Multi-Objective Exploration of Compiler Optimizations for Real-Time Systems[1]

Paul Lokuciejewski, Sascha Plazar, Heiko Falk, Peter Marwedel
Computer Science 12
TU Dortmund University
D-44221 Dortmund, Germany
FirstName.LastName@tu-dortmund.de

Lothar Thiele
Computer Engineering and Networks Laboratory
ETH Zurich
CH-8092 Zurich, Switzerland
thiele@tik.ee.ethz.ch

***Abstract*—With the growing complexity of embedded systems software, high code quality can only be achieved using a compiler. Sophisticated compilers provide a vast spectrum of various optimizations to improve code aggressively w. r. t. different objective functions, e. g., average-case execution time *(ACET)* or code size. Due to the complex interactions between the optimizations, the choice for a promising sequence of code transformations is not trivial. Compiler developers address this problem by proposing standard optimization levels, e. g., *O3* or *Os*. However, previous studies have shown that these standard levels often miss optimization potential or might even result in performance degradation.**

In this paper, we propose the first adaptive WCET-aware compiler framework for an automatic search of compiler optimization sequences which yield highly optimized code. Besides the objective functions ACET and code size, we consider the worst-case execution time *(WCET)* which is a crucial parameter for real-time systems. To find suitable trade-offs between these objectives, stochastic evolutionary multi-objective algorithms identifying Pareto optimal solutions are exploited. A comparison based on statistical performance assessments is performed which helps to determine the most suitable multi-objective optimizer. The effectiveness of our approach is demonstrated on real-life benchmarks showing that standard optimization levels can be significantly outperformed.

I. Introduction

Modern systems require both highly efficient hardware and aggressively optimized software. In particular, resource-restricted embedded systems rely on software tailored towards given specifications. With the growing complexity of embedded software, code generation and optimization must be automatically carried out by compilers. Modern compilers provide a vast portfolio of optimizations which exhibit complex mutual interactions and affect different objective functions, such as average-case execution time, code size, or energy dissipation in a hardly predictable fashion.

Since compiler optimizations are not considered separately, the search for suitable optimization sequences and optimization parameters that promise a positive effect on a single or multiple objective functions is not straightforward. To cope with this problem, compiler developers construct standard optimization levels, like *O3* or *Os*, which are based on their experiences. However, there is no guarantee that these optimization levels will also perform well on untested architectures or for unseen applications. Previous studies like [1], [2], [3], [4] have indicated the poor performance of standard optimization levels and pointed out that more sophisticated approaches are required for finding effective compilation optimizations.

Concerning the code generation for embedded systems acting as service-oriented hard real-time systems, the optimization problem becomes even more complex. 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.

As a consequence, system designers of real-time systems must consider different objectives in a synergetic manner. Concerning the compiler-based code generation, there is no single compiler optimization sequence which satisfies all objectives. Therefore, multiple trade-offs must be considered enabling the system designer to choose among different solutions which best suit the system specifications.

This paper proposes a novel modular and flexible framework to explore the performance of compiler optimizations with conflicting goals. Since typical state-of-the-art compilers provide a vast number of optimizations, the search space is too large to be exhaustively explored. To cope with this complexity problem, we apply evolutionary multi-objective (*EMO*) algorithms which efficiently find a good approximation of *Pareto fronts* representing the best compromise between the considered objectives. The advantages of our framework are twofold. First, our techniques reduce the complexity of compiler design/usage by relieving compiler writers/users from the tedious task of searching for appropriate optimization sequences. Second, the automatically determined optimization sequences clearly outperform commonly used standard optimization levels, leading to higher system

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

performance compared to traditional system design.

The main contributions of this paper are as follows:

1) We propose the first fully functional adaptive WCET-aware compiler to perform a multi-objective compiler optimization level search for service-oriented real-time systems. To the best of our knowledge, trade-offs with the objective WCET have not been considered yet.
2) Our framework approximates Pareto optimal solutions for the most crucial objectives in resource-restricted real-time systems: the WCET, ACET, and code size.
3) In contrast to other works, we consider optimizations applied on both abstraction levels of the code, the source code and assembly level, allowing the full exploitation of the optimization potential.
4) In a first large study, different evolutionary multi-objective optimizers are evaluated. Since the comparison of their performance is not straightforward, we conduct a performance assessment based on reliable statistical approaches.
5) To validate the effectiveness of the discovered optimization sequences, a cross-validation on a test set of benchmarks is conducted, allowing to predict how effective these sequences will be on unseen programs.

The rest of this paper is organized as follows. Section II gives a survey of related work. In Section III, concepts of adaptive compilers used for a search of the compiler optimization level space as well as the considered objective functions are discussed. Evaluating the objectives generates data that is used by evolutionary optimizers to explore the large multi-objective search spaces. These algorithms and statistical approaches for their performance assessment are presented in Section IV. Section V introduces our experimental environment, while results achieved on real-life benchmarks are discussed in Section VI. Finally, Section VII concludes the paper and gives directions for future work.

II. Related Work

The search for good compiler optimization sequences, also called iterative compilation, has been thoroughly studied in the past. The general idea behind iterative compilation is to explore the compiler optimization space by starting with a set of randomly chosen optimization sequences used to generate a binary executable. Random sequences are used since good sequences as starting point are usually not known. Measuring a single objective function, e. g., the ACET [1], [2], [3] or code size [4], the fitness of each sequence is determined and subsequent generations of optimization sequences yielding a higher fitness are computed. To reduce the cost of iterative compilation resulting from the search in the large space, Kulkarni [1] uses genetic algorithms to avoid an exhaustive search. To accelerate the search, Leather [2] applies fixed sampling plans. Agakov [3] uses machine learning approaches to focus on promising areas of the search space.

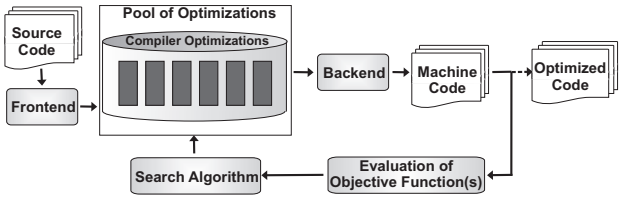

Figure 1. Workflow of an Adaptive Compiler

All these aforementioned publications consider a single objective function. This approach is, however, not sufficient for modern embedded systems where a trade-off between different, conflicting optimization criteria is required. The only work addressing compiler optimization level exploration was presented by Hoste [5]. However, Hoste's and our work differ in several ways. Most important, our main focus is the worst-case behavior of real-time systems, thus the trade-off between the WCET and other crucial objective functions (ACET, code size) is evaluated. Moreover, we don't rely on the performance of a single EMO algorithm, but evaluate different algorithms by a statistical performance assessment to find that algorithm that performs best for a multi-objective exploration of compiler optimizations.

WCET-aware compilation is a novel research area with an increasing academic and industrial interest as the number of embedded systems acting as real-time systems is rapidly growing. Similar to the traditional compilation, the published works in the context of the WCET-aware compiler optimizations consider a single objective function, the WCET. For example, the authors of [6] presented an algorithm for static locking of I-caches based on a genetic algorithm. Other works regard a WCET-aware software based cache partitioning for multi-task systems [7] or a WCET-aware register allocation [8]. Besides, fast scratchpad memories (SPM) for WCET minimization have bee exploited [9].

All these works have in common that novel optimizations driven by WCET data are applied to achieve a WCET minimization. However, none of them studies the impact of standard ACET compiler optimizations on the program's worst-case performance. The only work addressing this gap was presented in [10]. The authors apply a genetic algorithm to find a sequence of standard assembly level optimizations yielding the highest WCET minimization. However, in contrast to our work the authors focus on a single objective function to be optimized and do not consider trade-offs with other objectives. Moreover, just a single evolutionary algorithm is applied. Finally, exclusively assembly level optimizations are considered, neglecting the evaluation of source code optimizations on the program's WCET.

III. Compiler Optimization Sequence Exploration

This section discusses the exploration of the compiler optimization sequence search space. In Section III-A, we briefly introduce the general structure of adaptive compilers.

Section III-B provides an overview of the adaptive WCET-aware C compiler *WCC* [11] which is employed for our experiments. The compiler generates optimization sequences for the search of promising solutions via evolutionary algorithms. Optimization sequence encoding and their performance evaluation are presented in Section III-C and III-D, respectively. Based on this information, evolutionary multi-objective algorithms, which will be discussed in the next section, select promising sequences with conflicting goals.

A. Adaptive Compilers

The general workflow of an adaptive compiler is depicted in Figure 1. Similar to standard compilers, the source code is translated by a compiler frontend into an intermediate representation enabling an easier application of optimizations. However, in contrast to standard compilers, the optimizations are not performed in a fixed order. The search algorithm selects optimization sequences (of arbitrary order) that are exploited for code generation. Next, the code is evaluated and one or more objective functions are determined depending on whether a single- or multi-objective optimization is applied. Subsequently, the determined objective functions serve as input for the search algorithm that refines its selection of optimization sequences by choosing those optimizations for the next generation that exhibit an improved performance. This process is repeated until a termination condition is satisfied. Finally, the best optimization sequence is applied to generate optimized code.

Due to the enormous number of supported optimizations within modern compilers, the main problem with iterative compilation is the large search space making an exhaustive evaluation infeasible.

B. Structure of the WCC Compiler

The adaptive WCET-aware compiler WCC used in this work differs in two major aspects from other adaptive compilers. First, it is tightly coupled to a static WCET analyzer, the tool *aiT* [12], allowing an efficient estimation of the program's WCET in a transparent manner. Second, the compiler is more flexible than other compilers since it allows an arbitrary order of *equivalent* optimizations as will be explained in the following.

Internally, the input program is managed by three different intermediate representations (IRs). After processing the input by a compiler frontend, it is transformed into a high-level intermediate representation. Using a code selector, the source code level is lowered into assembly level by translating the high-level IR into a virtual low-level IR. *Virtual* means that no physical registers but place holders identifying dependencies among instructions are used. These registers are not restricted in their number, thus provide a higher flexibility for the optimizations. Next, a register allocation assigns each virtual register a physical CPU register, thus the virtual low-level IR is translated into a physical one. The latter is used by the compiler backend to generate the final machine code. This compiler structure yields high optimization potential since analyses and optimizations can be performed on different abstraction levels of the code. Consequently, the available optimizations are subdivided into equivalent classes according to the three different compiler's intermediate representations.

Available Compiler Optimizations

The optimizations available within WCC are both standard average-case execution time (ACET) optimizations and WCET-driven optimizations aiming at an automatic improvement of the worst-case performance. In this study, we exclusively focus on the ACET optimizations for two reasons. First, we want to explore the impact of standard compiler optimizations on the program's worst-case performance to show which trade-offs w. r. t. other objectives can be achieved. Using standard optimizations, the results of this study are more general and allow to draw conclusions for similar (standard) compiler frameworks. Second, WCET-aware optimizations are typically too time-consuming since they perform a costly static WCET estimation multiple times to keep their worst-case timing model up-to-date. This makes them not suitable for an iterative search.

WCC provides 21 standard source code optimizations that are applied on the high-level IR. In addition, some of the optimizations are parametric. For example, function inlining allows the specification of the maximal size of the callee function to be inlined. We consider each optimization used with a different parameter as a distinct optimization. In total, 30 source code optimizations are distinguished: 18 non-parametric and 3 parametric (4 parameters each).

The next class of optimizations are assembly level optimizations operating on a virtual low-level IR. The total number of distinguished optimizations is 9. The register allocation supports two different modes: a standard graph coloring based register allocation and a parametric optimal allocation leading to three different choices in total for this optimization class. Finally, a local instruction scheduling can be applied on the physical low-level representation. For details on the optimizations, refer to [13].

Based on this data, WCC's compiler optimization level search space consists of $30^{30} * 9^9 * 3 * 2$ ($\approx 10^{53}$) possible permutations. This huge number emphasizes that an exhaustive search is beyond any feasible computation.

C. Encoding of Optimization Sequences

According to Figure 1, the search algorithm maintains a population of optimization sequences. After compiling the code, the objectives are determined and depending on their values, some sequences are selected for the next generation. In addition, an evolutionary algorithm, inspired by biological evolution, performs the operators *mutation* and *crossover* to generate the next generation. For the exploration of compiler

optimization sequences, genetic algorithms are typically used. Each sequence is encoded as a string where each character denotes a specific optimization.

Using this string encoding, our algorithm performs a one-character mutation by randomly choosing an optimization class and within this class one character (optimization) is randomly replaced by another optimization. Note that mutation might result in classes where same characters (optimizations) occur multiple times. Such optimization sequences are intended since equal optimizations applied at different positions in the optimization chain might have a different impact on the code, thus such sequences represent unequal individuals. The second variation operator crossover is performed in a standard, well known manner by swapping two strings at a randomly chosen position.

Hence, the goal of the genetic algorithm is twofold. First, the algorithm specifies which optimizations are included in each sequence. Second, it defines for each sequence the order of performed optimizations in each class. In contrast to WCC's standard optimization levels, the order is arbitrary.

D. Objective Functions

The search algorithm requires information about the objective values when a particular optimization sequence is applied. Since we are interested in the worst-case behavior of real-time systems, the WCET has to be estimated for each generated machine code. This objective is provided by a static WCET analyzer which is tightly integrated into the WCC compiler. The analyzer does not run the program but performs static program analyses to estimate the WCET. This data is automatically made available to the compiler. Further objectives used to construct the Pareto fronts are the program's ACET and the resulting code size. The ACET is determined by an instruction set simulator, while the code size can be easily extracted from the binary executable.

IV. Multi-Objective Exploration of Compiler Optimizations

The discussion of the multi-objective exploration of compiler optimization sequences presented in this section begins with an introduction to this topic and a definition of basic terms in Section IV-A. The main characteristics of different popular EMO algorithms applied in this study are briefly presented in Section IV-B. Since the comparison of the quality of EMO algorithms for a specific problem, such as the compiler optimization level exploration, is not trivial, a performance assessment based on statistical methods is performed. Principles of the performance assessment are provided in Section IV-C.

A. Multi-Objective Optimization

In many real-life problems, the considered objectives exhibit conflicts. In case of code generation for embedded real-time systems, a trade-off between the WCET, ACET, and code size has to be taken into account. As a consequence, optimizing the application w. r. t. a single objective might yield unacceptable results for other objectives, thus an ideal multi-objective solution simultaneously optimizing each objective does not exist. To cope with this problem, a set of solutions is determined having the characteristics that on the one hand each solution satisfies the objectives at a tolerable level and on the other hand none of the solutions is dominated by another solution. Solutions meeting these characteristics are called *Pareto optimal* solutions.

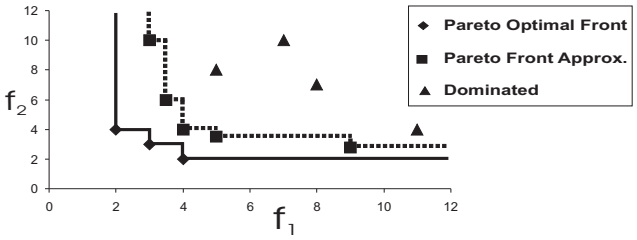

Figure 2. Pareto Fronts

Pareto Front Approximation

Without a loss of generality, we assume that all objectives are to be minimized. A translation into a maximization problem can be easily achieved by multiplying the objectives by -1. Furthermore, we assume that K represents the number of objectives under consideration, X denotes the search space and Z the objective space. With $f : X \rightarrow Z$, function f assigns each solution vector $x \in X$ a corresponding objective vector $z = f(x) \in Z$. Typically, f comprises one or more *objective functions* $f_1, f_2, ..., f_k$ with $f = (f_1, ..., f_k)$. If all objective functions are to be minimized, a feasible solution x *dominates* another feasible solution $y \in X$, if and only if, $f_i(x) \leq f_i(y)$, for $i = 1, ..., K$ and $f_j(x) < f_j(y)$ for at least one objective function j. The dominance relation $(x \prec y)$ is called *Pareto dominance*. Solutions that are not dominated by any other solution in the objective space are called *Pareto optimal*. They can not be improved w. r. t. any other objective function without worsening at least one of the other objectives. In the context of the compiler optimization sequence exploration, an optimization sequence is called Pareto optimal if there are no other optimization sequences that achieve a better performance for all objective functions.

The entirety of all Pareto optimal solutions in the search space constitutes the *Pareto optimal set*, while the set of all Pareto optimal objective vectors is referred to as the *Pareto optimal front*. In practice, the number of Pareto optimal solutions is often too large and the determination of a single Pareto optimum might be even *NP hard* [14]. Additionally, a proof of optimality is computationally infeasible for many real-life problems. Therefore, the goal is to find a *Pareto front approximation* that minimally deviates from the Pareto optimal front. The relationship between a Pareto optimal front, its approximation, and dominated solutions for a minimization problem involving two objective functions f_1

and f_2 is depicted in Figure 2.

The Pareto front approximation can be finally used by to find suitable optimization levels. By constructing an approximation set for a large number of benchmarks, particular points from this set that satisfy given trade-offs between the considered objectives can be chosen. The optimization sequences that represent these points will be implemented as optimization levels into the compiler and can be used in the future for new applications. This is also the scenario that we address in this paper. Moreover, Pareto front approximations can be also used to tune a single application.

B. Evolutionary Multi-Objective Algorithms

In the past, it was shown that randomized evolutionary multi-objective algorithms are best suited for the approximation of Pareto fronts. The algorithms basically differ in the fitness assignment, their strategy to maintain elitist solutions which will survive in the next generation, and their promotion of diversity, i. e., if a uniform distribution of solutions over the Pareto front can be attained.

In this study, we are interested in the evolutionary algorithm that performs best for our compiler problem. Other works [5] studying the impact of multi-objective optimizations in the context of iterative compilation explored a **single** EMO algorithm. Thus, it is not clear if the selected algorithm is suitable or if another optimizer would perform better in this problem domain. To cover a broad spectrum of principles used by evolutionary algorithms, we conduct a large study where three popular and credible algorithms, which have been exploited for different application domains in the past, are evaluated. The following algorithms are used: *Indicator Based Evolutionary Algorithm (IBEA)* [15], *Non-dominated Sorting Genetic Algorithm 2 (NSGA-II)* [16], and *Strength Pareto Evolutionary Algorithm 2 (SPEA-2)* [17]. These state-of-the-art optimization algorithms were chosen since each of them exhibits a different functionality.

C. Statistical Performance Assessment

The typical dilemma with multi-objective optimizations is to find a suitable multi-objective algorithm for a given problem. Modules that serve as a representation of a particular problem as well as for the evaluation and variation of the solutions are called *variators*. To approximate Pareto optimal solutions, each of these modules can be arbitrarily combined with any evolutionary multi-objective algorithm. This leads to the question: which combination performs best for a given scenario?

For our scenario dealing with the exploration of compiler optimizations, a manual combination and evaluation of the WCC compiler and the considered EMO algorithms is time-consuming and error-prone. Moreover, a reliable comparison of the quality of the stochastic multi-objective optimizers is not trivial. An example are crossing Pareto fronts where a visual comparison is not intuitive anymore. Therefore, automatic and reliable performance assessment is required.

Since many EMO algorithms, as well as those that we consider in this study, are based on a randomized search, the evaluation of the approximated Pareto optimal solutions generated for a specific seed is not sufficient. To deal with the stochastic nature of the algorithms, each algorithm has to be run multiple times for each problem with different seeds to generate a sample of different Pareto approximation sets which can be statically analyzed, i. e., a statistical hypothesis testing is conducted to indicate if the results are *significantly different* [14]. A result is considered significantly different if it is unlikely that it occurred by chance. A measure of evidence to accept that the result is unlikely to have arisen by chance is known as the *significance level* α. A widely used value for α is 5%.

For statistical testing, we apply *dominance ranking* [14]. Its main idea is to rank the points of the approximation sets based on the dominance relation. The approximation sets of all considered optimizers are collected into a pool and each set z is assigned a rank representing its dominance relation, i. e., how many sets in the pool are dominated by z. The lower the rank, the better the considered set is w. r. t. the pool. Finally, ranks between the optimizers are compared by statistical means to determine statistical significance.

V. Experimental Environment

To indicate the efficacy of the found multi-objective optimization sequences, we performed an evaluation on a large number of real-life benchmarks using a *cross-validation*. One set of benchmarks, the *training set*, is used during the multi-objective search. The determined sequences are subsequently evaluated on unseen benchmarks, the *test set*. This approach enables an estimation of the generalization ability, i. e., the results suggest which improvements of the objective functions can be expected for unseen programs.

Benchmarking was performed on programs from the suites DSPstone, MediaBench, MiBench, UTDSP, NetBench, and MRTC WCET Benchmarks. The large variety of benchmark suites emphasizes our focus on generality. Covering a large number of different service-oriented applications, future software should benefit in a similar fashion from our optimization sequences. For our study, the training and test set each contain 35 arbitrarily selected benchmarks.

The workflow for the multi-objective exploration of compiler optimizations is depicted in Figure 3. As mentioned in Section III, the WCET-aware C compiler WCC for the Infineon TriCore TC1796 processor is used to generate code transformed by different optimizations. Besides the previously discussed intermediate code representations, it can be seen that different optimizations are applied at different abstraction levels of the code. WCET-aware optimizations are excluded for previously mentioned reasons. The considered C programs are annotated with *flow facts*. This data

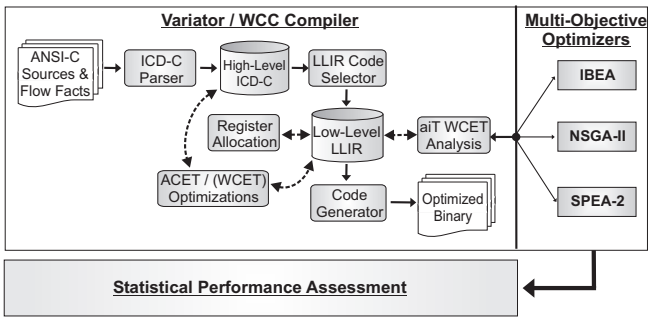

Figure 3. Workflow for the Multi-Objective Exploration of Compiler Optimizations

provides information about the code structure, such as the number of loop iterations, and is mandatory for a static WCET analysis. During optimization, WCC automatically takes care that flow facts are kept valid and consistent.

The communication between WCC via a variator and the different evolutionary multi-objective algorithms (cf. right-hand side of Figure 3) is fully automated. The process starts with a random generation of compiler optimizations representing the initial population. For each selected optimization sequence, the considered objectives are determined and passed to the EMO algorithms which in turn compute the approximated Pareto front based on the Pareto dominance. These approximation sets are retrieved by the variator which uses the selected Pareto solutions to generate the next generation of optimization sequences. These sequences are passed to the compiler to generate optimized code. In addition, the evolutionary algorithms provide the approximated Pareto front for the statistical performance assessment. Both, the collection of the evolutionary optimizers and the performance assessment are part of the *PISA* framework [18].

The following experimental parameters were used. Each EMO algorithm was invoked 5 times with a different random seed. For each run, we consider one population consisting of 50 individuals and an archive size (set of elitist solutions from previous runs used for creation of new generations) of 25 individuals. The evolutionary algorithms are each run for 50 generations. Each full run takes about 6 days on an Intel Quad-Core Xeon 2.4 GHz machine. These optimization times might seem long. However, it should be noted that these tests have to be performed once off-line, while the results (optimization sequences) can be re-used without additional overhead for a large number of devices. Therefore, the high performance requirements imposed on today's systems fully justify the observed optimization times. To breed a new generation of optimizations, we use a one-character mutation probability of 10% and a one-point cross-over probability of 90%. These are common settings for the exploration of the compiler optimization space [10].

VI. Results

The multi-objective optimizations are carried out for pairs of objective functions. The consideration of 2-dimensional Pareto fronts is intentional. Since the impact of standard optimizations is unknown so far for the trade-off between the WCET and other objectives, this paper is the first case study which investigates this issue. The results help to understand the fundamental interferences between the objectives. Starting with the investigation of more than two objectives may hide some objective interferences leading to a lack of the fundamental understanding.

Table I
Dominance Ranking Results for WCET&ACET and WCET&Code Size using Mann-Whitney Rank Sum Test

	WCET↔ACET			*WCET↔Code Size*		
	IBEA	**NSGA-II**	**SPEA2**	**IBEA**	**NSGA-II**	**SPEA2**
IBEA	—	0.760	0.949	—	0.5	0.5
NSGA-II	0.240	—	0.011	0.5	—	0.016
SPEA2	0.051	0.899	—	0.5	0.984	—

Statistical Evaluation

Table I presents results for dominance ranking using the *Mann-Whitney* rank sum test for the possible combinations of the considered algorithms IBEA, NSGA-II, and SPEA2. Columns 2-4 indicate results for the objective functions *WCET↔ACET*, while columns 5-7 present results for the objectives *WCET↔code size*. The statistical tests are performed pairwise using a significance level α=0.05. The tests are performed w. r. t. the *alternative hypothesis* that the dominance ranks for the algorithms in the first column are significantly better than those for the algorithm in the following columns. The results are expressed by the probability value, called *p-values*, which allow to draw conclusions about the statistical significance: if the p-value is less than the significance level α, then the null hypothesis is rejected. This implies that the alternative hypothesis can be accepted, i. e., there is a statistically significant difference between the ranking of the corresponding algorithms.

For the objectives *WCET↔ACET*, the p-value of 0.011 in the fourth row and fourth column denotes that the difference between NSGA-II and SPEA2 is significant, i. e., NSGA-II outperforms SPEA2. For other optimizer pairs, no significant differences were observered. The results for the objectives *WCET↔code size* lead to the same conclusion. NSGA-II outperforms SPEA2 since the dominance ranking results significantly differ (p-value=0.016) for α=5%. Hence, NSGA-II seems to be the most promising EMO for the given problems. There are also differences between other combinations of the algorithms but they are not significant. These results are conform with results reported in [19].

Dominance ranking is computationally cheap since it is based on simple sorting of the approximation sets. Therefore, the run time of this approach is negliable.

Analysis of Pareto Front Approximations

Figure 4 visualizes the Pareto front approximation generated by the algorithm NSGA-II which achieved best performance assessment results for the objective functions WCET

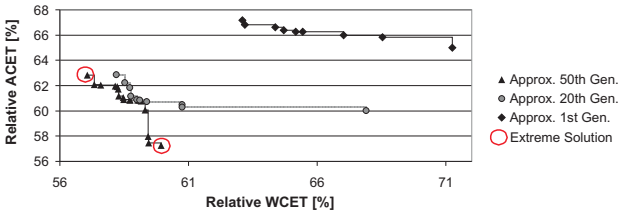

Figure 4. NSGA-II Pareto Front Approximation for WCET↔ACET

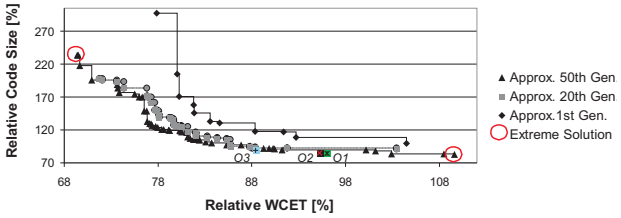

Figure 5. NSGA-II Pareto Front Approximation for WCET↔Code Size

and ACET. The horizontal axis indicates the relative WCET w. r. t. the non-optimized code, i. e., 100% represents the WCET with all disabled WCC optimizations. In a similar fashion, the vertical axis represents the relative ACET w. r. t. to the non-optimized code. Furthermore, the figure shows the results of the Pareto front approximation for the 1st, 20th, and 50th generation. Based on this figure, the following can be concluded:

- It is worthwhile to invest time in the evolutionary search. While the first generation achieves WCET and ACET reductions of 36.9% and 35.0%, respectively, on average for all benchmarks of the training set, the 50th generation reduces the WCET and ACET of up to 42.9% and 42.8%, respectively.
- The discovered sequences significantly outperform the standard optimization levels having the coordinates (due to space constraints not included in the figure) *O1*:(96.0, 89.1), *O2*:(95.9, 90.4), and *O3*:(88.4, 84.7). For example, *O3* is outperformed by 31.3% and 27.5% for WCET and ACET, respectively.
- Standard compiler optimizations have a similar impact on the WCET and ACET. This observation provides an important answer to the question which concerns all designers of real-time systems: *which impact can be expected from standard ACET optimizations on the system's worst-case behavior?* Our case study finally eliminates this uncertainty showing that similar effects on the average-case and worst-case behavior are likely.
- The results emphasize the importance of the development of WCET-driven optimizations. If a high WCET minimization is desired, novel optimizations are required which focus on an aggressive WCET reduction at the cost of a degraded ACET.

The Pareto front approximation computed by NSGA-II for the objectives WCET and code size is depicted in Figure 5. The relative WCET w. r. t. the non-optimized code (corresponds to 100%) is represented by the horizontal axis, while the relative code size w. r. t. to the non-optimized code is shown on the vertical axis. Again, Pareto front approximations of the 1st, 20th, and 50th generation are visualized. Compared to the Pareto front approximations for *WCET↔ACET*, the interpretation equals in two points:

- The evolutionary search pays off for both objective functions. For the first generation, a WCET reduction of 21.2% at the cost of the code size increase of 197.4% is achieved. If code size is the crucial objective, a code size reduction of 0.4% with a simultaneous WCET increase of 4.5% can be observed. For the 50th generation, the following *extreme solutions* were observed: a WCET reduction of 30.6% with a simultaneous code size increase of 133.4%, or a code size reduction of 16.9% with a WCET degradation of 9.6%.
- The Pareto solutions outperform WCC's standard optimization levels depicted in Figure 5. The standard optimization levels perform well for the code size reduction. Using *O2*, which does not include code expanding optimizations, a code size reduction of 14.9% can be achieved on average, while NSGA-II reduces the code size by up to 16.9%. Moreover, WCC's maximal WCET reduction of 13.6% found by *O3* can be outperformed by the found Pareto solutions by 17.0%, amounting to a WCET reduction of 30.6%.

However, there is also one **major difference** compared to the results of the objective pair *WCET↔ACET*. The WCET and the code size are typical conflicting goals. If a high improvement of one objective function is desired, a significant degradation of the other objective must be accepted. This is an important conclusion for memory-restricted real-time systems. To achieve a high WCET reduction, the system must be possibly equipped with additional memory to cope with the resulting code expansion.

An analysis of the optimization sequences which represent extreme solutions of the Pareto front approximation revealed that for *WCET↔ACET* there are no distinct optimizations which account for the good performance. In contrast, for the objective pair *WCET↔code size*, significant size expansions are often due to aggressively applied loop unrolling.

Cross-Validation

To estimate the generalization ability of the discovered sequences, we perform a cross-validation, i. e., we apply optimization sequences found by NSGA-II in the 50th generation for the training set on unseen benchmarks from the test set. For each objective pair, we picked out three different sequences: two sequences which represent the *extreme solutions* of the Pareto front approximation, i. e., Pareto solutions located at each end of the front approximation, and one point from the middle of the front. All measurements are compared with the performance achieved when benchmarks from the test set are compiled with the highest optimization

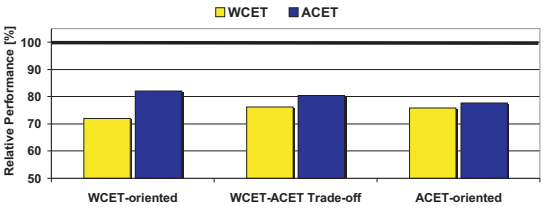

Figure 6. Cross-Validation for *WCET↔ACET*

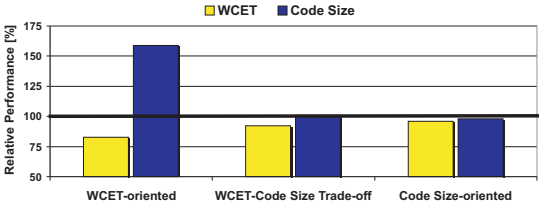

Figure 7. Cross-Validation for *WCET↔Code Size*

level *O3*. Figure 6 shows the results for *WCET↔ACET*. The figures present the average performance achieved for all benchmarks in the test set. Using the WCET-oriented optimization sequence, which is represented in Figure 4 by the Pareto solution with the *WCET,ACET* coordinate $(57.06, 62.82)$, our optimization sequences can outperform *O3* by 28.0% and 18.0% for the WCET and ACET reduction, respectively. The optimization sequence, denoted as *WCET-ACET Trade-off*, was determined by NSGA-II as a compromise between the WCET and ACET for the training set (cf. coordinate $(58.7, 60.9)$ in Figure 4). For the test set, the results are slightly worse, since a relative WCET estimation of 76.2% and a relative ACET of 80.5% were observed. However, the results still outperform *O3*. The ACET-oriented scenario for the test set could even improve both objectives compared to the trade-off. Analogously, Figure 7 reflects the performance of the sequences found for *WCET↔code size* in Figure 5. Either we choose a balanced compromise between the WCET and code size and slightly outperform *O3* (see bars labeled as *WCET-Code Size Trade-off*) or we focus on the reduction of one objective but also tolerate a degradation of the other one.

VII. Conclusions

The search for good compiler optimization sequences is challenging since optimizations exhibit complex interactions which have unpredictable effects on different objective functions. We propose the first adaptive WCET-aware compiler framework for service-oriented real-time systems which automatically finds Pareto optimal solutions that represent trade-offs between the WCET, ACET, and code size. To find the best evolutionary multi-objective optimizer for the search of the compiler optimization space, a statistical performance assessment is performed. The discovered optimization sequences significantly outperform standard optimization levels: the highest standard optimization level *O3* can be outperformed for the WCET and ACET on average by up to 31.33% and 27.43%, respectively. Moreover, for a trade-off between the WCET and code size, an improvement of one objective function can be only achieved at the cost of the other one. Providing the discovered optimization sequences, compiler writers and compiler users can select those solutions that best suit their system specifications.

In the future, we want to explore Pareto fronts with more than two dimensions. In addition, energy dissipation as a further objective will be included for the approximation of Pareto fronts. We also intend to investigate the performance of further EMO optimizers and the impact of different variator settings, e. g., mutation probability, on Pareto fronts.

References

[1] P. A. Kulkarni, S. R. Hines, D. B. Whalley *et al.*, "Fast and Efficient Searches for Effective Optimization-phase Sequences," *Transactions on Architecture and Code Optimization*, 2005.

[2] H. Leather, M. O'Boyle, and B. Worton, "Raced Profiles: Efficient Selection of Competing Compiler Optimizations," in *Proc. of LCTES*, 2009.

[3] F. Agakov, E. Bonilla, J. Cavazos *et al.*, "Using Machine Learning to Focus Iterative Optimization," in *Proc. of CGO*, 2006.

[4] K. D. Cooper, P. J. Schielke, and D. Subramanian, "Optimizing for Reduced Code Space using Genetic Algorithms," *SIGPLAN Not.*, vol. 34, no. 7, 1999.

[5] K. Hoste and L. Eeckhout, "COLE: Compiler Optimization Level Exploration," in *Proc. of CGO*, 2008.

[6] A. M. Campoy, I. Puaut, and A. P. I. et al., "Cache Contents Selection for Statically-Locked Instruction Caches: An Algorithm Comparison," in *Proc. of ECRTS*, 2005.

[7] S. Plazar, P. Lokuciejewski, and P. Marwedel, "WCET-aware Software Based Cache Partitioning for Multi-Task Real-Time Systems," in *Proc. of WCET*, 2009.

[8] H. Falk, "WCET-aware Register Allocation based on Graph Coloring," in *Proc. of DAC*, 2009.

[9] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen, "WCET Centric Data Allocation to Scratchpad Memory," in *Proc. of RTSS*, 2005.

[10] W. Zhao, P. Kulkarni, D. Whalley *et al.*, "Tuning the WCET of Embedded Applications," in *Proc. of RTAS*, 2004.

[11] H. Falk, P. Lokuciejewski, and H. Theiling, "Design of a WCET-Aware C Compiler," in *Proc. of ESTIMedia*, 2006.

[12] AbsInt Angewandte Informatik GmbH, "Worst-Case Execution Time Analyzer aiT," http://www.absint.com/ait, 2009.

[13] S. S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., 1997.

[14] J. Knowles, L. Thiele, and E. Zitzler, "A Tutorial on the Performance Assessment of Stochastic Multiobjective Optimizers," ETH Zurich, Switzerland, Tech. Rep., 2006.

[15] E. Zitzler and S. Künzli, "Indicator-Based Selection in Multiobjective Search," in *Proc. of PPSN*, 2004.

[16] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan, "A Fast Elitist Non-dominated Sorting Genetic Algorithm for Multi-objective Optimisation: NSGA-II," in *Proc. of PPSN*, 2000.

[17] E. Zitzler, M. Laumanns, and L. Thiele, "SPEA2: Improving the Strength Pareto Evolutionary Algorithm for Multiobjective Optimization," in *Proc. of EUROGEN*, 2002.

[18] S. Bleuler, M. Laumanns, L. Thiele, and E. Zitzler, "PISA — A Platform and Programming Language Independent Interface for Search Algorithms," in *Proc. of EMO*, 2003.

[19] S. Künzli, S. Bleuler, L. Thiele *et al.*, *Application of Multi-Objective Evolutionary Algorithms*. World Scientific, 2004.