SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. (2011)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.1079

Approximating Pareto optimal compiler optimization sequences—a trade-off between WCET, ACET and code size

Paul Lokuciejewski[1,*,†], Sascha Plazar[1], Heiko Falk[1], Peter Marwedel[1] and Lothar Thiele[2]

[1]*Computer Science 12, TU Dortmund University, D-44221 Dortmund, Germany*
[2]*Computer Engineering and Networks Laboratory, ETH Zurich, CH-8092 Zurich, Switzerland*

SUMMARY

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. Owing 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 worst-case execution time (*WCET*)-aware compiler framework for an automatic search of compiler optimization sequences that yield highly optimized code. Besides the objective functions ACET and code size, we consider the 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 for the objectives ⟨WCET, ACET⟩ and ⟨WCET, code size⟩ are exploited. A comparison based on statistical performance assessments is performed that 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. Copyright © 2011 John Wiley & Sons, Ltd.

Received 11 July 2010; Revised 22 December 2010; Accepted 21 February 2011

KEY WORDS: real-time; WCET; compiler; optimization; multi-objective; Pareto optimal

1. 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 (ACET), 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

*Correspondence to: Paul Lokuciejewski, Computer Science 12, TU Dortmund University, D-44221 Dortmund, Germany.
†E-mail: paul.lokuciejewski@tu-dortmund.de

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 [1–4] 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 (WCET). 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 synergistic manner. Concerning the compiler-based code generation, there is no single compiler optimization sequence that satisfies all objectives. Therefore, multiple trade-offs must be considered enabling the system designer to choose among different solutions that 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 threefold. 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 performance compared with the traditional system design. Third, the computation of good optimization sequences can be done once using representative test code, and these pre-computed sequences can afterwards be applied online to unseen codes without any further computational overhead.

This paper is an extension of the ISORC conference paper entitled *Multi-Objective Exploration of Compiler Optimizations for Real-Time Systems* [5]. Its main contributions 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: we perform trade-offs and compare the objectives ⟨WCET, ACET⟩ and ⟨WCET, 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 the first large study, different EMO 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 remainder of this paper is organized as follows. Section 2 gives a survey of the related work. In Section 3, 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

DOI: 10.1002/spe

in Section 4. Section 5 introduces our experimental environment, while results achieved on real-life benchmarks are discussed in Section 6. Finally, Section 7 concludes the paper and gives directions for future work.

2. 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 [1–3] 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 *et al.* [1] use genetic algorithms to avoid an exhaustive search. In [6], a characterization of the search space by enumerations and explorations is used to find good compilation sequences more efficiently. Enumerations examine a subset of the optimizations and evaluate each point in that subspace for a single program and a sample input. Explorations use a specific search algorithm to find good sequences for a variety of programs under some objective function. To accelerate the search, Leather *et al.* [2] apply fixed sampling plans. In [7], a framework statically predicting the impact of applied optimizations is presented. Agakov *et al.* [3] use machine learning approaches to focus on promising areas of the search space.

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 and Eeckhout [8]. 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 do not rely on the performance of a single EMO algorithm, but evaluate different algorithms by a statistical performance assessment to find the algorithm that performs the best for a multi-objective exploration of compiler optimizations. In addition, our framework is more flexible. In contrast to the GNU Compiler Collection (GCC) compiler used by Hoste and Eeckhout [8], which performs all optimizations in a fixed order where each optimization can be switched on or off, our compiler allows the construction of arbitrary optimization sequences. This leads to an increased complexity of the problem but allows also the exploitation of a higher optimization potential.

In contrast to traditional ACET 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 ACET compilation, the published works in the context of the WCET-aware compiler optimizations consider a single objective function, the WCET. Most works in the domain of WCET minimization operate on assembly level and exploit memory hierarchies. For example, the authors of [9] presented an algorithm for static locking of I-caches based on a genetic algorithm, while compile-time cache analysis combined with static data cache locking was presented in [10]. Other works regard a WCET-aware software-based cache partitioning for multi-task systems [11] or a WCET-aware register allocation (RA) [12].

Further studies exploit fast scratchpad memories (SPM) for WCET minimization. Greedy algorithms for a WCET-aware SPM allocation of data are presented in [13], while optimal approaches based on an ILP formulation are explored for data and program code in [14, 15], respectively.

Besides the exploitation of memory hierarchies to reduce WCETs, Kadlec *et al.* [16] propose code transformations applied by a compiler in order to avoid timing anomalies of modern processor hardware. The authors observe that out-of-order pipelines of processors might lead to timing anomalies. They propose to move all scheduling decisions from the hardware instruction scheduler to the compiler's instruction scheduler and propose three code re-scheduling techniques to avoid timing anomalies.

DOI: 10.1002/spe

A complete compiler framework for the reduction of WCETs called *WCC* is presented in [17]. It consists of a complex infrastructure aiming at the tight and fully automated integration of a WCET analyzer into the compilation process. For this purpose, interfaces at machine code level have been exploited in order to establish communication between compiler and analyzer. Mechanisms have been integrated into this compiler in order to maintain the so-called *flow facts* which are required for the WCET analysis, and a loop analyzer is able to derive such flow facts automatically for large classes of loops. On top of this infrastructure, a couple of WCET-aware optimizations are presented (procedure cloning and positioning, scratchpad allocations for program code and data, and RA). Although the present publication is technically based on this WCC compiler, [17] differs from the present publication in the sense that no adaptive compilation and no multiple objectives beyond WCET have been considered.

All these previous 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 [18]. 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 (EA) is applied; thus, it is not clear how good the algorithm performs and whether other algorithms, like *Hill climbing*, might even produce better results. Finally, exclusively assembly level optimizations are considered, neglecting the evaluation of source code optimizations on the program's WCET.

3. COMPILER OPTIMIZATION SEQUENCE EXPLORATION

This section discusses the exploration of the compiler optimization sequence search space. In Section 3.1, we briefly introduce the general structure of adaptive compilers. Section 3.2 provides an overview of the adaptive WCET-aware C compiler WCC [17, 19], which is employed for our experiments. The compiler generates optimization sequences for the search of promising solutions via EAs. Optimization sequence encoding and their performance evaluation are presented in Sections 3.3 and 3.4, respectively. Based on this information, EMO objective algorithms, which will be discussed in the next section, select promising sequences with conflicting goals.

3.1. 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.

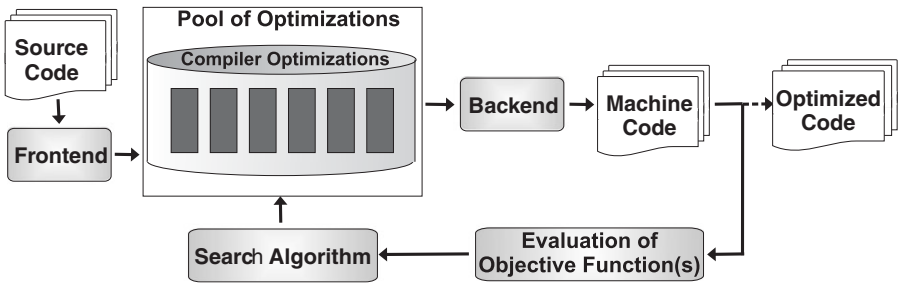

Figure 1. Workflow of an adaptive compiler.

DOI: 10.1002/spe

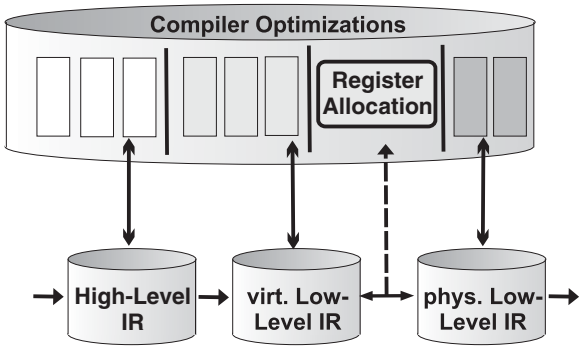

Figure 2. Internal code representation within the WCC compiler.

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 exhibits an improved performance. This process is repeated until a termination condition is satisfied. Finally, the best optimization sequence is applied to generate optimized code.

Owing 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. For example, the GCC v4.1 [20] provides 60 compiler flags, which can be arbitrarily enabled or disabled, yielding 2^{60} possible combinations—a number of combinations that makes an exhaustive evaluation infeasible.

3.2. 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* [21], 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) as shown in Figure 2. 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, an RA 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 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, 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. Second, 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.

DOI: 10.1002/spe

WCC provides 21 standard source code optimizations that are applied to the high-level IR (cf. Figure 2). These optimizations are briefly summarized below. The reader is referred to the standard compiler literature [22] for more details on these optimizations.

Constant folding evaluates constant arithmetic expressions like e.g. `a = 3 * 7;` and replaces them by their pre-computed result `(a = 21;)`.

Dead code elimination removes computations from the code whose results are not used anywhere in the remaining code.

Common subexpression elimination replaces several occurrences of an arithmetic expression (e.g. `a = b * 3 + c; ... b * 3 + c ...`) by accesses to temporary variables in order to avoid the repeated computation of the same expression (`tmp = b * 3 + c; a = tmp; ... tmp ...`).

Merging of identical string constants like e.g. `"\t%d\n"` in `printf("\t%d\n", i); printf("\t%d\n", j);` helps to reduce a program's data segment.

Code simplifications e.g. eliminate double negations, remove unnecessary type casts or pointer dereferences, or translate simple if-then-else statements into the ANSI-C select operator `?`.

Value propagation propagates constant values in the code (e.g. `a = 21; b = a + 14;` becomes `a = 21; b = 21 + 14;`) in order to create more optimization potential for constant folding.

Creation of multiple function exits avoids unnecessary jumps by moving return statements in the then- and else-parts of if-then-else statements if possible.

Life range splitting creates new local variables for any single local variable that is used for different purposes, i.e. that has distinct life ranges. This increases the degree of freedom for other optimizations and RA.

Loop collapsing transforms n-fold nested loops iterating over n-dimensional arrays to a one-dimensional loop in order to reduce repeated tests of loop exit conditions and conditional branching.

Loop deindexing replaces well-structured accesses to arrays inside loops by pointer accesses and pointer arithmetic using the `++` and `--` operators in order to support the auto-increment/decrement addressing modes of embedded processors.

Loop unswitching moves if-statements whose conditions are not loop-dependent out of loops. This helps to achieve a more linear and regular control flow in loop bodies.

Optimization of if-statements in loop nests removes if-statements whose conditions are provably true or false during all loop iterations. Again, this helps to simplify the control flow in loop bodies.

Removal of unused function arguments helps to keep parameter lists of functions small and thus potentially reduces function calling overhead.

Removal of return values of a function `f` can be done if it is known that `f`'s return value is never used. This reduces the overhead for returning from function calls.

Removal of unused symbols keeps the compiler's symbol tables small and might result in smaller overall code sizes.

Struct scalarization replaces an ANSI-C `struct` by a couple of atomic scalar variables (e.g. `struct {int x; int y;} point` becomes `int point_x; int point_y;`). This makes the struct's elements eligible for RA so that they do not necessarily have to be stored in the slow main memory.

Tail recursion elimination replaces simple classes of recursive functions by non-recursive functions in order to remove function calling overhead and to increase the potential of subsequent optimizations.

Transformation of head-controlled loops replaces `for`- and `while`-do-loops by `do-while`-loops since this reduces the number of required tests of the loop's exit condition.

Function inlining copies the body of a function to those places where the function is called. This increases code size, but reduces function calling overhead and possibly enables other subsequent optimizations.

Function specialization propagates constants passed to called functions as arguments into the called function and creates a specialized version of the called function for the given

DOI: 10.1002/spe

constant argument. In analogy to inlining, this increases code size, but reduces function calling overhead and possibly enables other subsequent optimizations like e.g. constant propagation and folding.

Loop unrolling by an unrolling factor *n* replicates a loop's body *n* times (e.g. `for (i=0; i<10; i++) a += i;` becomes `for (i=0; i<10; i+=2) { a += i; a += i; }` for *n* equal 2). This reduces loop iteration overhead and possibly enables other optimizations, but obviously increases code size.

Some of the optimizations are parametric. Function inlining, for example, 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. Three optimizations, namely function inlining, function specialization and loop unrolling, use the constant values 20, 50, 100 and 200 to restrict the number of optimized functions and loops, respectively. 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 that operate on a virtual low-level representation of the code. This class includes the following seven optimizations:

Constant folding, dead code elimination and **value propagation** are exactly the same techniques as already explained above, except that they now operate on machine code instead of C code.

Redundant code elimination is comparable to the common subexpression elimination described above. However, WCC's redundant code detection works at a much finer granularity and is able to eliminate all those code regions which, bit per bit, compute exactly the same results.

Peephole optimizations perform simple machine code transformations where e.g. redundant load or store operations are removed or where operations implementing type casts are eliminated.

Loop-invariant code motion is a technique that moves all machine operations that do not depend on a surrounding loop out of this loop. This helps to keep loop bodies small and improves the loop's performance.

Instruction scheduling reorders machine instructions such that the parallelism available in the processor's pipelines is exploited in the best possible way.

Loop-invariant code motion is a parametric optimization depending on one parameter. In total, eight virtual low-level optimizations are distinguished here: six non-parametric and one parametric used in two different configurations.

Following the workflow in Figure 2, RA is applied next. This step can be considered as an optimization but in contrast to other optimizations, its application is not optional but mandatory in order to generate valid code. Our modular adaptive compiler supports different register allocators which can be freely selected; thus, the register allocator is a part of the optimization sequence that can be constructed by the search algorithm. WCC implements a standard graph coloring-based RA [23] and a parametric optimal allocation [24] (using two different allocation strategies) leading to three different choices in total for this optimization class. It should be mentioned that this is the first work that considers different register allocators during iterative compilation. This is important since different spilling strategies could lead to better performance w.r.t. one objective while worsening another. Moving for instance spill code from the worst-case execution path to a concurrent path could decrease the WCET, whereas it potentially increases the ACET of a program.

Finally, a local instruction scheduling can be applied as assembly level optimization on the physical low-level representation. An overview of all considered optimizations classified by WCC's standard optimization levels is depicted in Table I. Higher levels include all optimizations from lower levels. The numbers in parentheses indicate the number of different parameter values used for the corresponding optimization. Note that the order of the optimizations in the table does not correspond to the order in which the optimizations are applied by WCC in the particular optimization levels.

DOI: 10.1002/spe

Table I. Considered compiler optimizations.

O1	Constant folding	O2	Loop deindexing
	Dead code elimination		Loop unswitching
	Local common subexpression elimination		Optimize if-statements in loop nests
	Merge identical string constants		Remove unused function arguments
	Simplify code		Remove unused returns
	Value propagation		Remove unused symbols
	Low-level redundant code elimination		Struct scalarization
	Low-level dead code elimination		Tail recursion elimination
	Low-level loop-invariant code motion (2x)		Transform head-controlled loops
	Low-level constant folding		Instruction scheduling before register allocation
	Low-level value propagation		Instruction scheduling after register allocation
	Low-level peephole optimizations		
O2	Create multiple function exit points	O3	Function inlining (4x)
	Life range splitting		Function specialization (4x)
	Loop collapsing		Loop unrolling (4x)

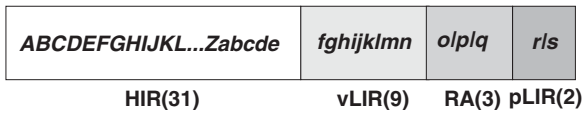

Figure 3. Encoding of optimization sequences.

Based on the optimizations described above, WCC's adaptive compilation techniques might produce various different optimization sequences for different objectives. These sequences might differ in the order in which individual optimizations occur not only within the sequences, but also in the sequences' length. Since EAs are used within WCC to generate these sequences, these algorithms should be able to deal with sequences of variable length. However, evolutionary algorithms usually rely on fixed-length encodings. Choosing a too small length for this encoding makes the EA find solutions of poor performance [25]. Thus, the encoding used within WCC has a sufficient fixed length as described in the next section 3.3. In order to model variable-length optimization sequences within a fixed-length evolutionary encoding, a so-called dummy optimization is introduced per optimization class (except for RA). This dummy optimization does not represent any existing optimization of WCC. Instead, it is simply a placeholder denoting that no actual code optimization should be applied at the position where the dummy is placed within an optimization sequence. In this way, positions in an optimization sequence can be left free by the EAs so that they are able to produce sequences of variable length.

3.3. 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 EA, 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. This problem representation can be easily handled by the genetic algorithm operators.

Figure 3 shows a possible encoding for the optimization classes when all optimizations are chosen in a sorted order. The numbers in parentheses denote the number of possible optimizations. For the classes *RA* and *physical low-level IR* (*pLIR*), only one optimization is encoded in each sequence. Based on this data, WCC's compiler optimization level search space consists of $31^{31} \times 9^9 \times 3 \times 2$ (in the order of $\approx 10^{54}$) possible permutations. This huge number emphasizes that an exhaustive search is beyond any feasible computation.

DOI: 10.1002/spe

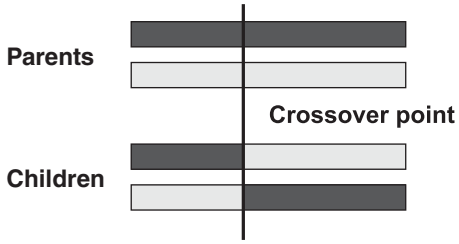

Figure 4. One-point crossover reproduction.

Using this string encoding, the algorithm for the *one-character mutation* operator works as follows:

1. Randomly choose the optimization IR class $Class \in \{HIR, vLIR, RA, pLIR\}$.
2. In $Class$, choose a character c at a random position.
3. Replace c by a character $c' \in Class$, with $c \neq c'$.

Note that mutation might result in sequences where the same character (optimization) occurs multiple times in the string. 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 operator *one-point crossover* is performed in a standard, well-known manner by swapping two strings at a randomly chosen position. The reproduction of two parent chromosomes using a single crossover point is depicted in Figure 4.

The randomness in the evolutionary operators ensures that the genetic algorithm does not get stuck in locally optimal solutions and is likely to reach the global optimum if it is run for a sufficient number of generations.

In summary, the goal of the genetic algorithm to create optimization sequences is twofold. The algorithm

- specifies which optimizations are included in each sequence and whether some positions in the optimization sequences are filled with dummy optimizations having no effect on the code;
- defines for each sequence the order of performed optimizations in each class. In contrast to WCC's standard optimization levels, the order inside each IR class is arbitrary.

3.4. 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 employing the instruction set simulator *CoMET* [26] from Synopsys. CoMET comprises virtual system prototypes for diverse target platforms that enable the simulation of complex embedded systems without the existence of silicon. Benchmarking on real hardware would also not be feasible since it is hardly automatable and would require multiple evaluation boards for parallel execution. Finally, the objective code size can be easily extracted from the binary executable.

To accelerate the evaluation of different objectives, maps are utilized that hold the evaluated objective values for each considered optimization sequence. Whenever an objective of a sequence has to be determined that was already evaluated in the past, a costly re-evaluation is omitted and the objective value is efficiently obtained from the map.

DOI: 10.1002/spe

After the introduction of the adaptive WCET-aware compiler WCC, which demonstrates the communication between the compiler and the search algorithm to find good optimization sequences, the focus of the following section is shifted toward evolutionary search algorithms.

4. 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 4.1. The main characteristics of different popular EMO algorithms applied in this study are briefly presented in Section 4.2. 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 4.3.

4.1. 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 among 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.

Pareto front approximation
Without loss of generality, it should be assumed that all objectives are to be minimized. A translation into a maximization problem can be easily achieved by multiplying the objectives by -1. Pareto optimality, dominance and Pareto sets are formally defined as follows [27]:

Definition 1 (*Pareto optimality, dominance, Pareto set*)
Let X denote the decision space (or search space), Z represents the objective space, $f : X \to Z$ is a function that assigns each decision vector $x \in X$ a corresponding objective vector $z = f(x) \in Z$ and m denotes the number of objectives under consideration. A decision vector $x^* \in X$ is *Pareto optimal* iff there is no other $x \in X$ that dominates x^*. x *dominates* x^*, denoted as $x \succ x^*$, iff $f_i(x) \leqslant f_i(x^*), \forall i = 1, \ldots, m$ and $f_i(x) < f_i(x^*)$ for at least one index i. The set of all Pareto optimal decision vectors X^* is called *Pareto set*.

In other words, the decision vectors of the Pareto set cannot be improved w.r.t. any other objective function without worsening at least one of the other objectives. Based on Definition 1, the *Pareto front* is defined as follows:

Definition 2 (*Pareto front*)
Let X^* be a Pareto set. $F^* = f(X^*)$ is the set of all Pareto optimal objective vectors and is denoted as the *Pareto front*.

In practice, the generation of a set of decision vectors representing the entire Pareto front is often infeasible due to several reasons. For example, the number of Pareto optimal decision vectors may be too large, and even the determination of a single Pareto optimum may be $\mathcal{NP}$-hard [28]. Therefore, the goal is to find a *Pareto front approximation* that is as close to the Pareto optimal front as possible. The relationship among 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 5.

DOI: 10.1002/spe

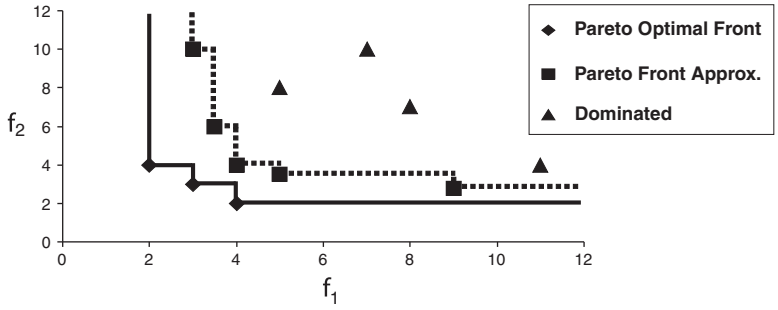

Figure 5. Pareto fronts.

In the compiler domain, Pareto front approximation can be finally used for two different purposes. On the one hand, it helps compiler writers to find suitable optimization levels. By constructing an approximation set for a large number of benchmarks, particular points from this set that satisfy the given trade-offs between the considered objectives can be chosen. The optimization sequences that represent these points will be implemented as optimization levels in the compiler and can be used in the future for new applications. This is also the scenario that we address in this paper. On the other hand, Pareto front approximations can be used by compiler users to tune the optimization sequence toward a single application. In contrast to the construction of optimization levels, the approximation set is computed for a particular application and the most suitable points are selected.

4.2. EMO algorithms

EAs stem from the domain of artificial intelligence and try to implement the principles of biological evolution. By employing reproduction mechanisms including mutation and recombination, offspring generations are created from which stronger individuals w.r.t. a certain fitness function are selected as parents for the next generation. With such an approach, preferably optimal solutions are 'cultured' instead of tackling an optimization problem analytically.

In the past, it was shown that randomized EMO algorithms are the 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 [8] studying the impact of multi-objective optimizations in the context of iterative compilation explored almost exclusively single EMO algorithms. Thus, it is not clear whether a selected algorithm is suitable or whether 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. These state-of-the-art optimization algorithms were chosen since each of them exhibits a different functionality. In the following, each algorithm will be briefly introduced and its specific features will be pointed out.

Indicator-based evolutionary algorithm

In contrast to other algorithms, the indicator-based evolutionary algorithm (*IBEA*) [29] determines fitness values by comparing individuals based on binary performance measures (called *indicators*) such as the additive ε-indicator. This technique has two advantages. First, the algorithm can be adapted to the user's preferences. Second, a preservation of the diversity of solutions is not required.

Non-dominated sorting genetic algorithm 2

The fitness assignment of the computationally fast non-dominated sorting genetic algorithm (*NSGA-II*) [30] involves a non-dominated sorting of individuals. Besides the low computational requirements, NSGA-II is a parameter-less approach. The only parameter defines the number of best

DOI: 10.1002/spe

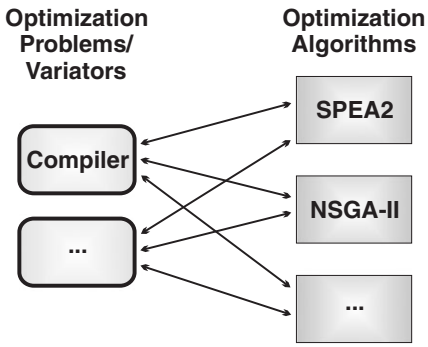

Figure 6. Combining optimization problems and algorithms.

solutions that a selection operator determines from a mating pool combining the parent and child population.

Strength Pareto evolutionary algorithm 2

The elitist strength Pareto evolutionary algorithm 2 (*SPEA2*) has three main characteristics. First, the fitness assignment for each individual i in the archive (set of Pareto solutions among all so far considered generations, used for creation of new generations) is based on the number of solutions that i dominates. Second, the density estimation utilizes a kth nearest neighbor method in order to discriminate between individuals with the same raw fitness value. Therefore, the inverse of the distance to the kth nearest neighbor (data point) serves as the density estimate. Finally, an enhanced archive truncation method ensures that extreme points are preserved in the solution space: if the number of non-dominated individuals does not fit into the archive, the individual with the kth nearest neighbor is selected for removal in each truncation iteration. This method only removes the individual who has the minimum distance to another individual in order to keep extreme solutions. For more details on fitness evaluation and archive truncation, the interested reader is referred to [31], Sections 3.1 and 3.2.

4.3. Statistical performance assessment

The typical problem with multi-objective optimizations is indicated in Figure 6. As shown in the left-hand side, numerous problem-specific modules exist. These modules serve as a representation of the problem as well as for the evaluation and variation of the solutions. An example is a compiler or a design space exploration for network processor architectures [32]. Owing to their purpose, these modules are often called *variators*. To approximate Pareto optimal solutions, each of these modules can be arbitrarily combined with any EMO algorithm.

While an algorithm expert is interested in the performance of his novel optimizer on real-life problems, application engineers are looking for an EMO optimizer that generates the best results for their specific problem. Typically, each user group represents experts of their own domain lacking an in-depth knowledge for the other field. Thus, it is good practice to separate the optimization problem from the algorithm and to allow arbitrary combinations of both parts for an independent performance evaluation.

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 is crossing Pareto fronts where a visual comparison is not intuitive anymore. To this end, an 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 that can be statically analyzed, i.e. a statistical hypothesis testing is

DOI: 10.1002/spe

conducted to indicate if the results are *significantly different* [28]. 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* [28]. 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.

Our performance assessment of the stochastic multi-objective optimizers is based on techniques proposed in [28] and comprises the following steps:

1. *Preprocessing*: The computed approximation sets of the three considered EMO algorithms are collected for runs with different seeds. Based on this collection C, the lower and upper bounds of the objective vectors are computed. The bounds are used to normalize all objective vectors, such that all values lie in the interval $[1, 2]$. Moreover, based on all Pareto solutions in C, a non-dominated front of objective vectors is determined, serving as *reference set* for the subsequent step. The reference set can be seen as an overall Pareto front approximation considered for all optimizer results.
2. *Dominance ranking*: For each normalized optimizer approximation set, the dominance ranking procedure is applied. The EMO algorithms are considered in ordered pairs and the *Mann–Whitney* rank sum test [33] for the determination of the statistical significance is performed.

For a thorough discussion of the performed statistical performance assessment, the interested reader is referred to [34].

5. 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 a total of 70 programs from the suites DSPstone [35], MediaBench [36], MiBench [37], MRTC [38], NetBench [39] and UTDSP [40]. Owing to these different benchmark suites, programs from various application domains were used for benchmarking: digital signal processing, scientific computing, audio, image and video en- and decoding and embedded control. The benchmarks' complexity ranges from rather simple filter and matrix computation kernels having two functions and only a dozen basic blocks up to entire GSM or ADPCM codecs with more than 50 functions and hundreds of basic blocks. This 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 7. As mentioned in Section 3, the WCET-aware C compiler WCC 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* within the ANSI-C source code. This data provides information about the code structure, such as the number of loop iterations or recursion depths, and is mandatory for a static WCET analysis. WCC supports the annotation

DOI: 10.1002/spe

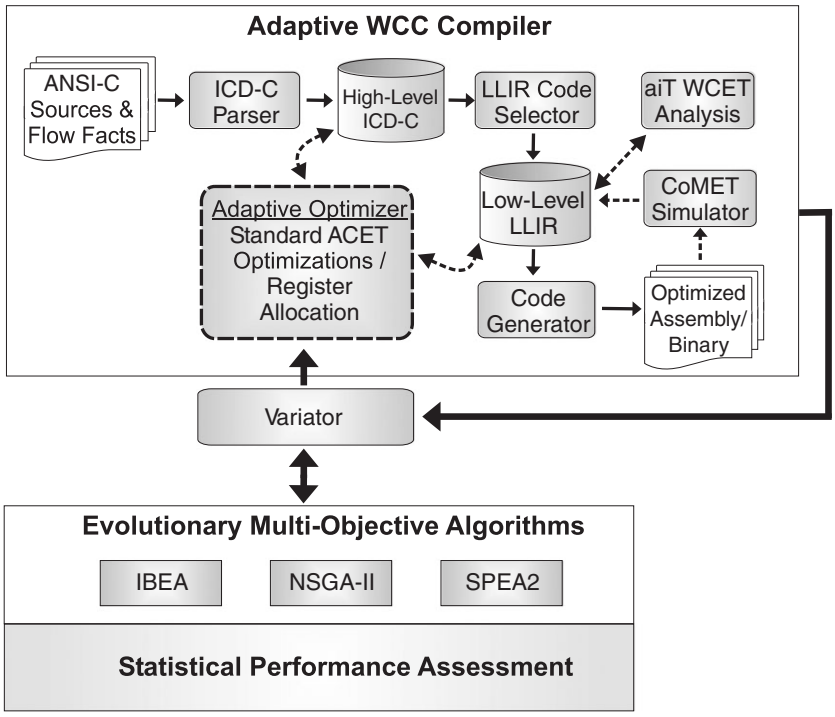

Figure 7. Workflow for the multi-objective exploration of compiler optimizations.

of the so-called *loop bound* and *flow restriction* flow facts. Using loop bounds, the developer can specify minimal and maximal iteration counts for regular for-, while-do- and do-while-loops. Irregular loops, recursions or other control flow structures can be described using flow restrictions that allow to relate the execution frequency of one C statement with that of other statements based on linear inequalities. WCC uses *flow fact translation* in order to close the semantic gap between the place where flow facts are specified (C code) and where they are actually used for static WCET analysis (assembly code). *Flow fact update* techniques guarantee that the flow facts passed to the WCET analyzer are semantically equivalent to the specifications provided at the source code level. In particular, WCC's loop optimizations restructure loops heavily so that iteration counts might change as a result. These changes of iteration counts are automatically considered during flow fact update so that always valid and consistent flow facts are used for the WCET analysis. For more details, the interested reader is referred to [17].

The communication between WCC via a variator and the different EMO algorithms (cf. Figure 7) is fully automated. The process starts with the variator that generates random compiler optimization sequences, representing the initial population. The optimizations are encoded as strings. The sequences are passed to the adaptive WCC framework, which is extended by an adaptive optimizer that can perform code transformations in an arbitrary order (cf. Section 3.2). WCC uses these sequences to generate code that is processed by the WCET analyzer aiT and the TriCore simulator *CoMET*. The estimated WCET, ACET and code size (obtained from LLIR) are returned to the variator. In this way, the variator manages for each optimization sequence the corresponding objective values. For distinction, each sequence is indexed using a unique ID.

In the next step, for each evaluated optimization sequence of a particular generation, the variator passes the corresponding indices and objective vectors to the EMO algorithms. Exclusively based on the objective vectors, the EMO algorithms compute the approximated Pareto front and return the respective indices of the Pareto solutions back to the variator. The returned solutions are finally used by the mutation and crossover operators to generate individuals for the next generation of optimization sequences. Moreover, the computed Pareto front approximations serve as input to the statistical performance assessment.

DOI: 10.1002/spe

Both, the collection of the EMO algorithms and the performance assessment, are part of the *Platform and Programming Language Independent Interface for Search Algorithms* (*PISA*) framework [41]. An existing PISA variator was re-implemented in order to be applicable with the WCC framework.

For the conducted experiments, the genetic algorithms used to explore the compiler optimization space were configured as follows, according to the settings already described in the related literature [8, 18]:

- the algorithms IBEA, NSGA-II and SPEA2 were run 5 times with different random seeds;
- for each run, each population comprises 50 individuals (optimization sequences);
- the archive of Pareto solutions among all generations considered during the evolutionary algorithms [41] holds 25 individuals;
- a one-character mutation (cf. Section 3.3) with a probability of 10%;
- a one-point crossover probability of 90%;
- optimization was run for 50 generations;
- statistical performance assessment with a significance level α=5%.

In the following, aiT was configured such that WCET analyses were performed with the highest precision. This particularly means that context-sensitive WCET analysis was enabled for all benchmarks by setting aiT's virtual unrolling factor and call string length to infinite. Only for a few extremely complex benchmarks, these settings lead to situations where WCET analysis using aiT does not terminate in reasonable time. For these few cases, context sensitivity of aiT was reduced by manually setting the above parameters to small values.

For all experiments, the Infineon TriCore TC1796 processor heavily used in the automotive domain was considered. It is a 32-bit RISC controller with the following features:

- DSP-like instruction set with extensions to support SIMD and bit-packet processing;
- single-precision floating-point unit;
- 16 32-bit address registers, out of which 10 can be used freely by the compiler; 16 32-bit general-purpose data registers;
- three pipelines for parallel ALU operations, memory transfers and zero-overhead hardware loops;
- Harvard architecture with separate busses and memories for instructions and data;
- 48 kB L1 instruction scratchpad memory, 16 kB L1 I-cache (2-way set associative, least-recently used replacement, 32 B line size), 2 MB L2 instruction flash memory;
- 56 kB L1 data scratchpad memory, 64 kB L2 data memory, no data caches.

6. RESULTS

The multi-objective optimizations are carried out for pairs of objective functions. The consideration of 2-dimensional Pareto fronts is motivated by two issues:

- Since the impact of standard optimizations is unknown so far for the trade-off between the WCET and other objectives, this work is the first case study to investigate this issue. The results help to understand the basic interferences between different objectives. Starting with the investigation of more than two objectives may hide some objective interferences, leading to a lack of the fundamental understanding.
- Compiler writers typically consider a trade-off between two objective functions; thus, the results of this work are more valuable for them than the presentation of complex, often not intuitive, objective interferences.

6.1. Statistical evaluation

Table II presents the 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

DOI: 10.1002/spe

Table II. 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	—

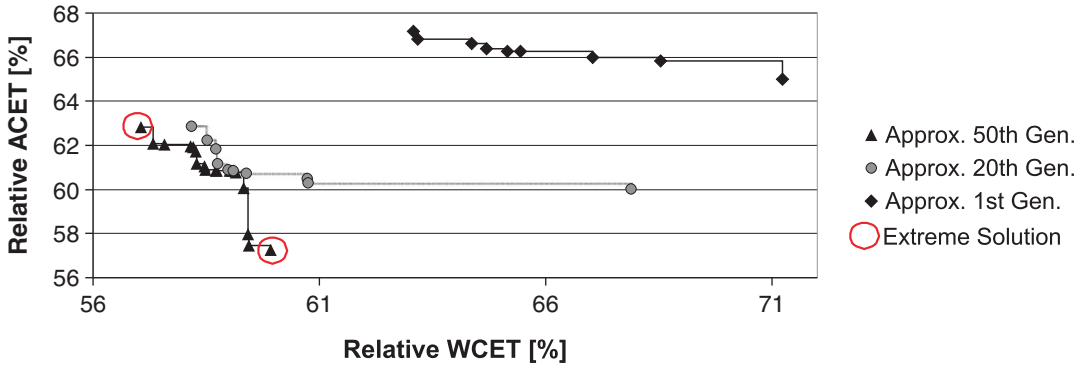

Figure 8. NSGA-II Pareto front approximation for ⟨WCET, ACET⟩.

indicate results for the objective functions ⟨WCET, ACET⟩, whereas 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 observed. 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 conform with the results reported in [42] where the authors observed that NSGA-II is able to outperform SPEA2 for problems with two objectives. This behavior does not conform to the results determined in various other publications [31, 43–45], where SPEA2 outperforms NSGA-II for different number of objectives. Although the evaluation of different selection algorithms in this paper does not provide significant evidence that NSGA-II always performs the best for all problems, we have shown that different EMO algorithms should be evaluated for solving new problem constellations.

Provided that Pareto front approximation has been done once using the above EMO algorithms, dominance ranking is computationally cheap since it is based on simple sorting of the approximation sets. Therefore, the runtime of dominance ranking by itself is negligible.

6.2. *Analysis of Pareto front approximations*

Figure 8 visualizes the Pareto front approximation generated by the algorithm NSGA-II, which achieved the best performance assessment results for the objective functions WCET and ACET.

DOI: 10.1002/spe

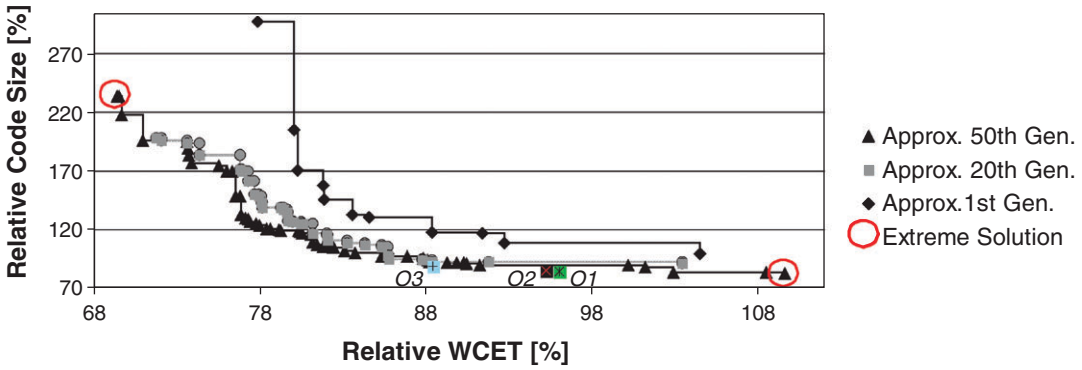

Figure 9. NSGA-II Pareto front approximation for ⟨WCET, code size⟩.

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 by up to 42.9 and 42.8%, respectively.
- The discovered sequences significantly outperform the standard optimization levels, having the $(WCET, ACET)$-coordinates $(96.0, 89.1)$ for *O1*, $(95.2, 90.4)$ for *O2* and $(88.4, 84.7)$ for *O3* (in order to enable better readability of the approximated Pareto front, the figure has been scaled and these points are not included in the figure).
  For example, the performance of *O3* is outperformed by 31.3% and 27.5% for WCET and ACET, respectively, when the *extreme* situations (circled in Figure 8) are selected. If a trade-off between WCET and ACET is chosen $(58.5, 60.8)$, *O3* can still be outperformed by 29.9% for WCET and 23.9% for ACET.
  The lower optimization levels *O2* and *O1* are outperformed by 38.1 and 38.9% for WCET as well as 27.4% and 33.1% for ACET.
- Standard compiler optimizations have a similar impact on the WCET and ACET. This observation provides an important answer to the question that concerns all designers of real-time systems: *which impact can be expected from standard ACET optimizations on the system's worst-case behavior*? This case study shows that similar effects on the average-case and worst-case behavior are likely.

The Pareto front approximation computed by NSGA-II for the objectives WCET and code size is depicted in Figure 9. The relative WCET estimation w.r.t. the non-optimized code (corresponds to 100%) is represented by the horizontal axis, whereas 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 and are constructed of Pareto solutions found in the 5 runs of the algorithm. The 50th-generation front comprises 53 points. Compared with 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% can be achieved (left-most solutions of the corresponding front). 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 (marked by circles): a WCET reduction of 30.6% with a simultaneous code size increase of 133.4%, or a WCET degradation of 9.6% with a simultaneous code size reduction of 16.9%. Hence, the results for later generations yield substantially better results.

DOI: 10.1002/spe

- The Pareto solutions outperform WCC's standard optimization levels, which are depicted in Figure 9. The standard optimization levels perform well for the code size reduction compared with the Pareto solutions. 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% as found by NSGA-II.

However, there are also two major differences compared with 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. Also, compiler writers developing WCET-aware optimizations must be aware of these conflicting objectives and should always consider the impact of their optimizations on the code size. The second difference is that standard compiler optimizations available in WCC are not capable of accomplishing a notable code size decrease (up to 16.9% which is approximately half of the achievable WCET reduction). Therefore, tailored optimizations are required if code size reduction is the primary goal.

6.3. Analysis of the optimization sequences

A closer look at the Pareto optimal optimization sequences for the objective pair ⟨WCET, ACET⟩ reveals the following observations:

- Most of the optimization sequences contain an aggressive loop unrolling or function inlining in the very beginning. Aggressive means that loops/callees with a maximal size of 200 expressions (maximal parameter value considered during exploration) were transformed.
- In addition to these two optimizations, the found Pareto sequences often contain the optimization procedure cloning. Since cloning, unrolling and inlining are all contained in WCC's optimization level *O3*, it can be concluded that this standard optimization level holds promising optimizations for maximal WCET/ACET reduction.
- Other optimizations frequently found in the Pareto solutions are: instruction scheduling applied at physical LLIR, ILP-based RA (hence, the optimization's complexity pays off) and loop-invariant code motion.
- Optimizations that were infrequently contained in the sequences—hence can be considered less beneficial–are: instruction scheduling applied at virtual LLIR, loop collapsing, life range splitting and loop deindexing.
- There is no clear separation that optimizations are the best suitable for a particular objective; hence in general, many optimizations have a comparable effect on the estimated WCET and ACET.

The analysis of the Pareto optimal optimization sequences for the objective pair ⟨WCET, code size⟩ leads to the following observations:

- Function inlining can often be found in code size-oriented sequences. Although widely believed that the optimization always yields a code expansion, inlining can also reduce code size if functions, which are invoked once in the code, are inlined and further optimized.
- Especially for the code size-oriented solutions, many sequences begin with procedure cloning of small functions (limited to 20 expressions). A possible explanation is that cloning of small function has a negligible impact on the code size but may significantly improve the estimated WCET; thus, overall good Pareto solutions can be generated that way. Moreover, some of the clones can be inlined afterwards, possibly leading to a code size decrease.
- In contrast to the WCET-oriented sequences, none of the code size-oriented optimization strategies contained loop unrolling. Hence, it can be expected that unrolling is likely to yield a code size increase.

DOI: 10.1002/spe

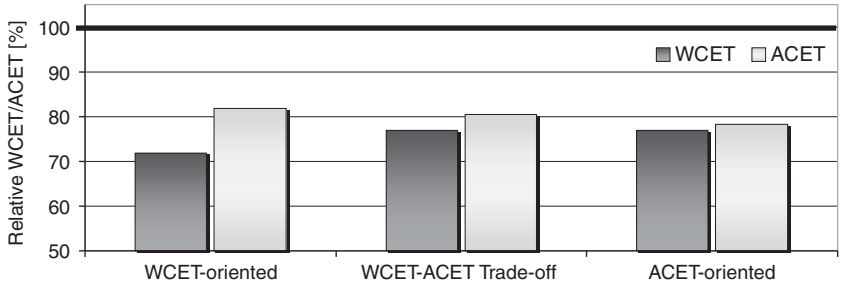

Figure 10. Objectives ⟨WCET, ACET⟩: comparison of three Pareto optimal optimization sequences with optimization level *O3*.

- Loop unswitching is another optimization that is rarely found in the code size-oriented sequences. This is due to the fact that unswitching also increases code size and thus should be used with caution if code size is critical.
- As shown above, the objectives WCET and code size have conflicting goals. Therefore, some standard optimizations, such as peephole optimizations or dead-code elimination, have a positive effect on both objectives, while other optimizations should be applied only if improvements of one objective at the cost of the other can be tolerated.

6.4. Cross-validation

To estimate the generalization ability of the discovered sequences, a cross-validation was performed, i.e. optimization sequences found by NSGA-II in the 50th generation for the training set are applied to unseen benchmarks from the test set.

Among the large number of solutions constructing a Pareto front approximation, three solutions are discussed in more detail since they provide typical optimization scenarios. If the system designer is interested in a maximal reduction of a particular objective function, an optimization sequence represented by one of the extreme points from the fronts should be considered. Another alternative is to choose a Pareto solution from the middle of the front that represents a trade-off between the respective objectives.

For the objective pair ⟨WCET, ACET⟩, the optimization sequences defined by the extreme points in Figure 8 as well as the trade-off, which is represented by the solution with the coordinates $(58.7, 60.9)$, were evaluated. These three optimization sequences were applied to each of the 35 benchmarks from the test set. For each benchmark, the results for the estimated WCET and ACET using the new sequence were compared with the WCET/ACET results when the benchmark was compiled with WCC's highest optimization level *O3*. The averaged results for all benchmarks, with 100% being the base line representing results achieved with *O3*, are shown in Figure 10. The following results can be observed:

- Using the *WCET-oriented* optimization sequence, which is represented in Figure 8 by the left-most Pareto solution with the *WCET*, *ACET* coordinate $(57.1, 62.8)$, outperforms WCC's default optimization level *O3* by 28.0 and 18.0% for the estimated WCET and ACET reduction, respectively. It can also be seen that the estimated WCET was improved at the cost of ACET.
- The optimization sequence (labeled with *WCET-ACET Trade-off*) was determined by NSGA-II as a compromise between the WCET estimation and ACET for the training set (see solution with coordinate $(58.7, 60.9)$ in Figure 8). For the test set, the results are slightly worse than for the training set, since a relative WCET estimation of 76.2% and a relative ACET of 80.5% were observed. However, the optimization level *O3* can still be significantly outperformed by 23.8 and 19.5% for WCET and ACET, respectively. Hence, the discovered optimization sequence is a promising candidate for the substitution of WCC's optimization level *O3*.

DOI: 10.1002/spe

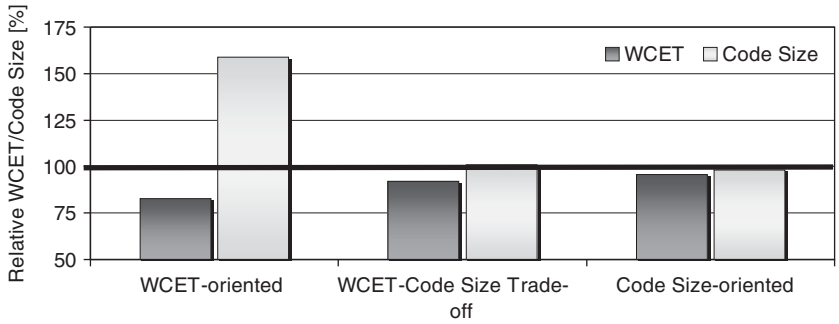

Figure 11. Objectives ⟨WCET, code size⟩: comparison of three Pareto optimal optimization sequences with optimization level *O3*.

- The optimization sequence (coordinate (59.9, 57.2) in Figure 8) aiming at the maximal ACET reduction (labeled with *ACET-oriented*) could achieve the maximal ACET reduction of 23.4% among the considered optimization strategies. This sequence has even a slightly better impact on the average WCET than on the average ACET. Hence, again *O3* can be improved.

Analogously, Figure 11 reflects the performance of the sequences found for ⟨WCET, code size⟩ in Figure 9. Using the *WCET-oriented* optimization scenario, WCET reduction of 17.4% can be achieved on average for the test set. However, this improvement comes at the cost of a code size increase of 58.7%. The optimization sequence labeled with *WCET–code size Trade-off* not only improves the WCET by 7.8% w.r.t. *O3* but also results in a slight code size increase of 1.0%. Concerning the *Code Size-oriented* strategy, the WCET and code size can be reduced by 4.25 and 1.9%, respectively, compared with *O3*.

6.5. Optimization runtime

The runtime of the multi-objective exploration of optimization sequences was measured for each EMO algorithm. Five optimization runs with different seeds, each computing 50 generations, took about 6 days on a Intel Quad-Core Xeon 2.4 GHz machine with 8 GB RAM. This optimization runtime might seem long. However, it should be noted that these automatic tests have to be performed once off-line, while the results (optimization sequences) can be reused 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.

The application of the new optimization sequences found by the EMO algorithm NSGA-II does not considerably increase the compilation time compared with WCC's *O3*. In some cases, like for an extensive loop unrolling or function inlining, slight increases of the compilation time (typically few seconds) could be observed.

7. CONCLUSIONS

The search for good compiler optimization sequences is challenging since optimizations exhibit complex interactions that 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 among the WCET, ACET and code size. To find the best EMO 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 28.0 and 23.4%, 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.

DOI: 10.1002/spe

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.

We plan to categorize classes of programs by evaluating a set of features in order to distinguish multimedia, network or DSP algorithms in the future. For such classes of algorithms, special optimization sequences should be derived. If classes of algorithms are automatically determined, the WCC compiler should choose the most promising set of optimizations in order to achieve the best possible performance of the generated code.

ACKNOWLEDGEMENTS

The authors thank AbsInt Angewandte Informatik GmbH for their support concerning WCET analysis using the aiT framework. The authors also thank Synopsys for the provision of the instruction set simulator CoMET enabling the determination of the average-case execution times.

The research leading to these results has been partially funded by the European Community's Artist-Design Network of Excellence and by the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement no. 216008.

REFERENCES

1. Kulkarni PA, Hines SR, Whalley DB, Hiser JD, Davidson JW, Jones DL. Fast and efficient searches for effective optimization-phase sequences. *Transactions on Architecture and Code Optimization* 2005; **2**(2):165–198.
2. Leather H, O'Boyle M, Worton B. Raced profiles: Efficient selection of competing compiler optimizations. *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (*LCTES*), Dublin, Ireland, 2009; 50–59.
3. Agakov F, Bonilla E, Cavazos J, Franke B, Fursin G, O'Boyle MFP, Thomson J, Toussaint M, Williams CKI. Using machine learning to focus iterative optimization. *Proceedings of the Fourth Annual IEEE/ACM International Symposium on Code Generation and Optimization* (*CGO*), New York, U.S.A., 2006; 295–305.
4. Cooper KD, Schielke PJ, Subramanian D. Optimizing for reduced code space using genetic algorithms. *ACM SIGPLAN Notes* 1999; **34**(7):1–9.
5. Lokuciejewski P, Plazar S, Falk H, Marwedel P, Thiele L. Multi-objective exploration of compiler optimizations for real-time systems. *Proceedings of the 13th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing* (*ISORC*), Carmona, Spain, 2010; 115–122.
6. Almagor L, Cooper KD, Grosul A, Harvey TJ, Reeves SW, Subramanian D, Torczon L, Waterman T. Finding effective compilation sequences. *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (*LCTES*), Washington, U.S.A., 2004; 231–239.
7. Zhao M, Childers B, Soffa ML. Predicting the impact of optimizations for embedded systems. *Proceedings of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems* (*LCTES*), San Diego, U.S.A., 2003; 1–11.
8. Hoste K, Eeckhout L. COLE: Compiler optimization level exploration. *Proceedings of the Sixth Annual IEEE/ACM International Symposium on Code Generation and Optimization* (*CGO*), Boston, U.S.A., 2008; 165–174.
9. Campoy AM, Puaut I, Ivars AP, Mataix JVB. Cache contents selection for statically-locked instruction caches: An algorithm comparison. *Proceedings of ECRTS*, Palma de Mallorca, Spain, 2005; 49–56.
10. Vera X, Lisper B, Xue J. Data cache locking for higher program predictability. *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (*SIGMETRICS*), San Diego, U.S.A., 2003; 272–282.
11. Plazar S, Lokuciejewski P, Marwedel P. WCET-aware software based cache partitioning for multi-task real-time systems. *Proceedings of the Ninth International Workshop on Worst-case Execution Time Analysis* (*WCET*), Dublin, Ireland, 2009; 78–88.
12. Falk H. WCET-aware register allocation based on graph coloring. *Proceedings of the 46th Design Automation Conference* (*DAC*), San Francisco, U.S.A., 2009; 726–731.
13. Deverge JF, Puaut I. WCET-directed dynamic scratchpad memory allocation of data. *Proceedings of the 19th Euromicro Conference on Real-time Systems* (*ECRTS*), Pisa, Italy, 2007; 179–190.
14. Suhendra V, Mitra T, Roychoudhury A, Chen T. WCET centric data allocation to scratchpad memory. *Proceedings of the 26th IEEE International Real-time Systems Symposium* (*RTSS*), Miami, U.S.A., 2005; 223–232.
15. Falk H, Kleinsorge JC. Optimal static WCET-aware scratchpad allocation of program code. *Proceedings of the 46th Design Automation Conference* (*DAC*), San Francisco, U.S.A., 2009; 732–737.
16. Kadlec A, Kirner R, Puschner P. Avoiding timing anomalies using code transformations. *Proceedings of the 13th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing* (*ISORC*), Carmona, Spain, 2010; 123–132.

DOI: 10.1002/spe

17. Falk H, Lokuciejewski P. A compiler framework for the reduction of worst-case execution times. *The International Journal of Time-critical Computing Systems* (*Real-time Systems*) 2010; **46**(2):251–300.
18. Prasad WZ, Kulkarni P, Whalley D, Healy C, Mueller F, Uh G-R. Tuning the WCET of embedded applications. *Proceedings of the 10th IEEE Real-time and Embedded Technology and Applications Symposium* (*RTAS*), Toronto, Canada, 2004; 472–481.
19. Falk H, Lokuciejewski P, Theiling H. Design of a WCET-aware C compiler. *Proceedings of the 2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia* (*ESTIMedia*), Seoul, Korea, 2006; 121–126.
20. GCC, GNU Compiler Collection. Available at: http://gcc.gnu.org/ [10 June 2010].
21. AbsInt Angewandte Informatik GmbH. Worst-case execution time analyzer aiT for triCore. Available at: http://www.absint.com/ait [10 June 2010].
22. Muchnick SS. *Advanced Compiler Design and Implementation*. Morgan Kaufmann: San Francisco, U.S.A., 1997.
23. Briggs P. Register allocation via graph coloring. *PhD Thesis*, Rice University, Houston, U.S.A., 1992; 1–104.
24. Goodwin DW, Wilken KD. Optimal and near-optimal global register allocation using 0–1 integer programming. *Software—Practice and Experience* 1996; **26**(8):929–965.
25. Guo Y, Subramanian D, Cooper KD. An effective local search algorithm for an adaptive compiler. *Proceedings of the First Workshop on Statistical and Machine Learning Approaches Applied to Architectures and Compilation* (*SMART*), Ghent, Belgium, 2007; 7–11.
26. Synopsys, Inc. Available at: http://www.synopsys.com [10 June 2010].
27. Laumanns M, Thiele L, Zitzler E, Welzl E, Deb K. Running time analysis of multi-objective evolutionary algorithms on a simple discrete optimization problem. *Proceedings of the Seventh International Conference on Parallel Problem Solving from Nature* (*PPSN*), Granada, Spain, 2002; 44–53.
28. Knowles J, Thiele L, Zitzler E. A tutorial on the performance assessment of stochastic multiobjective optimizers. *Proceedings of Third International Conference on Evolutionary Multi-criterion Optimization* (*EMO*), Guanajuato, Mexico, 2005; 1–35.
29. Zitzler E, Künzli S. Indicator-based selection in multiobjective search. *Proceedings of the Ninth International Conference on Parallel Problem Solving from Nature* (*PPSN*), Birmingham, U.K., 2004; 832–842.
30. Deb K, Agrawal S, Pratap A, Meyarivan T. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimisation: NSGA-II. *Proceedings of the Sixth International Conference on Parallel Problem Solving from Nature* (*PPSN*), Paris, France, 2000; 849–858.
31. Zitzler E, Laumanns M, Thiele L. SPEA2: Improving the strength pareto evolutionary algorithm for multiobjective optimization. *Proceedings of the Conference on Evolutionary Methods for Design, Optimisation and Control with Application to Industrial Problems* (*EUROGEN*), Athens, Greece, 2001; 95–100.
32. Thiele L, Chakraborty S, Gries M, Künzli S. A framework for evaluating design tradeoffs in packet processing architectures. *Proceedings of the 39th Design Automation Conference* (*DAC*), New Orleans, U.S.A., 2002; 880–885.
33. Conover WJ. *Practical Nonparametric Statistics*. Wiley: New York, U.S.A., 1971.
34. Lokuciejewski P, Marwedel P. *Worst-case Execution Time Aware Compilation Techniques for Real-time Systems*. Springer: Dordrecht, Netherlands, 2010.
35. Zivojnović V, Martinez J, Schläger C, Meyr H. DSPstone: A DSP-oriented benchmarking methodology. *Proceedings of the International Conference on Signal Processing and Technology* (*ICSPAT*), Dallas, U.S.A., 1994; 715–720.
36. Lee C, Potkonjak M, Mangione-Smith WH. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. *Proceedings of the 30th Annual International Symposium on Microarchitecture* (*MICRO*), Washington, DC, U.S.A., 1997; 330–335.
37. Guthaus MR, Ringenberg JS, Ernst D, Austin TM, Mudge T, Brown RB. MiBench: A free, commercially representative embedded benchmark suite. *Proceedings of the Fourth IEEE International Workshop on Workload Characteristics* (*WWC*), Austin, U.S.A., 2001; 3–14.
38. Mälardalen WCET Research Group. WCET benchmarks. Available at: http://www.mrtc.mdh.se/projects/wcet [10 June 2010].
39. Memik G, Mangione-Smith WH, Hu W. NetBench: A benchmarking suite for network processors. *Proceedings of the International Conference on Computer-aided Design* (*ICCAD*), San Jose, U.S.A., 2001; 39–42.
40. TDSP Benchmark Suite. Available at: http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html [10 June 2010].
41. Bleuler S, Laumanns M, Thiele L, Zitzler E. PISA—A platform and programming language independent interface for search algorithms. *Proceedings of Second International Conference on Evolutionary Multi-criterion Optimization* (*EMO*), Faro, Portugal, 2003; 494–508.
42. Künzli S, Bleuler S, Thiele L, Zitzler E. A computer engineering benchmark application for multiobjective optimizers. *Application of Multi-objective Evolutionary Algorithms*. World Scientific: Singapore, 2004; 269–294.
43. Maneeratana K, Boonlong K, Chaiyaratana N. Compressed-objective genetic algorithm. *Parallel Problem Solving from Nature—PPSN IX* (*Lecture Notes in Computer Science*). Springer: Berlin, 2006; 473–482.
44. Künzli S. Efficient design space exploration for embedded systems. *PhD Thesis*, Swiss Federal Institute of Technology Zurich, Switzerland, 2006.
45. Khare VR, Yao X, Deb K. Performance scaling of multi-objective evolutionary algorithms. *EMO*, Faro, Portugal, 2003; 376–390.

DOI: 10.1002/spe