Let's Get Less Optimistic in Measurement-Based Timing Analysis

Sven Bünte, Michael Zolda
Vienna University of Technology
{sven,michaelz}@vmars.tuwien.ac.at

Raimund Kirner
University of Hertfordshire
r.kirner@herts.ac.uk

Abstract—**Measurement-based timing analysis (MBTA) is a hybrid approach that combines execution time measurements with static program analysis techniques to obtain an estimate of the worst-case execution time (WCET) of a program. In order to minimize the chance that the WCET estimate is below the real WCET, the set of representative execution-time measurements has to be selected advisedly. We present an input data generation technique that uses a combination of model checking and genetic algorithms in order to heuristically optimize the set of measurements in terms of safety.**

Index Terms—**Real-time systems, validation, worst-case execution time, measurement-based timing analysis**

I. Introduction

A real-time computer system is a computer system in which correctness does not only encompass functional behavior, but also compliance to temporal constraints. If the violation of timing constraints can have catastrophic consequences we speak of a *hard real-time system*. On the other hand, a *soft real-time system* can tolerate violations of temporal constraints to some extent. An example of the latter is a mobile phone application involving minor communication delays. As an example for hard real-time systems, an airbag not releasing in time or a non-reacting aircraft control unit can lead to catastrophic consequences. Consequently, there is an inherent interest in verification and validation techniques that focus on the temporal behavior of real-time systems.

Many real-time computer systems are implemented as a collection of individual tasks in order to handle complexity. A valid schedule for those tasks ensures the adherence to temporal dependencies [1]. Most of the common scheduling algorithms rely on the *Worst-Case Execution Time* (WCET) of each single task.

Determining the WCET of a program by simple end-to-end measurements exercising the program with different input data is unlikely to find a safe upper bound on the real WCET, i.e., a bound that is never exceeded under any circumstances. As a more systematic approach that provides much higher confidence into safety of the obtained WCET bound, static WCET analysis is based on the provision of an accurate timing model of the processor [2]. Given that the involved analysis techniques are sound and that the timing model is correct, static WCET analysis can yield a safe upper bound on the WCET. However, timing properties of modern architecture features like caches, branch predictors or out-of-order execution are hard to model and analyze precisely. As a consequence, static analysis struggles to deliver *precise* results, i.e., WCET bounds that are only slightly larger than the real WCET, due to conservative assumptions it has to act on in order to be safe. Further, the manual construction of a detailed timing model is often only economically feasible for processors used in safety-critical systems where comprehensive verification efforts are mandatory for certification.

Thus, measurement-based timing analysis has emerged [3], where the timing model is obtained from execution time measurements, making sophisticated hardware models unnecessary. However, without those detailed models, it is not possible to derive safe WCET estimates in general for non-trivial hardware architectures. Thus, it is crucial for MBTA to select the input vectors for measurements advisedly.

The problem we address in this work is the following:

How can we generate test vectors for the execution time measurement phase of MBTA such that we minimize the chance of WCET underestimation?

To answer this question, we will first shine a light on the aspect of *optimism* in the context of measurement-based timing analysis in Section IV. This will provide us means to empirically compare test suites used for measurement in MBTA in terms of safety.

We will then introduce FROO: The *FORTAS Reduction Of Optimism* input data generation technique which addresses the objective of reducing optimism in MBTA, thereby decreasing the chance of WCET underestimation (see Section V). The heuristic approach follows the principle of a genetic algorithm where model checking is used for guiding the search for promising test vectors by providing high-quality seeds. In contrast to existing methods that use genetic algorithms for WCET estimation of end-to-end executions, we try to jointly maximize **local** WCET estimates of program parts. This is reflected by a fitness function that is specifically tailored to our application.

With a focus on the automotive domain, we experimentally show the effectiveness of FROO for the Infineon TriCore 1796 processor and a domain-specific benchmark in Section VI. Before we address optimism and FROO we will give a detailed introduction to measurement-based timing analysis in Section II. Then, Section III will introduce basic concepts needed for our investigations. Readers familiar with genetic algorithms might directly skip to Section III-B which provides a small set of notions used throughout the paper.

Related work is provided by Section VII after which this article concludes and motivates future work in Section VIII.

II. Measurement-Based Timing Analysis

Measurement-based timing analysis (MBTA) is a hybrid WCET analysis technique. It combines static program analysis techniques and execution time measurements.

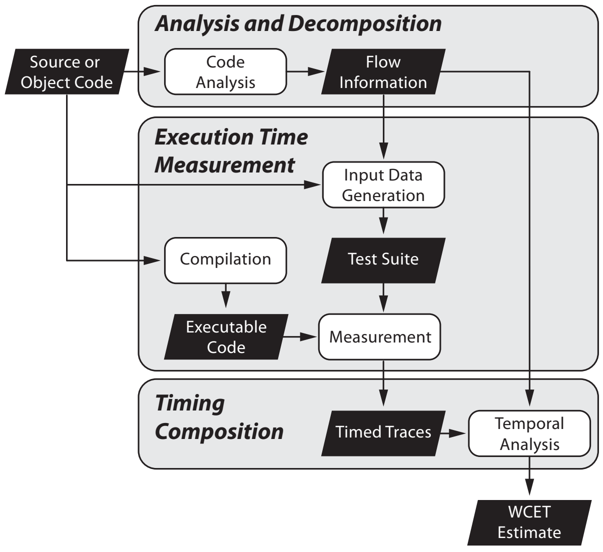


Fig. 1. Measurement-based timing analysis (MBTA)

As shown in Figure 1, MBTA first analyzes and decomposes a program into *segments* following the divide and conquer principle. WCET estimates for program segments are then derived in the phase of *execution time measurement*. Finally, all local timing information is composed to yield a global WCET estimate for the whole program in the phase of *timing composition*.

- **Analysis and Decomposition:** For WCET analysis, the maximal end-to-end execution time of the software is of interest. In general, to obtain a perfectly accurate timing model, we would have to consider the execution time of all possible operation sequences that can be performed by the computer for all possible initial states of the system under scrutiny while executing the given computer program. Measuring all these sequences is intractable in general, as there are simply too many. Therefore, reducing the number of execution time measurements is crucial. Usually, MBTA approaches try to access the local WCET of program segments (subgraphs [4]–[6], sequences between instruction points [7], down to basic blocks [4]).
- **Execution Time Measurement:** Once the program is decomposed, the execution time is estimated for each segment. Execution times are measured on real hardware, which allows to take hardware characteristics into account without modeling them in full detail. This phase introduces *optimism*: the maximal observed execution time for each program segment is gathered from measurements and optimistically assumed to be the real WCET.
- **Timing Composition:** The timing results from all segments are composed via IPET [8], [9] or a tree-based approach [7], [10] to obtain a global WCET estimate. This phase introduces *pessimism*, i.e., whenever the composition technique lacks context information for deriving WCETs, it conservatively chooses those scenarios leading to higher WCET estimates.

The major drawback of MBTA is that, in general, it cannot provide sufficient coverage of all system states to guarantee that the maximal observed execution time (MOET) for a segment is indeed the WCET. This is due to hardware features like pipelines, caches, and out-of-order execution that blow up the state space of the processor. Further, all possible system states at the entry of a segment would have to be covered in order to guarantee that the WCET is among the MOETs. Since we cannot guarantee safety of WCET estimates for program segments, we also cannot guarantee to produce safe WCET estimates of the overall program as there is no means to compensate unsafe WCET estimates for program segments in the timing composition phase of MBTA in general. However, in contrast to static WCET analysis, MBTA does not try to guarantee safety for all of its results. It rather aims for balancing analysis efforts to serve both needs: to aim for adequately safe results (i.e., reduce optimism) on the one hand and to be precise on the other hand (i.e., reduce pessimism).

With respect to the verification of temporal requirements in embedded systems, this implies that MBTA targets soft real-time systems primarily, where precision of a calculated WCET bound is as crucial as safety. However, MBTA can also be used to verify hardware models used in static WCET analysis or for design space exploration of both hard and soft real-time systems.

A discussion on how to extend MBTA such that pessimism can be handled is presented in [11]. In contrast, this article, targets optimism as a diametrically opposed aspect. We will therefore focus only on the phase of execution time measurement, more specifically, on its input data generation.

III. Preliminaries

A. Genetic Algorithms

Evolutionary algorithms aim for solving optimization problems stochastically. They follow the idea of Darwin's *survival of the fittest* theory [12]. In evolutionary algorithms, potential solutions to an optimization problem are regarded as *individuals*. A *fitness function* quantifies the survival probability of an individual in such a way that the fitter an individual is, the better the corresponding solution is with respect to the optimization problem. Evolutionary algorithms iteratively generate *populations*, i.e., sets of individuals, by means of reproduction features using stochastic operators such as *selection*, *recombination* and *mutation*.

The term *evolutionary algorithm* encompasses several subtopics or techniques. One of these derived techniques is *genetic algorithm* [13] where individuals are represented as strings of numbers. Usually, those numbers are binary-coded but this is not mandatory.

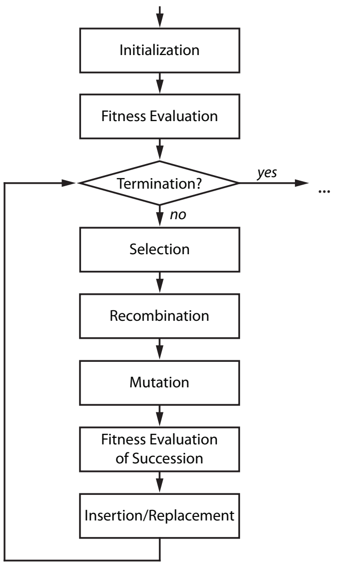


Fig. 2. General structure of a genetic algorithm [14]

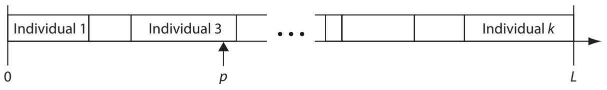
The general structure of a genetic algorithm is depicted in Figure 2. One characteristic feature of genetic algorithms is that they always incorporate a *selection* phase, which is not necessarily part of an evolutionary algorithm in general. In this phase, only those individuals showing an appropriate fitness level are selected for reproduction in contrast to treating all individuals equally. Further, recombination always plays an essential role in genetic algorithms, which again is not mandatory for evolutionary algorithms in general. In the phase of recombination, a new individual is produced out of two or more selected parent individuals. The subsequent mutation phase in which individuals are randomly altered is optional. Finally, the *offspring* is evaluated with respect to its fitness to determine, in the phase of *insertion/replacement*, which newly generated individuals will be integrated into the current population and which ones will be removed.

We will now separately introduce the particular phases at a level that is sufficient to understand the techniques and principles to be presented in this paper. For a detailed introduction to evolutionary and genetic algorithms, the interested reader might refer to Wegener's PhD thesis [14] from which most of the introduction here evolved.

1) Initialization: There is no principle that generally describes the initialization phase. A straightforward and popular approach is to randomly generate an initial population, also referred to as *seed*, having the same size as the populations in the following iterations. However, neither the size of the seed is fixed nor is the generation technique in general.

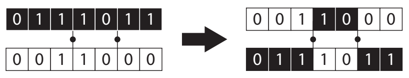
2) Fitness Evaluation: Determining the fitness of an individual involves two steps. First, a *target function* assigns a value to each individual. This value exclusively depends on the individual's variables and represents the individual's quality. Second, based on the target function values of all individuals, the fitness (survival probability) is computed for each individual. Whereas the character of the target function strongly depends on the optimization problem, the assignment of survival probabilities usually follows known principles. We will use a *proportional fitness assignment* [15], where the fitness is proportional to the target function value.

3) Selection: Those individuals suggesting to be promising candidates for reproduction are identified in the selection phase. Many techniques exist for selection, a detailed evaluation of which is given in [16]. For our experiments we choose *Roulette Wheel Selection* [17]. Here, each individual gets initially assigned to a single section of a line with length L. The extent of each section is proportional to the fitness of its assigned individual. In a next step, a random value p is picked (uniformly distributed) where p is in the interval $[0, L]$.



In roulette wheel selection, the unique section that includes p indirectly selects the corresponding individual. The process is repeated until the required amount of individuals for reproduction is selected.

4) Recombination: We will restrict the introduction to *discrete recombination*, which targets representation domains of individuals that are countable sets, e.g., integers, bit strings or finite vectors thereof. Further, we will only target *multi-point crossover* [18]. Here, a pre-defined number of points at variable positions along the parents' bit strings specify areas that are to be interchanged mutually as one atomic unit. The locations of those points are randomly regenerated for each recombination:



If only one point is selected the recombination process is referred to as *single-point crossover*. Analogously, *two-point crossover* calls for two intersection points, as illustrated above.

5) Mutation: Selection and recombination aim for identifying and recombining parts of promising individuals. However, both phases cannot generate individuals that are completely new in a sense that there are no parts of an individual that have never occurred in any individual of a preceding population. Hence, in order to increase the population diversity, and to thereby avoid stagnation at local optima, *mutation* introduces random changes of individuals.

6) Insertion/Replacement: If the population is set to be constant during evolution, a decision has to be made which of the newly created individuals are to be taken over to the next iteration and which of the individuals in the current population are to be discarded. One possible technique is *elitist* *reinsertion*, where the amount of newly generated individuals is less then the size of the population they have to be inserted into. Further, all newly generated individuals are inserted into the population and those individuals in the population showing the lowest fitness are replaced.

7) Termination: The decision on when to stop the loop is influenced by two factors.

- **Limitation of resources.** Usually, there is a time limit after which a solution has to be provided. Memory size is typically not an issue as the population size usually remains constant. However, if a memory-intensive fitness function is involved, a user-defined limit on memory usage can make sense.
- **Solution quality.** In general, it is not possible to decide if an optimal solution has been found. Consequently, one can stop the loop if the solution is "good enough", but sometimes even such a metric is not available. Thus, a further option is to stop if the fitness of newly generated individuals stagnates.

B. Basic Concepts

One aspect of a software program that is basic, not only in measurement-based timing analysis but in nearly all WCET analysis techniques, is flow of control:

Definition III.1. A *Control Flow Graph (CFG)* of a program $\mathcal{P}$ is a directed graph $G_{\mathcal{P}} = \langle V, E, v_0, \rangle$ with vertices V and edges $E \in V \times V$ with $v_0 \in V$ as the unique start of the program.

Control flow graph vertices express basic blocks, i.e., instruction sequences of maximal length where only the last instruction might be a jump. Control initially resides in v_0 and can then flow along the CFG edges through the various basic blocks.

Definition III.2. Given a program $\mathcal{P}$, an *input vector* iv assigns values to free variables in $\mathcal{P}$. A *test suite* is a set Γ of input vectors. A *trace* $\pi_{\mathcal{P}}(iv)$ denotes a time-stamped path in the control flow graph $G_{\mathcal{P}}$ that results from an execution of $\mathcal{P}$ where all free variables are assigned with respect to iv. We say a vertex v is *exercised* by Γ iff there is an $iv \in \Gamma$ such that v is in the path $\pi_{\mathcal{P}}(iv)$. v is *feasible* iff there exists an input vector iv such that v is exercised. The *maximal observed execution time (MOET)* for vertex v during an execution of input vector iv is denoted $MOET_{\mathcal{P}}(v, iv)$. We set $MOET_{\mathcal{P}}(v, iv)$ to be zero iff v is not in $\pi_{\mathcal{P}}(iv)$. In the following, we will omit $\mathcal{P}$ for the sake of readability and assume that all notations refer to the same program.

IV. About Optimism

We have already discussed that optimism is an inherent property of measurement-based timing analysis that emerges in the phase of execution time measurement. As a first step towards tackling the objective of reducing the chance of WCET underestimation in MBTA, we will start by presenting means to quantify optimism.

Recall that the phase of execution time measurement in MBTA is dedicated to derive worst-case execution time estimates for program segments. In the following we consider those segments to be basic blocks without loss of generality as other common segment concepts in MBTA (subgraphs, sequences between instruction points) are all composed of basic blocks. Under the premise that no hardware model is available in measurement-based timing analysis, the worst-case execution time of a basic block b is optimistically estimated to be the maximal observed execution time during a measurement performed on a given test suite Γ:

$$WCET_{est}(b, \Gamma) \quad = \quad max\{MOET(b, iv) \mid iv \in \Gamma\}$$

An ideal metric to derive how optimistic a WCET estimate for a basic block is, would precisely denote the difference between the estimate and the real worst-case execution time. As the real WCET is generally unknown, we cannot absolutely determine optimism. However, we can compare test suites in terms of optimism, i.e., a test suite that leads to a higher MOET for a basic block is less optimistic:

$$\Gamma_1 \ll_b \Gamma_2 \quad \equiv \quad WCET_{est}(b, \Gamma_1) > WCET_{est}(b, \Gamma_2)$$

where $\ll_b$ denotes the relation "less optimistic" for a basic block b. So far, the relation considers only a single basic block. In order to generalize the concept such that test suites for a program as a whole can be evaluated in terms of optimism, we need to generalize the notion of $WCET_{est}$. Recall that the phase of execution time measurement in MBTA happens before timing composition, i.e., we have no knowledge about how basic blocks contribute to the global WCET estimate that is derived in the timing composition phase. We do not know how often a basic block is exercised by the path that generates the overall WCET estimate. Thus, we consider all basic blocks to be equally important for the overall analysis and derive an *average WCET estimate* $WCET^{avg}_{est}$ over all basic blocks to yield a generalized version of $\ll$:

$$\begin{aligned} \Gamma_1 \ll \Gamma_2 \quad &\equiv \quad WCET^{avg}_{est}(\Gamma_1) > WCET^{avg}_{est}(\Gamma_2) \\ WCET^{avg}_{est}(\Gamma) \quad &= \quad \frac{1}{|V|} \sum_{b \in V} WCET_{est}(b, \Gamma) \end{aligned}$$

To summarize, if a test suite Γ_1 leads to higher local WCET estimates in average than test suite Γ_2, we consider it to be less optimistic. Further, we claim that the chance of underestimating the global WCET is lower when using Γ_1 than using Γ_2 in the context of measurement-based timing analysis. Recall that this claim does only hold if one has no knowledge about how the global WCET estimate is derived in the phase of timing composition. For instance, if IPET is used for composition, $WCET_{est}$ would rather take only those basic blocks into consideration that are in the WCET path, weighted by their execution frequencies. However, in this paper we restrict our investigations to the more general version $WCET^{avg}_{est}$.

The "less optimistic"-relation $\ll$ will be used later for an experimental safety evaluation of our proposed input data generation technique FROO.

V. FROO

FORTAS *Reduction of Optimism*, in short FROO, is an input data generation technique that combines the principles of genetic algorithms and model checking. The objective for FROO is to maximize the average WCET estimate $WCET_{est}^{avg}$ of the set of input vectors that are used in the phase of execution time measurement in MBTA. That means, the maximal observed execution times for all basic blocks are *jointly* to be increased during the generation process in a best effort manner.

The motivation for using model checking in combination with a genetic algorithm results from our previous work [19]: Random input data generation is shown to be very fast but cannot guarantee to cover all parts of a program. Model checking, on the other hand, can be used to generate test suites that satisfy sophisticated structural coverage criteria, however, it requires considerable resources of computation.

The basic idea in this paper is to use model checking for generating a test suite that satisfies basic block coverage, thereby guaranteeing that each basic block is exercised at least once. This test suite forms the seed for a genetic algorithm that iteratively attempts to further improve the quality of the population.

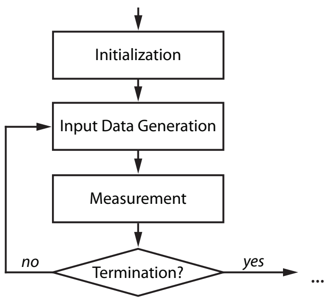


Fig. 3. Input data generation in MBTA

First, let us take a closer look at the MBTA phase of execution time measurement. Figure 3 illustrates the general structure: an input data generation module iteratively produces input vectors that are executed on the target and measured. FROO is embedded into this general procedure as depicted in Figure 4.

In the following, we will focus on the modules of FROO:

1) *Initialization*: Input vectors form individuals. Consequently, populations are test suites. For example, if the program under scrutiny has two free integer variables, an individual iv could have the form:

$$iv_k \quad = \quad [5, -1]$$

...where each index k refers to a unique variable. Populations are denoted P_i, where i is the iteration in which the according population is generated. We want

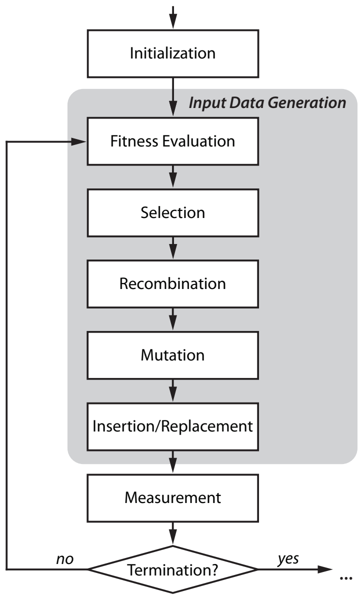


Fig. 4. FROO in the context of measurement-based timing analysis

the initial population (seed) P_0 to satisfy basic block coverage. First, we generate a test suite Γ_B that satisfies basic block coverage with *FShell*[1], a prototype implementation based on the principles described in [20], [21]. FShell relies on the C Bounded Model Checker (CBMC), version 3.8 [22]. The input to FShell is a test suite specification, expressed by the *FShell Query Language (FQL)* [23]. In our case, FQL queries are of the form:

```
IN @FUNC(foo) cover @BASICBLOCKENTRY
```

The population size must be large enough to include Γ_B. In our setup, we set $|P_0| = 200$. An additional random test suite Γ_R is generated to fill up $P_0 = \Gamma_B \cup \Gamma_R$.

2) *Fitness Evaluation*: A straightforward way to evaluate an input vector's fitness F_i at iteration i would be to take the average WCET estimate:

$$F_i(iv) \quad = \quad WCET_{est}^{avg}(\{iv\})$$

However, this would not specifically promote those input vectors that yield to new maximal observed execution times of basic blocks. Hence, we use a fitness function that considers the execution time distance to the maximal execution time, observed with respect to the current population. In order to account for jitter, and to even further promote distances that are close to the MOET, we square the distance.

$$F_i(iv) = \sum_{b \in V} \frac{1}{1 + [WCET_{est}(b, P_i) - MOET(b, iv)]^2}$$

3) *Selection*: Inspired by [24], we use roulette-wheel selection and pick 50% of the individuals in the current population for recombination.

[1]http://code.forsyte.de/fshell

4) *Recombination*: As suggested in [25], we choose single-point crossover.
5) *Mutation*: The mutation probability is set to 2%, i.e., for each individual in the offspring, there is a chance of 0.02 to be mutated at one randomly chosen variable. For a variable that is chosen to be mutated, a new uniformly distributed value in the variable's domain is generated.
6) *Insertion/Replacement*: We use a modified version of elitist reinsertion: the whole offspring is inserted into the current population, where those individuals showing the lowest fitness are replaced. An important exception is that we want to keep alive those input vectors satisfying basic block coverage in Γ_B. Hence, those input vectors are never replaced. Our experiments reveal that this modification improves FROO considerably.

Note that this paper is not intended to be a detailed investigation of how genetic algorithms can be used in measurement-based timing analysis. Although we find the question very interesting how specific parameters and strategies of a genetic algorithm affect optimism, we only want to present a proof of concept for FROO in this paper. Studies on how to optimize FROO are subject to future research.

VI. EVALUATION

In [19], we introduce *BPG-II*, an input data generation technique with the objective of covering the worst-case temporal behavior of program segments in a best-effort manner. Our experiments revealed that BPG-II is superior to all other investigated input data generation techniques in terms of minimizing optimism. However, in the following we will show by our experiments that FROO is even less optimistic than BPG-II and more efficient in terms of input vector generation time.

A. BPG-II Revisited

BPG II uses FShell to generate multiple **different** test suites (the FShell option `MULTIPLE_COVERAGE` enables the option to yield mutually different test suites for an FQL expression), each satisfying the union of basic block and condition coverage. The FQL query has the form:

```
IN @FUNC(foo)
cover (@BASICBLOCKENTRY | @CONDITIONEDGE)
```

The union of all test suites forms the result. Thus, BPG-II guarantees that all basic blocks are exercised at least once. Also, the option to generate mutually different test suites supports that basic blocks are exercised with different execution histories. Although the input vectors generated with BPG-II are of high quality in terms of optimism, it suffers from scalability issues that often go hand in hand with model checking, i.e., the number of input vectors produced per time interval is rather low. In particular, the more mutually different test suites are generated, the more difficult it gets for the model checker to find new instances. For all our benchmarks we observed that the amount of time needed for finding a new, dissimilar test suite grows polynomially.

B. Target and Measurement

All measurements are performed on the TriCore 1796 microprocessor by Infineon. Programs under analysis are compiled with HighTec's GCC[2] compiler. The TriCore 1796 includes branch prediction, a superscalar pipeline and an instruction cache. Furthermore, it provides *On-Chip Debug Support (OCDS)* level 2 for cycle-accurate execution tracing. We utilize the Lauterbach LA-7690 Powertrace device to extract both timing and flow of control. Code instrumentation is not required with this setup and measurements are cycle-accurate.

C. Benchmarks

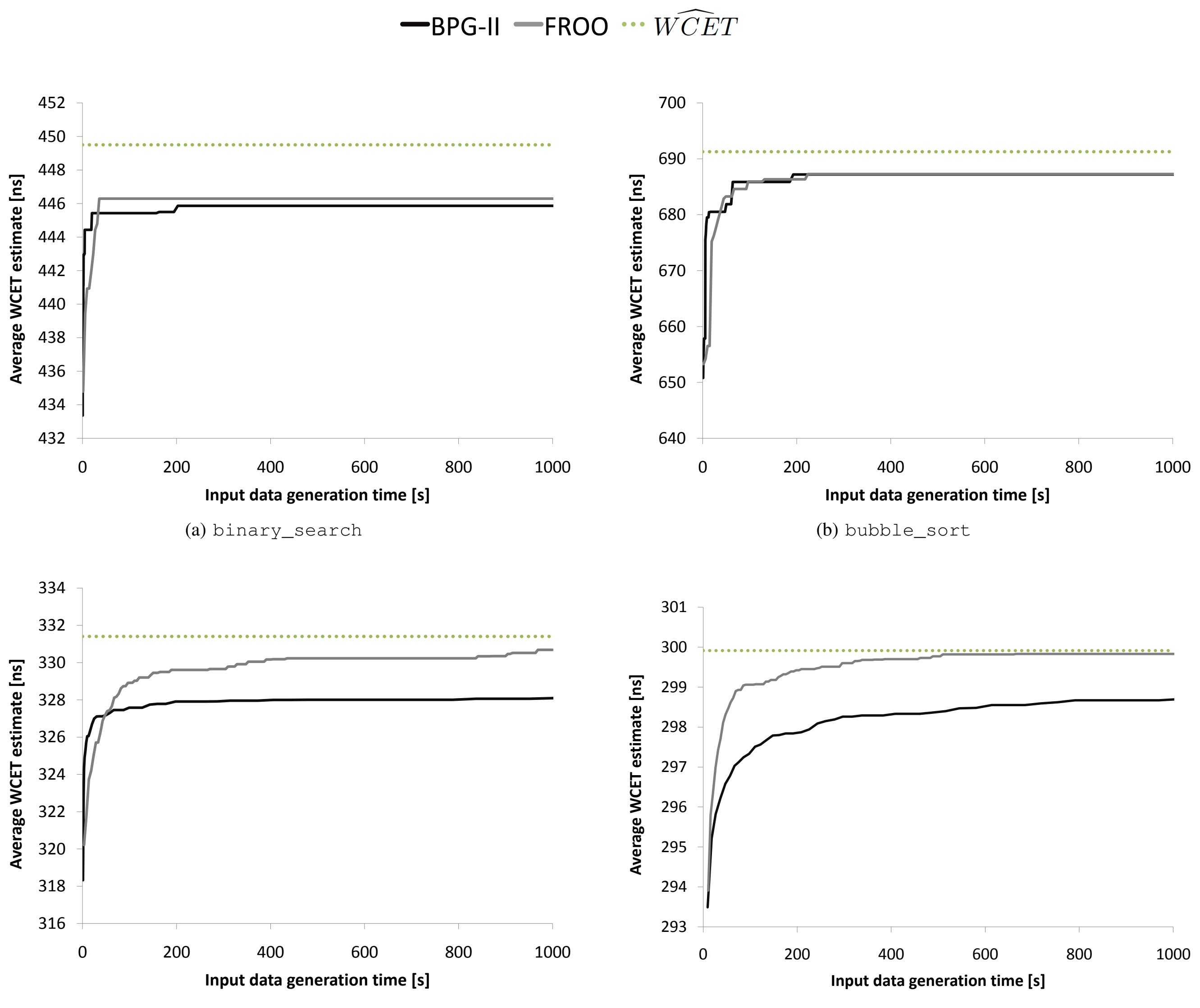
For benchmarking our methods, we use the following programs in ANSI C. Lines of code are counted via `CLOC`[3].

- `binary_search`: An implementation of the binary search algorithm, taken from the *Mälardalen WCET Project*[4]. The benchmark has 44 lines of code and 14 CFG nodes. The only free variable is the key to search for, all respective input vectors are therefore of size 1.
- `bubble_sort`: Another problem from the Mälardalen WCET Project. The size of the input list for `bubble_sort` is reduced from 100 to 10 as we utilize a bounded model checker that does not scale well for this particular benchmark. The benchmark has 44 lines of code and 15 CFG nodes. Input vectors are of size 10.
- `lift_control`: The central control unit for an elevator that we translated to C has 210 lines of code. It is originally intended for the *Java Optimized Processor* [26]. The original version is used in the field and can be found on the web[5]. The according CFG has 119 nodes, input vectors hold 68 variables.
- `engine_control`: An engine control unit from our industry partners in the automotive domain. The code is generated by Matlab/Simulink and involves a more complex control flow structure than `lift_control`. It has 976 lines of code, its CFG has 398 nodes, and an input vector consists of 33 variables.

D. Results and Discussion

Both BPG-II and FROO are evaluated according to optimism of the accumulated input vectors over time. Figure 5 outlines our experimental results. For all benchmarks, the horizontal axis denotes the duration Δt of the respective input data generation process. Then, for any Δt, the vertical axis illustrates the corresponding average WCET estimate $WCET_{est}^{avg}(\Gamma)$ for all input vectors Γ that were generated in Δt. The dotted line denotes $\widehat{WCET} = WCET_{est}^{avg}(\Gamma_\cup)$: a cross-experimental approximation of the real average WCET for all basic blocks with $\Gamma_\cup$ as a reference test suite including all input vectors ever used for the respective benchmark.

[2]http://gcc.gnu.org/
[3]http://cloc.sourceforge.net
[4]http://www.mrtc.mdh.se/projects/wcet/benchmarks.html
[5]http://www.soc.tuwien.ac.at/trac/jop/browser/java/target/src/bench/jbe/lift



(a) `binary_search` (b) `bubble_sort` (c) `lift_control` (d) `engine_control`

Fig. 5. FROO versus BPG-II

The results show that FROO produces less optimistic input vectors than BPG-II for all benchmarks. For `bubble_sort`, the difference of $0.1‰$ is negligible. The biggest difference was encountered for benchmark `lift_control` with an improvement of 0.8%. These differences are rather small. Both BPG-II and FROO produce results close to $\widehat{WCET}$.

In contrast, the time Δt needed to get to the maximum, is far less for FROO compared to BPG-II. For instance, using benchmark `engine_control`, FROO yields BPG-II's maximum of $298,74$ in 6% of BPG-II's time. Analogously, for `binary_search` and `lift_control` FROO needs 18% and 7% of BPG-II's time, respectively. Only in benchmark `bubble_sort`, FROO and BPG-II perform equally both in terms of optimism and runtime performance.

The fact that BPG-II is outperformed by FROO is due to the polynomial complexity of BPG-II. FROO on the other hand needs constant time for generating a new population.

Independent of the benchmark it takes FROO about $4s$ to generate a new population. This is rather slow, as we calculate $WCET_{est}^{avg}$ not from main memory but from a database. Also, we do not cache any data. We believe that by addressing these issues and by tweaking the parameters of the genetic algorithm, FROO's performance can be further improved considerably. This paper is intended to be a proof of concept, though. Optimizations are subject to future work.

VII. Related Work

This paper is basically a follow-up on our investigations presented in [19] which is—to the best of our knowledge—the only work that empirically evaluates test suites in terms of how well the worst-case temporal behavior of program segments is covered. There are two major improvements that are new in this work:

1) We figured, that the concept of *relative safety* from [19]

is inconvenient in a sense that it might be misinterpreted easily, as the term safety refers to a binary attribute (either a system is safe or it is not). In contrast, we used relative safety to describe what is actually optimism. This work suppresses possible causes for diverging interpretations and introduces concise and more convenient definitions on how to quantify optimism.
2) [19] presents BPG-II, an input data generation technique that is shown to generate test suites with a very low amount of optimism that beats all other investigated input data generation techniques in this regard. In this paper, we demonstrate that FROO produces even less optimistic test suites than BPG-II. Further, FROO safes analysis time by orders of magnitude.

Genetic algorithms have been studied quite intensively for WCET analysis [25], [27]–[31].

All related work on genetic algorithms targets **global** WCET estimation, i.e., the fitness of individuals is defined in terms of end-to-end execution times that is to be maximized by the genetic algorithm. We, however, try to jointly maximize **local** WCET estimates for all segments of a program. Beside this major difference in assigning fitness, also the way we seed the search is deviant,i.e., we use model checking to yield a high-quality initial population where usually random techniques are used.

As there is a huge amount of related work, we will highlight the numerous contributions more specifically:

In [29], [32], genetic algorithms are evaluated with a primary focus on aerospace applications. Other search-based input data generation techniques are investigated as well such as hill-climbing and simulated annealing.

The automotive domain, on the other hand, has been targeted analogously by Wegener et al. [29], [33].

A comparison of genetic algorithms and static analysis for WCET estimation is given in [25], [29], [34] where it is also illustrated that optimism in search-based methods can counteract pessimism by static analysis tools.

In general, the applicability of genetic algorithms for WCET estimation depends on the structure of the program under scrutiny. Gross et al. investigate the impact of specific software properties on how well search-based methods perform is investigated in [35].

In [24], [31], the fitness function does not exclusively depend on end-to-end execution times but also on branch prediction misses, cache misses, or number of loop iterations.

VIII. CONCLUSION

Using a metric to quantify optimism in measurement-based analysis and a set of representative benchmarks, we have shown empirically for the TriCore 1796, a commonly used microprocessor in the automotive domain, that FROO is an adequate tool for generating test suites that are to be used in the phase of execution time measurement in MBTA.

As this paper is intended to present a proof of concept for FROO, the next steps are to further optimize the approach. Another idea is to combine model checking with a different search heuristic, as for instance, *Particle Swarm Optimization* [36], [37].

ACKNOWLEDGMENT

The research leading to these results has received funding from the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) within the research project "Formal Timing Analysis Suite of Real-Time Systems" (FORTAS-RT) under contract P19230-N13. We especially thank our colleague Benedikt Huber who came up with the idea of using model checking for seeding.

REFERENCES

[1] A. Burns and A. J. Wellings, *Real-Time Systems and Programming Languages*, 2nd ed. Addison Wesley, 1996.

[2] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 1–53, 2008.

[3] R. Kirner, I. Wenzel, B. Rieder, and P. Puschner, *Intelligent Systems at the Service of Mankind*. Augsburg, Germany: UBooks Verlag, Jan. 2006, vol. 2, ch. Using Measurements as a Complement to Static Worst-Case Execution Time Analysis, pp. 205–226, iSBN: 3-86608-052-2.

[4] S. Stattelmann and F. Martin, "On the use of context information for precise measurement-based execution-time estimation," in *Proceedings of 10th International Workshop on Worst-Case Execution Time (WCET) Analysis*, B. Lisper, Ed. Austrian Computer Society, July 2010, pp. 68–79.

[5] I. Wenzel, R. Kirner, B. Rieder, and P. Puschner, "Measurement-based timing analysis," in *Proc. 3rd Int'l Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, Porto Sani, Greece, Oct. 2008.

[6] M. Zolda, S. Bünte, and R. Kirner, "Towards adaptable control flow segmentation for measurement-based execution time analysis," in *Proc. 17th International Conference on Real-Time and Network Systems (RTNS)*, Paris, France, Oct. 2009.

[7] A. Betts and G. Bernat, "Tree-based WCET analysis on instrumentation point graphs," in *Proc. 9th IEEE International Symposium on Object-oriented Real-time distributed Computing*, Gyeongju, Korea, Apr. 2006.

[8] P. P. Puschner and A. V. Schedl, "Computing maximum task execution times - a graph-based approach," *Real-Time Systems*, vol. 13, no. 1, pp. 67–91, July 1997.

[9] Y.-T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," in *DAC '95: Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*. New York, NY, USA: ACM, 1995, pp. 456–461.

[10] P. Puschner and C. Koza, "Calculating the maximum execution time of real-time programs," *Real-Time Syst.*, vol. 1, no. 2, pp. 159–176, 1989.

[11] M. Zolda, S. Bünte, and R. Kirner, "Context-sensitivity in IPET for measurement-based timing analysis," in *4th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'10)*, Oct. 2010.

[12] C. Darwin, *On the Origin of Species by Means of Natural Selection*. London: Murray, 1859.

[13] J. H. Holland, *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI, USA: University of Michigan Press, 1975.

[14] J. Wegener, "Evolutionärer Test von Realzeit-Systemen," Ph.D. dissertation, Humboldt-Universität zu Berlin, 2001.

[15] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989.

[16] T. Blickle and L. Thiele, "A comparison of selection schemes used in genetic algorithms," ETH Zürich, Switzerland, Tech. Rep. 11, 1995.

[17] J. E. Baker, "Reducing bias and inefficiency in the selection algorithm," in *Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application*. Hillsdale, NJ, USA: L. Erlbaum Associates Inc., 1987, pp. 14–21. [Online]. Available: http://portal.acm.org/citation.cfm?id=42512.42515

[18] W. M. Spears and K. A. D. Jong, "An analysis of multi-point crossover," in *Proceedings of the First Workshop on Foundations of Genetic Algorithms*, G. J. E. Rawlins, Ed. Morgan Kaufmann, 1991, pp. 301–315.

[19] S. Bünte, M. Zolda, M. Tautschnig, and R. Kirner, "Improving the confidence in measurement-based timing analysis," in *Proc. 14th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC'11)*, Mar. 2011, To appear.

[20] A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith, "Fshell: Systematic test case generation for dynamic analysis and measurement," in *Proceedings of the 20th International Conference on Computer Aided Verification (CAV 2008)*, ser. Lecture Notes in Computer Science, vol. 5123. Princeton, NJ, USA: Springer, July 2008, pp. 209–213.

[21] ——, "Query-driven program testing," in *Proceedings of the Tenth International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2009)*, ser. Lecture Notes in Computer Science, N. D. Jones and M. Müller-Olm, Eds., vol. 5403. Savannah, GA, USA: Springer, January 2009, pp. 151–166.

[22] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, ser. Lecture Notes in Computer Science, K. Jensen and A. Podelski, Eds., vol. 2988. Springer, 2004, pp. 168–176.

[23] A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith, "How did you specify your test suite ?" in *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE 2010)*, Sep. 2010.

[24] U. Khan and I. Bate, "WCET analysis of modern processors using multi-criteria optimisation," in *Proceedings of the 1st International Symposium on Search Based Software Engineering (SSBSE '09)*, 2009, pp. 103–112.

[25] P. Atanassov, "Experimental assessment of worst-case program execution times," Ph.D. dissertation, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 2003.

[26] M. Schoeberl, W. Puffitsch, R. U. Pedersen, and B. Huber, "Worst-case execution time analysis for a Java processor," *Software: Practice and Experience*, vol. 40/6, pp. 507–542, 2010.

[27] P. Puschner and R. Nossal, "Testing the results of static worst-case execution-time analysis," in *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, dec 1998, pp. 134 –143.

[28] J. Wegener, M. Grochtmann, and B. Jones, "Testing temporal correctness of real-time systems by means of genetic algorithms," in *International Software Quality Week (QW'97)*, May 1997.

[29] N. Tracey, "A search-based automated test-data generation framework for safety-critical software," Ph.D. dissertation, University of York, Department of Computer Science, 2000.

[30] H.-G. Gross, B. Jones, and D. Eyres, "Evolutionary algorithms for the verification of execution time bounds for real-time software," in *Applicable Modelling, Verification and Analysis Techniques for Real-Time Systems (Ref. No. 1999/006), IEE Colloquium on*, Jan. 1999, pp. 8/1 –8/8.

[31] I. Bate and U. Khan, "WCET analysis of modern processors using multi-criteria optimisation," *Empirical Software Engineering*, vol. 16, pp. 5–28, February 2011.

[32] N. Tracey, J. Clark, and K. Mander, "The way forward for unifying dynamic test case generation: The optimisation-based approach," in *In International Workshop on Dependable Computing and Its Applications*, 1998, pp. 169–180.

[33] J. Wegener, H. Sthamer, B. F. Jones, and D. E. Eyres, "Testing real-time systems using genetic algorithms," *Software Quality Control*, vol. 6, pp. 127–135, October 1997.

[34] J. Wegener and F. Mueller, "A comparison of static analysis and evolutionary testing for the verification of timing constraints," *Real-Time Systems*, vol. 21, no. 3, pp. 241–268, November 2001.

[35] H.-G. Gross, "A prediction system for evolutionary testability applied to dynamic execution time analysis," *Information and Software Technology*, vol. 43, no. 14, pp. 855 – 862, 2001.

[36] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Neural Networks, 1995. Proceedings., IEEE International Conference on*, vol. 4, Aug. 2002, pp. 1942–1948.

[37] ——, "A discrete binary version of the particle swarm algorithm," in *Systems, Man, and Cybernetics, 1997. 'Computational Cybernetics and Simulation'., 1997 IEEE International Conference on*, vol. 5, Oct. 1997, pp. 4104 –4108 vol.5.