyperblock formation, register allocation, and data prefetching are discussed in [11]. Machine learning techniques (e. g., reinforcement learning) in the context of instruction scheduling are presented in [12, 13].

Neural networks and decision trees are used in [14] to predict branch behavior based on static features associated with each branch. In [15, 16], evolutionary algorithms are used in an iterative adaptive compiler to find effective compiler phase orderings.

The influence of the optimization function inlining on the ACET has been studies in different publications. In [17], equations representing the execution time performance of inlined versions of the programs have been formulated revealing which factors affect the speed of inlined code. The effectiveness of inlining has been also evaluated in [18].

From the experiments, the authors of both works conclude that this optimization was not always beneficial for the program performance. Inlining was also studied in the context of evolutionary algorithms. [19] uses genetic algorithms to tune the parameters of a dynamic Java compiler's inlining heuristics. This publication is close to the work presented in this paper, but also differs in several important ways. Most importantly, our goal is the minimization of the worst-case and not average-case performance. Furthermore, the authors employ a genetic algorithm while our approach utilizes supervised learning of a classifier. In order to receive understandable results which can easily be converted into `if-then-else` rules we have chosen the learning of random forests.

3 Function Inlining

Function inlining is a well known transformation replacing the function call with the body of the callee while storing the arguments in variables that correspond to function parameters [20]. The optimization can be either applied by the compiler at the high-level or low-level IR, or by the linker. Typically, inlining is applied to reduce the ACET.

It has several positive effects on the program execution. By copying the callee's function body into the caller, the calling overhead is reduced since the function call and return instructions as well as the parameter handling is removed. Moreover, the control transfer entailed with a call instruction is avoided, thus improving the pipeline behavior.

However, the most significant improvements result from additional optimizations that could not be applied to the original code, since they were restricted by function boundaries, but become possible after inlining. For example, the optimization *constant propagation* can replace all parameter variables with constant values used as arguments. This enables a static evaluation of conditions which can be followed by *dead code elimination* removing code within conditions evaluated to be always false. In addition, inlining generates code for the callees which can be tailored to the context of a particular call site. This code specialization usually provides further optimization potential.

The evaluation of the impact of function inlining on the program execution is challenging since the transformation influences different components, e. g. register allocation, instruction scheduling, and the usage of the memory hierarchy. Thus, its consequences are not directly visible but are noticed as side effects. This complicates the decision of whether a function should be inlined. Although it is widely believed that function inlining substantially improves the program run-time, different studies like [18] came to the conclusion that its application is not always beneficial.

One of the main drawbacks of this optimization is the increased register pressure. By inserting additional variables from the inlined function into the caller, possibly more registers are required. If the callee is inserted in an area with an already high register pressure, the register allocator is afterwards forced to add spill code. These additional accesses to the memory degrade the performance. Another problem that function inlining entails is a possible degraded cache behavior. With an increasing code size, critical sections may not remain in the instruction cache but are evicted by the inlined function. The resulting cache conflict misses slow down the program execution. Furthermore, cache performance may suffer from inlining since the locality of references is decreased.

The decision if a particular function should be inlined is made by compiler heuristics. They try to predict if the application of the optimization will be beneficial. The most common heuristic found in literature and many compilers is the consideration of the callee size. This parameter can be expressed as the number of high-level expressions or machine instructions. If a predefined value is exceeded, function inlining is omitted. Usually, this heuristic is conservative and allows inlining of only small functions.

However, due to the complex interaction between function inlining and other optimizations as well as the architecture, a simple heuristic based on the callee size is not

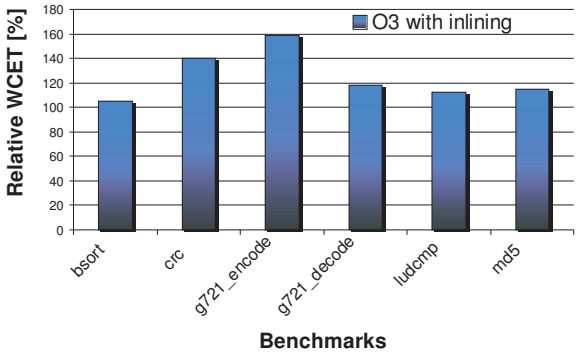

Fig. 1. Negative Impact of Function Inlining on WCET

sufficient for an effective run-time optimization. Even worse, the simple heuristic may inline inappropriate functions which substantially degrade the system performance. This is illustrated in Figure 1 showing the influence of function inlining of selected real-world benchmarks. The bars represent relative WCET estimations for the code compiled with the highest optimization level (O3) and enabled inlining using the simple heuristic based on the callee size which is bounded to 50 expressions, while 100% corresponds to the WCET for O3 with disabled inlining. It can be seen that the WCET was increased by up to 59.1% for the *g721_encode* benchmark. This figure shows that the optimization is highly dangerous and might adversely affect the program.

Hence, it is important to correctly decide whether to apply function inlining, or not. The decision should be based on empirical observation, should be easily understandable, and easily be put to practice. Machine learning techniques demand a fraction of effort compared to the hand-crafted heuristic generation. The task is then to extract features that characterize a function call, thus establishing the observations to choose an appropriate learning algorithm, and to integrate the learning result into the compiler.

4 WCET-driven Function Inlining

In this section, we begin with a brief introduction to supervised learning and discuss a technique called random forests which is employed for the construction of our inlining classifier. Next, we present our approach to gather static features for the WCET-driven heuristic. Finally, we show how the heuristic is integrated into our optimizing compiler and how the WCET-driven function inlining is applied to achieve a high WCET minimization while keeping the code expansion as small as possible.

4.1 Supervised Learning

Figure 1 emphasizes the problem encountered with function inlining. When inappropriate functions are optimized, the resulting maximal program run time can substantially increase. The reasons are non-trivial interactions between interfering optimizations as well as the memory system. These complex interactions which represent a high-dimensional space can hardly be analyzed manually to find a suitable combination of parameters that helps to make good decisions about which functions should be inlined without degrading the system performance. In addition, the simple heuristic presented in Section 3 is conservative and does not explore the entire optimization potential. Thus, it would be desirable to find an inlining heuristic that on the one hand prevents negative inlining decisions leading to a performance degradation, but on the other hand performs inlining for all functions that promise a WCET reduction.

Machine learning techniques provide a flexible, automatic and adaptive framework to effectively handle a large number of parameters which allows an evaluation of compiler optimizations in specific scenarios. The application of function inlining poses a

typical *classification problem*, i. e. we are interested in a classification rule that decides if a particular function in a particular context should be inlined or not.

Classifier learning is performed on observations $x_i, i = 1, ..., N$ that were gathered in the past, together with their binary classification (or *label*) $y_i \in \{0, 1\}, i = 1, ..., N$. Due to the given classifications, this learning task is called *supervised*. An example is a pair $\langle x_i, y_i \rangle$. The set of N examples used for learning is called the *training set*. Another set of examples, the *validation set*, is used for the evaluation of the learning result: the learned classifier is applied to observations and its output labels are then compared to the true labels. If the true label and the one outputted by the learned classifier are the same, the observation was classified correctly, otherwise it is an error. The *accuracy* of a learning result is simply the ratio of all correctly classified observations divided by the total number of observations in the validation set.

Here, an observation x_i is represented by a *feature vector*. The features are extracted from the source code as well as information obtained from the WCET analysis. Label y_i denotes whether a function call i characterized by x_i should be inlined. For the training and validation set, the labels were determined by a static WCET analysis. We measured the global WCET twice, for the program compiled with O3 with and without inlining function call i. If the WCET was decreased after performing the optimization to call i, the label was set to 1 indicating that a call with the characteristics x_i yields a WCET reduction, otherwise the label was set to 0.

The task of training a classifier is to find a mapping from feature vectors to labels such that the classification error is minimal or the accuracy is maximal, respectively. This model is used afterwards for the classification of new examples that were not considered during the training phase. For our WCET-aware inlining the trained classifier predicts for inlining candidates in new programs if their inlining promises a WCET reduction omitting an evaluation via a costly WCET analysis.

There are several algorithms for supervised learning. Where linear models which determine a hyperplane in the feature space separating positive and negative examples often deliver highly accurate results even in high-dimensional spaces, their learning result is hard to understand and, hence, its plausibility can hardly be checked. In contrast, partitional approaches like decision trees can easily be interpreted. A decision tree splits the set of examples according to the values of one feature (a vector component). In each subspace, this step is recursively repeated until all examples in a subspace belong to the same class (i. e., have the same y-value). In other words, a decision tree consists of a sequence of decisions represented by a tree. Each node in tree performs a test on the values of a single or several features, splitting the path based on possible outputs. Leaves in decision trees represent the final output, which is the predicted class. To classify an object, the tree is traversed on a particular path from the root node to a leaf depending on the test results based on the values of the feature vector.

Ensemble techniques which combine several learning results and classify according to the majority have shown to be robust. Hence, an ensemble of decision trees is a promising approach for our task.

Random Forests

Random forests consist of many *unpruned* decision trees constructed from different *bootstrap* samples which are obtained from a training set D of size N by sampling examples from D uniformly and with replacement. In contrast to standard trees where node splits are based on all features from the training set, random forests use a randomly chosen subset of features to find the best split for each node. This counterintuitive strategy turned out to be highly accurate and generates results comparable with other state-of-the-art algorithms like *Support Vector Machines* or *Adaboost* [21]. Other important advantages of random forests are their speed, their robustness against *overfitting*, and

their user-friendliness since just two parameters (number of considered features for node splitting and number of trees in forest) have to be defined.

To predict novel data, the object under consideration is classified by each of the trees in the forest and the outputs are aggregated by a majority vote, i. e. the most frequently predicted class is the final output.

Random forests provide an unbiased estimation of the classification error. For each constructed decision tree based on the bootstrap sample, examples not in the bootstrap sample, called *out-of-bag* data, are predicted. In a second step, these predictions are aggregated to calculate the error rate. The probability for bootstrapping of sampling a particular example i from a set of size N is $1/N$, while the probability of not to sample i is $1 - 1/N$. Thus, the probability that i is not sampled after N iterations is

$$\left(1 - \frac{1}{N}\right)^N \approx e^{-1} = 0.368 \tag{1}$$

This means that the training set contains approximately 63.2% of examples. Thus 36.8% of examples are not used in the training phase and the error estimation is pessimistic.

Combining understandability, efficiency, and robustness has led to the decision to apply random forests to learning heuristics for function inlining.

4.2 Feature Extraction for WCET Minimization

Before machine learning can be applied, the training set has to be established. In particular, an appropriate description of the inlining candidates need to be determined. Which features capture the main function characteristics that influence the WCET on a modern processor? How to extract these features from the program?

The feature extraction is based on the high-level intermediate representation (IR) of the program. In addition, further features are obtained from a static WCET analysis. As will be discussed in more detail in Section 5, WCET information that is provided by the static analyzer at the low-level is automatically transformed into the high-level IR and made accessible to the feature extraction. We present a subset of the 22 features used in this paper that are considered to be most significant for the classification, as indicated by the results of the *Variable Importance Measure* discussed later in Section 6.

First, integer features, generally having a value range of $\mathbb{N}_0$, are presented. It should be noted that all relative features are given as percentages, whereas WCET estimations are measured in cycles. Furthermore, *callee* denotes the function to be inlined, while *caller* represents a function containing a call to the inline candidate.

- **Caller/callee size**: Size of caller/callee measured in number of expressions
- **Calls in caller**: The number of function calls within the caller
- **Caller/callee WCET**: Accumulated WCET of the caller/callee over all calling contexts, i. e. for all invocations
- **Call-related WCET**: The WCET of a function related to a particular function call, i. e. the product of the function's WCET for a single invocation and the execution frequency of the corresponding call
- **Call execution frequency**: Worst-case execution counts for a particular call expression derived from the WCET analysis
- **Relative caller/callee WCET**: The relative WCET of the caller/callee considered for all calling contexts w.r.t. the overall program WCET

Next, two important binary features are introduced.

- **One-Call function**: This feature indicates if the considered function is invoked by exactly one call expression.
- **Call in loop**:When a call to the inline candidate is contained in a loop, this feature is assigned the value yes.

This selection of features indicates that our approach significantly differs from previous works. Monsifrot [9] or Stephenson [10] aim at a ACET reduction and exclusively consider features that are extracted from the source code. In contrast, we focus on heuristics for a WCET minimization. Besides the consideration of features based on the source code, our approach additionally includes valuable features that are provided by a static WCET analyzer. These key parameters allow a construction of inlining heuristics for an efficient WCET minimization.

Register Pressure Analyzer

As discussed in Section 3, function inlining can potentially increase the register pressure. Calling a function, *caller-saved* registers are saved before the callee is entered and restored after returning to the caller. This context save enables the usage of saved registers in the callee. Performing function inlining and replacing the call by the callee's function body makes the context save redundant. However, as a consequence, the number of registers that can be exploited in the inlined function is decreased, leading to a higher register pressure in the caller.

Increasing the register pressure can potentially lead to a generation of spill code which has a negative effect on the program run time. Ideally, a WCET-driven inlining heuristic should be able to predict whether inlining a function yields spill code in order to prevent the application of this optimization and to anticipate an increase in the WCET. For this purpose, the *register pressure analyzer (RPA)* was developed.

The RPA takes those registers into account that can be potentially spilled, i. e. all address and data registers for which the compiler can produce spill code. In our case, input for the RPA is the program under analysis in its low-level representation after the register allocation. Since our compiler incorporates both a high- and a low-level IR, we have full control of the generated code and the mapping between both code representations. Thus, results provided by the RPA can be easily assigned to the corresponding high-level constructs.

For the feature extraction, the RPA provides significant information for the classification of inline candidates based on the register distribution. Following additional features are taken into account:

- **Number of live address/data registers across calls**: These are registers that must be preserved across a call, i. e. their lifetime spans a call. Inlining of calls crossing a high number of lifetimes increases the probability for spilling.
- **Max. number of address/data registers with co-existing lifetimes**: The maximal number of registers that are simultaneously live within a function. This feature is also an indicator for potential spill code. Even functions with a small number of registers with a lifetime crossing a call may be potential spilling candidates when the inlined function has a high register pressure.

Further features characterizing the call to function *f* represent the execution frequency of the basic blocks containing the call expression to *f*, the number of call expressions in *f*, the number of call expression to *f*, and the number of call expressions to function *f* lying on the worst-case execution time path.

4.3 Label Extraction

The second phase of establishing the training set is the generation of labels for the corresponding feature vectors. The labels are automatically determined. For this purpose, our compiler WCC is run twice. In a first run, the analyzed program is compiled with the highest optimization level (O3) with disabled compiler's standard function inlining. For this program, the overall program WCET $WCET_{ref}$ is computed and serves as reference value. In a second run, the same program is again compiled with O3 and function inlining disabled for all function calls except for function call i. Function call

i is the currently considered inlining candidate for which an example in the learning set is generated. For this generated code where exclusively the function at call i was inlined (independent of the function's size), the program's WCET $WCET_i$ is determined. Thus, a comparison between both WCETs indicates the influence of inlining function call i on the WCET. The value of $label_i$ is determined as follows:

$$label_i = \begin{cases} yes, & \text{if } \frac{WCET_i \cdot 100}{WCET_{ref}}\% \leq 99\% \\ no, & \text{otherwise} \end{cases} \quad (2)$$

The value of 100% means that inlining of function call i had no influence on the WCET. If the value is less than 100%, a WCET reduction was achieved, otherwise inlining had a negative impact on the worst-case performance. We set the threshold to 99% , i. e. if inlining at call i reduced the program's WCET by at least 1%, this call is considered as being beneficial. This strategy is motivated by the potential code expansion which is a crucial issue in particular for embedded systems. The threshold ensures that a minimal WCET reduction of less than one percent coming with a potentially large code size increase is not classified as a positive inlining example.

The label extraction is automatically performed for all function calls in the program that allow function inlining. For each call, in a first step its features are extracted and in a second step the corresponding label is determined. These examples are collected for all considered benchmarks and serve as input to the automatic generation of a heuristic as described in the following.

4.4 Application of WCET-driven Function Inlining

One of the advantages of random forests is the possibility of transforming the classification rules into equivalent programming language constructs. Since random forests are a collection of decision trees representing tests of conditions concerning the features, they can be translated into `if-then-else` statements that are incorporated into the compiler as optimization heuristics.

To achieve a maximal WCET reduction simultaneously with a small increase of code size, we propose a complementary heuristic for the selection of inlining candidates. Except for some specialized compilers like the HPUX Itanium compiler performing a selective inlining called *inline specialization* [22], ordinary compilers consider inlining candidates in a top-down manner by traversing the source code. In contrast, we start with the optimization of callees having the largest WCET. These are usually functions that promise the largest WCET reduction when inlined. By following this order, we exploit the maximal optimization potential with the minimal number of transformations, thus avoiding a too heavy code expansion in the first optimization steps. After finding such an inlining candidate with maximal WCET, our novel heuristic is applied to decide if function inlining should be performed.

5 Experimental Environment

To indicate the efficacy of our WCET-driven function inlining, evaluation on a large number of different benchmarks was performed. The 41 benchmarks come from the the test suites MRTC WCET Benchmark Suite [23], NetBench Suite [24], MiBench [25], and our own collection of real-world benchmarks containing miscellaneous applications, e. g. an H263 coder or a G.721 encoder. The size of the benchmark codes ranges between 58 and 12082 Bytes.

The tests are conducted using our WCET-aware C compiler WCC for the Infineon TriCore TC1796 processor using two different types of memories. The first memory is the program scratchpad memory (SPM) with a capacity of 48Kbyte, the second memory utilized in our experiments is a 2 MB cached program flash memory. Our WCET

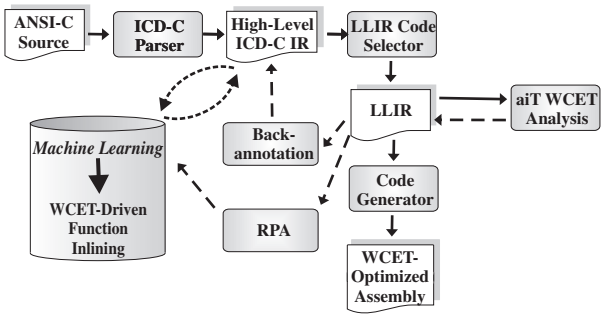

Fig. 2. Workflow for WCET-driven Inlining

analyzer allows the modification of the I-cache capacity. To take cache effects, which are crucial for the code expansion during inlining, into account, a cache capacity of 2 KByte was chosen. Due to architectural reasons, the system behavior differs depending on whether the program is executed from the SPM or flash. Thus, we collected two different learning sets for each memory type, each of which contains 275 feature vectors that were extracted from all benchmarks with approximately 70% of *negative examples*, i. e. function inlining had a negative effect on the WCET for these cases. This number points out that function inlining is an optimization that should be used with caution. The number of extracted examples ranges from 1 example for small benchmarks up to 59 examples for larger benchmarks like *g721_encode*.

The workflow is depicted in Figure 2. Before our WCET-driven inlining can be applied, the heuristics are trained offline. As described in the previous section, the construction of the learning set for supervised learning is done automatically. After parsing the C source code, the program is translated into the high-level intermediate representation ICD-C IR [26]. In the next step, a code selector is used to translate the program into the low-level IR called LLIR [27]. Finally, the program is passed to AbsInt's WCET analyzer aiT. This workflow is depicted by solid arrows.

Subsequently, WCET information generated by aiT is imported into our compiler. To make it accessible to the feature extraction in the ICD-C IR, a Back-Annotation, which establishes a connection between LLIR and ICD-C IR objects using the code selector, is performed in order to translate WCET information from the low- to the high-level IR. After this step, each ICD-C object is annotated with WCET information that is derived from the corresponding LLIR object counterpart. In addition, the register pressure analyzer (see Section 4.1) collects information about the allocation of physical registers in the LLIR. Together with the ICD-C IR, the RPA and the Back-Annotation provide information about the features of calls to the inlining candidates. With the WCET information from the Back-Annotation, the labels and the feature vectors are determined and finally passed to the machine learner, in our case the tool *RapidMiner* [28], to generate the random forests based inlining heuristics. After this learning phase, the generated heuristics are incorporated into the ICD-C optimizer of our compiler and exploited by the WCET-driven function inlining. The learning workflow and the application of the novel optimization are shown by dashed arrows.

6 Results

Worst-Case Execution Time

In order to evaluate the effectiveness of our machine learning based (MLB) inlining heuristic and to compare it against the standard ACET inlining heuristics, we measured the WCET for the real-world benchmarks after the code transformation. Table 1 shows the overall results achieved for the two memory types SPM and the cached flash. The reference value is the WCET of the program compiled with the highest optimization

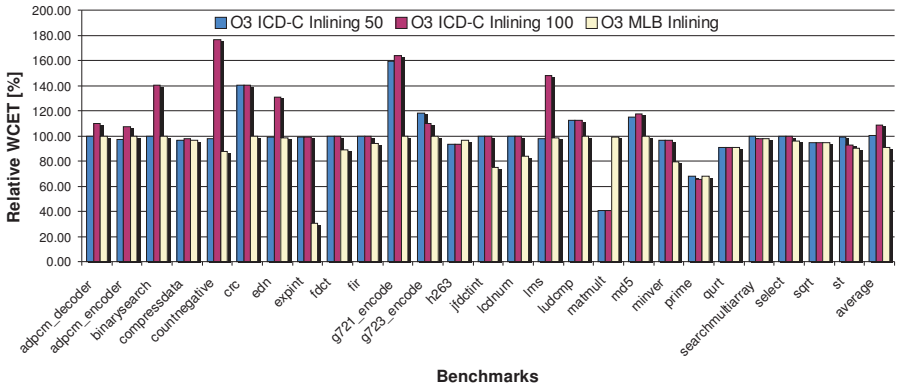

Fig. 3. Relative WCET after Inlining

level (O3) with disabled function inlining. In the first row, the standard ICD-C inlining heuristic was used (inlining functions smaller than 50 expressions) and the relative WCET (average for all benchmarks) was estimated. For a comparison, in the second row the ICD-C inlining heuristic was increased to 100 expressions. Finally, the last row represents the relative WCETs when the standard heuristic is replaced by the machine learning based heuristic. The results point out that our new heuristic outperformed the

	WCET - SPM	**WCET - Flash**
O3 ICD-C Inlining 50	101.7%	100.3%
O3 ICD-C Inlining 100	104.6%	105.5%
MLB WCET-driven Inlining	92.6%	94.1%

Table 1. Overview of Resulting WCET

standard hand-crafted heuristic (limiting callee size to 50 expressions) on average by 9.1% for the scratchpad and by 6.2% for the flash memory. When the standard inlining heuristic is increased to 100 expressions for the inlining candidates, our MLB inlining achieves a WCET reduction of 12% for the SPM and 11.4% for the flash memory. Moreover, comparing row two and three in Table 1, the conclusion can be drawn that it is more beneficial to have a conservative heuristic that prefers optimization of smaller functions since inlining of larger function might have a strong negative effect on the program run time.

In general, unlike the standard ICD-C heuristic, the MLB heuristic never significantly increases the WCET in all our experiments. For two benchmarks a marginal WCET increase of less than 2% was observed. Figure 3 shows a selected subset of the 41 considered benchmarks omitting benchmarks for which comparable results were achieved for the MLB and standard heuristic, i. e. inlining was performed or prevented in both cases. The bars represent relative WCET estimations for the flash, with 100% corresponding to the WCET for O3 with disabled inlining. Three benchmarks are discussed in more detail which summarize the typical impact of our heuristic.

The benchmark *crc* benefits from the new heuristic since it prevents inlining. In contrast, the standard heuristic performs inlining of function *icrc1* which enlarges the caller preventing the compiler to apply further optimizations like *procedure cloning* with code size constraints yielding an WCET increase of 40.0%. Moreover, inlining leads to additional spill code within a loop that originally contained the function call.

Another typical example is benchmark *expint* with a strong WCET reduction of 69.0% w.r.t. the standard ICD-C inlining. The reason for this WCET improvement is an additional optimization potential that was established with our MLB heuristic. The benchmark contains a function *expint* having more than 100 expressions, thus not con-

	Code Size - SPM	Code Size - Flash
O3 ICD-C Inlining 50	102.9%	103.0%
O3 ICD-C Inlining 100	107.6%	107.8%
MLB WCET-driven Inlining	94.2%	95.3%

Table 2. Relative Code Size after Inlining

sidered for standard inlining. In contrast, our heuristic which does not use the function size as the only restriction, permits inlining and thus enables an additional application of *constant propagation* for some functions parameters which in turn make *constant folding* possible. This additional optimization potential helps to significantly improve the code quality.

The only negative example encountered during our experiments arises for benchmark *matmult* where the standard heuristic performed better than our MLB approach. With standard inlining, a small function was inlined leveraging further optimizations, while our classifier predicted to prevent inlining. This inaccurate decision can be attributed to the small number of feature vectors (61 positive examples) used to train the model for classifying an inlining candidate as beneficial. Such classification errors are typical for small learning sets and with an increased number of positive examples inaccurate decisions can be expected to be smaller.

Code Size Increase

Table 2 shows the influence of function inlining on the code size for the scratchpad and flash memory, with 100% being the code size of the benchmark compiled with O3 and disabled inlining. Unlike standard inlining, our MLB optimization could reduce the code size on average by up to 5.8%. The reasons are twofold. First, our heuristic performed in total less function inlining than the standard ICD-C inliner. Our heuristic is more conservative and prevents optimization in potentially dangerous situations where a WCET increase might be expected. This explains why the code expansion is smaller than for ICD-C inlining. Second, some of the inlined functions are so called *one-call functions*. These are functions that have a single function call within the source code. By inlining them and assuming that the code is not executed by other programs, the original function can be removed from the source code. In addition, the calling overhead is avoided and the inlined function body can be further optimized. This leads to smaller code than for the non-inlined code as can be seen in the last row of the table. Thus, negative effects on the cache performance due to a code size increase coming from inlining are very unlikely.

	Accuracy - SPM	Accuracy - Flash
Correctly classified examples	84.0%	83.5%
Correctly classified positive examples	36.1%	19.2%
Correctly classified negative examples	97.7%	99.5%

Table 3. Accuracy and class recall based on LOOCV

Accuracy of Classification

The quality of a classifier can be estimated by the classification error. This is an important issue since we are interested in the accuracy of our classifier for novel programs not considered in the training phase. To evaluate our random forests heuristic, we apply a common approach called *Leave-One-Out Cross Validation (LOOCV)*. It subdivides the learning set (consisting of all collected feature vectors) into two classes, a training and a validation set. In particular, LOOCV eliminates a single example from the learning set of size n, exploits the remaining $n-1$ examples to learn a classifier and finally uses the eliminated example to validate the trained model. This validation is repeated n times

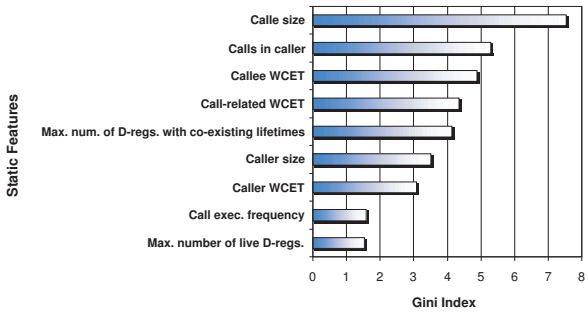

Fig. 4. Variable Importance Measure

for each example considered once as validation set. This approach is best suited for applications with only a small number of available learning examples since the learning algorithm can be precisely trained with almost all examples.

Table 3 summarizes the LOOCV results based on the criterion *gini index* which reaches a value of zero when only one class is present at a node defined by a particular static feature. The first row represents the accuracy while the next two lines indicate the so-called *class recall* for the positive and negative examples. In total, 84.0% and 83.5% of the examples could be correctly classified for SPM and the flash, respectively. It can be observed that the classification error for positive examples (classifier predicts inlining as beneficial) is significantly larger than the error for the negative tests. A reason for that is the small number of positive examples used in our experiments. Thus, the learning algorithm was not able to generate an accurate model for positive examples as was accomplished for the negative examples. In general, these results are fully satisfactory for our WCET-driven function inlining since we focussed on an accurate classification of negative examples to avoid strong performance degradations due to incorrect inlining decisions.

Variable Importance Measure

One advantage of random forests is their capability of estimating the importance of variables for the classification, called *impurity measures*. The importance of a variable is determined by its contribution for an effective classification and reveals if the features chosen for the learning algorithm are appropriate. Figure 4 depicts the importance of attributes discussed in Section 4.1 based on the gini index. As expected, the most important variable for the classification of inlining candidates is the size of the callee. This is also the attribute that is found in inlining heuristics of most compilers. However, as could be seen with the WCET results, the exclusive consideration of this attribute is often not sufficient and might lead to an inappropriate inlining decision.

It can be also seen that attributes concerning the program's worst-case behavior have a high importance. This underlines that an optimization that is tailored towards a WCET minimization must take WCET information during the learning phase into accout. Last but not least, Figure 4 reveals that the attribute characterizing the number of data registers with co-existing lifetimes, being an indication for potential spill code, is important. Thus, a register pressure analyzer is a source of crucial information when dealing with code-size critical optimizations.

Simulated Time

Finally, we measured the simulated time of the benchmarks using our MLB WCET-driven heuristic in order to compare the heuristic's impact on the ACET. In general, the results for the simulated time performed slightly worse than for the WCET. However, in contrast to general-purpose machines, this small increase in the simulated time is usually not crucial for embedded systems for which the adherence of real-time constraints

is the key objective. For some benchmarks, e. g. *fir*, our MLB heuristic yields a WCET reduction of 6.0% on the one hand and an increase in the simulated time of 1.9% on the other hand. This shows that our heuristic is aiming at an effective WCET minimization which can be only accomplished when a specific cost function is considered.

Compilation Time
Our WCET-aware function inlining has an impact on the compilation time which mainly results from the feature extraction. During this phase, the original program must be passed to aiT in order to obtain WCET information about the code. The evaluation of the MLB heuristics which are simple `if-then-else` statements is in contrast negligible.

7 Conclusions and Future Work

Machine learning was recently employed to automatically generate compiler heuristics for the improvement of the system's average-case performance. These techniques are promising since they reduce the complexity of compiler design by relieving compiler writers of tedious heuristic tuning. In addition, the automatically generated heuristics often outperform hand-crafted models.

In this paper, machine learning techniques are integrated for the first time into an optimizing compiler to automatically reduce the WCET. We demonstrate how random forests, a supervised learning technique, can be exploited for the construction of WCET-centric compiler optimization heuristics. For this purpose, we developed an automatic system to extract program features based on data from a WCET analyzer. This data is provided by WCC's Back-Annotation which transforms worst-case execution data from the low-level to the high-level IR. Subsequently, this data is used to construct a heuristic for the optimization function inlining. Our novel WCET-driven inlining outperforms standard inlining heuristics by up to 9.1% on average w.r.t. WCET reduction. We also show that random forests are well suited for compiler heuristics and yield classifiers of high accuracy. In our experiments, leave-one-out cross validation estimated a prediction rate of 84.0% and 83.5% for SPM and flash memory, respectively. In addition, we present results of the variable importance measure indicating attributes that are most relevant for an effective classification of inlining candidates.

In the future, we intend to automate the incorporation of random forests heuristics into our WCET compiler, i. e. the textual tree representation of the classifier should be automatically parsed into equivalent programming language constructs and not translated manually as is done currently. This would enable, in addition to the current estimation of the accuracy, an automated LOOCV w.r.t. the WCET, i. e. to learn iteratively a heuristic from all but one benchmark and subsequently determine the WCET for that benchmark applying the learned heurisitc. We also plan to study the potential of other machine learning approaches for further optimizations to minimize the WCET. Last but not least, we work on the integration of further benchmarks providing more examples that improve the accuracy of generated compiler heuristics.

Acknowledgments

The authors would like to thank AbsInt Angewandte Informatik GmbH for their support concerning WCET analysis using the aiT framework.

References

1. AbsInt Angewandte Informatik GmbH: Worst-Case Execution Time Analyzer aiT for TriCore. (2008)

2. Campoy, A.M., Puaut, I., et al., A.P.I.: Cache contents selection for statically-locked instruction caches: An algorithm comparison. In: Proc. of ECRTS. (July 2005)
3. Vera, X., Lisper, B., Xue, J.: Data cache locking for higher program predictability. In: Proc. of SIGMETRICS. (July 2003)
4. Lokuciejewski, P., Falk, H., Marwedel, P.: WCET-driven Cache-based Procedure Positioning Optimizations. In: Proc. of ECRTS. (July 2008)
5. Zhao, W., Kulkarni, P., Whalley, D., et al.: Tuning the WCET of Embedded Applications. In: Proc. of RTAS. (May 2004)
6. Lee, S., Lee, J., Park, C.Y., Min, S.L.: A Flexible Tradeoff between Code Size and WCET using a Dual Instruction Set Processor. In: Proc. of SCOPES. (September 2004)
7. Lokuciejewsi, P., Falk, H., Marwedel, P., Henrik, T.: WCET-Driven, Code-Size Critical Procedure Cloning. In: Proc. of SCOPES. (March 2008)
8. Falk, H., Lokuciejewski, P., Theiling, H.: Design of a WCET-Aware C Compiler. In: Proc. of ESTIMedia. (October 2006)
9. Monsifrot, A., Bodin, F., Quiniou, R.: A Machine Learning Approach to Automatic Production of Compiler Heuristics. In: Proc. of AIMSA. (September 2002)
10. Mark Stephenson and Saman Amarasinghe: Predicting unroll factors using supervised classification. In: Proc. of CGO. (March 2005)
11. Stephenson, M., Amarasinghe, S., Martin, M., O'Reilly, U.M.: Meta Optimization: Improving Compiler Heuristics with Machine Learning. SIGPLAN Not. **38**(5) (2003)
12. McGovern, A., Moss, E.: Scheduling Straight-line Code using Reinforcement Learning and Rollouts. In: Proc. of NIPS. (September 1999)
13. Cavazos, J., Moss, J.E.B.: Inducing Heuristics to Decide Whether to Schedule. SIGPLAN Not. **39**(6) (2004)
14. Calder, B., Grunwald, D., Jones, M., Lindsay, D., Martin, J., Mozer, M., Zorn, B.: Evidence-based Static Branch Prediction Using Machine Learning. ACM Trans. Program. Lang. Syst. **19**(1) (1997)
15. Cooper, K.D., Schielke, P.J., Subramanian, D.: Optimizing For Reduced Code Space using Genetic Algorithms. SIGPLAN Not. **34**(7) (1999)
16. Guo, Y., Subramanian, D., Cooper, K.D.: An Effective Local Search Algorithm for an Adaptive Compiler. In: Proc. of SMART. (January 2007)
17. Davidson, J.W., Holler, A.M.: Subprogram Inlining: A Study of its Effects on Program Execution Time. Technical report, Charlottesville, VA, USA (1989)
18. Cooper, K.D., Hall, M.W., Torczon, L.: An Experiment with Inline Substitution. Softw. Pract. Exper. **21**(6) (1991)
19. Cavazos, J., O'Boyle, M.F.P.: Automatic Tuning of Inlining Heuristics. In: Proc. of Supercomputing. (November 2005)
20. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1997)
21. Freund, Y., Schapire, R.E.: Experiments with a New Boosting Algorithm. In: Proc. of Int. Conference on Machine Learning. (June 1996)
22. Chakrabarti, D.R., Liu, S.M.: Inline Analysis: Beyond Selection Heuristics. In: CGO '06: Proceedings of the International Symposium on Code Generation and Optimization, Washington, DC, USA, IEEE Computer Society (2006) 221–232
23. Mälardalen WCET Research Group: WCET Benchmarks. http://www.mrtc.mdh.se/projects/wcet (September 2008)
24. Memik, G., Mangione-Smith, W.H., Hu, W.: NetBench: A Benchmarking Suite for Network Processors. In: Proc. of ICCAD '01. (November 2001)
25. Guthaus, M., Ringenberg, J., Ernst, D., Austin, T., Mudge, T., Brown, T.: MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In: Proc. of International Workshop on Workload Characteristics. (December 2001)
26. Informatik Centrum Dortmund: ICD-C Compiler framework. http://www.icd.de/es/icd-c (September 2008)
27. Informatik Centrum Dortmund: ICD Low Level Intermediate Representation Backend Infrastructure (LLIR) – Developer Manual. Informatik Centrum Dortmund (September 2008)
28. Mierswa, I., Wurst, M., Klinkenberg, R., Scholz, M., Euler, T.: YALE: Rapid Prototyping for Complex Data Mining Tasks. In: Proc. of KDD. (August 2006)