Testing the Results of Static Worst-Case Execution-Time Analysis *

P. Puschner and R. Nossal†
Technische Universität Wien, A-1040 Wien, Austria
peter@vmars.tuwien.ac.at

Abstract

Analytically derived worst-case execution time (WCET) bounds are prone to errors, because they often rely on information provided by the user. This paper presents a method for testing the results of static WCET analysis. The proposed test method is a black-box test method that uses a genetic algorithm (GA) for test-case generation. Important properties of the method are (a) that it requires minimal information about possible input data from the user and (b) that the GA guides data generation into directions that have a good chance to yield the real WCET of the program under test. Experimental results show that GA-based testing produces results of high quality.

1 Introduction

The knowledge of task execution times is crucial for the development and verification of dependable real-time systems. This information must be available to allocate resources, to guide scheduling decisions, and to demonstrate the temporal correctness of real-time application programs.

In recent years researchers developed concepts and tools for computing worst-case execution time (WCET) information for real-time tasks by static analysis [11, 21, 12, 15]. These tools derive WCET bounds for tasks as follows: they analyze the code of the tasks for possible execution paths and then model the timing of the code on the target hardware. While the latter step of the analysis can be fully automated, the first step usually has to be supported by the programmer — the programmer annotates the task code with information about the possible execution paths. Due to this need for human interaction, WCET analysis is prone to errors. The correctness of the results of static WCET analysis must therefore be tested.

We developed a task-test method that increases the confidence on the results of static WCET analysis. This method executes the task under test, compares the execution times of the test runs to the computed WCET bound, and reports a timing error if a measured time exceeds the computed bound. The data generator for the tests is based on a genetic algorithm (GA). It generates test data that promise to yield long execution times based on feedback about the execution times of the task in the preceding test runs.

The proposed test method uses a test data generation paradigm known as *dynamic test data generation* [17]. Dynamic test data generation treats parts of a program under test as functions. These functions are evaluated by executing the tested program. The returned function values indicate the quality of test data with respect to a given criterion. Test data generation tries to find inputs that maximize (or minimize) the function values to fulfill this criterion. Dynamic test data generation thus maps the problem of generating test data onto the well-known problem of function maximization. For WCET testing the function to be maximized is the function that yields the execution time of the task for given input data. We realize this function by measuring the execution time of the task.

Note that our test method, like any dynamic test generation method, has its limits. Most tasks will have a vast input space, of which only a small number of inputs are tested within the available test time. Given such a limited set of input samples, no search strategy can guarantee that the tests hit the WCET or even yield tight estimates. Therefore, if WCET tests for a task do not invalidate the result of static WCET analysis, then this may increase the *confidence* in the correctness of the computed WCET bound. Such a result, however, *never guarantees* the absence of a failure.

Dynamic test data generation was introduced in [17]. An extension of this idea can be found in [14]. The first approaches to dynamic test data generation used

*This work has been supported by the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) under contract P12476-INF".

†R. Nossal is now with Siemens AG, Regensburg, Germany.

simple, local search strategies that got easily stuck in local maxima or minima. To avoid this problem, more sophisticated global search techniques were introduced. Jones et al. [9] and McGraw et al. [16] use Genetic Algorithms (GA) for dynamic test case generation, and in [25] WCET testing is presented as an example for GA-based testing.

The problem of real-time systems testing was covered in various publications [7, 2, 5, 23]. None of them, however, addressed the issue of testing the timing of single tasks. Only recently, researchers started to work on methods for testing execution-time bounds. Work, very similar to ours, gives a detailed description of GA-based WCET testing [27, 26] and compares evolutionary testing to static WCET analysis [18]. The here-presented work describes a novel test environment for dynamic WCET tests. The paper focuses on GA-based testing and compares its results to those of random testing, best-effort testing, and static analysis. Also, it demonstrates shortcomings of GA-based test data generation by means of a set of test programs we designed.

Section 2 describes our test environment and the proposed test approach. Section 3 focuses on the use of GAs for WCET testing. It gives a short introduction to GAs and shows how we apply them for controlling the tests and generating test data. Section 4 describes our experiments which compared GA-based testing to other test methods. It presents the lessons we learned from these experiments. Section 5 summarizes and concludes the paper.

2 Test Environment

The test environment consists of the target computer system, the host computer system, and a high-resolution timer/counter.[1] The target computer system has the same hardware timing characteristics (CPU clock rate, memory access times, etc.) as the computer on which the software under test will be executed in the real application. During the tests the task under evaluation executes on this system and the properties of interest—the start and the stop events of tasks generated at the beginning and the end of the tasks—are observed. The host computer system is a PC that accommodates the software development tools, the user interface of the test environment, and the test control software. The timer counts the number of CPU clock ticks between the start and stop events generated for the task under investigation. The durations measured by the timer are corrected on the host system to account for the constant overhead of the instructions that trigger the timer and start the task.

Note, that the environment is conceived for embedded target systems with scarce CPU and memory resources. We execute a small software system on the target and use the host for the resource-intensive operations. For targets with sufficient resources the host is not needed; the entire software can run on the target.

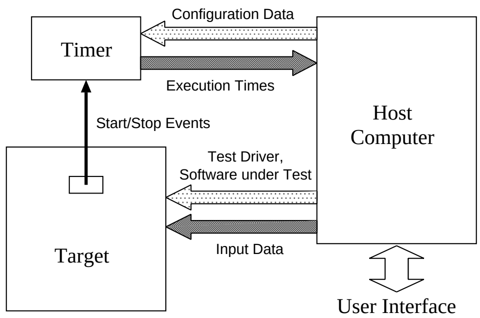

Figure 1. The test environment

Figure 1 shows the setup of the test environment as well as its data paths. Dotted arrows mark the data paths for initializations, gray arrows show the data flow during the operation of the test system.

2.1 Software Components

The most important software components of the test environment are, on the one hand, the *task* under investigation and a simple *test driver* that reside on the target system and, on the other hand, the *user interface* and the *test control software* on the host system.

The test environment assumes that the *tested task* is a so-called *simple task* [13]. A simple task communicates with other tasks and the environment solely by the exchange of messages. The operating system assures that all input messages of the task are available at the point in time the task starts. Outputs are written to the output message buffers of the task. The output message buffers are emptied by the operating system after the completion of the task. Thus all tasks execute completely without blocking I/O, synchronization, and communication.

The simple task model precludes that one task blocks another active task. It allows us to test the timing of single tasks in separation. As in the real application, the test environment provides the inputs to the tested task before the task starts. For repetitive tasks with a state, the task execution times may depend on the values of the state variables at task activation time. To handle state dependencies, the test

[1] We use an external counter since not all target boards have timers that provide the clock resolution we are interested in.

environment also sets the state variables of the tasks before execution. Thus, for the test environment the interface of a task comprises not only the inputs and outputs of the task, but also includes the task state.

The *test driver* receives the test data for the task from the host. For each set of input data it copies the data to the input buffers of the task and executes the task. To support the timing measurements the test driver generates trigger signals for the timer. It writes to an output port, immediately before the task starts and after the task has terminated. Further, the test driver watches the proper termination of the task. It sets a local timeout to some value greater than the WCET (e.g., 2∗WCET) before it generates the task-start signal and starts the task. The purpose of this timeout is to avoid that the tests get stuck in an endless loop of a faulty task being tested.

Via the *user interface* the programmer selects the task to be tested together with the description of the task interface and the computed WCET bound. Further he/she defines the test termination criteria, starts, and controls the tests by means of this interface.

The *test control software* implements the main control function for the tests. It obtains the measured task execution times from the timer and evaluates them against the WCET. It repeatedly issues calls to the *test data generator* to produce new test data and it sends these test data to the target. Finally, the test control software evaluates the termination criteria and decides about the termination of the tests. In the approach described in this paper a GA realizes the main functions of the test control software (see Section 3).

2.2 Testing Steps

The following information is needed to start a WCET-test suite for a task:

- A specification of the interface of the task. This description comprises all input, output, and state data items together with their types, ranges, and the specified relations between the different data values.
- The code of the task to be tested.
- The WCET bound of the task computed by static WCET analysis.

The tests consist of two phases. In the *initialization phase* the programmer selects the task to be tested. The user interface of the test environment displays the task interface and the WCET bound and allows the programmer to refine the interface description. Further the programmer defines the stopping criteria for the tests (see Section 3.3.3).

After the task selection, the programming environment generates task-specific versions of the test control software and the test driver. To this end, it reads the input-data description of the task and instantiates the available templates for the test control software and the test driver accordingly.

Finally, the test driver and the task are linked and loaded onto the target. The timer is initialized. After completion of these steps the test driver is ready to receive input data sets and to start test execution.

In the *test phase* the timer measures the durations between start and stop events and transmits the results to the host. The test control software, executing on the host, reads the measured execution times, evaluates the stopping criteria for the tests, calls the test data generator, and sends the new test data to the test driver on the target. The test driver, in turn, executes the tested task with the data received from the host.

3 Test Control and Data Generation

This section deals with the issue of generating test data. After an introduction, it describes Genetic Algorithms in general before it takes a closer look at their adaptation for use in a WCET test environment. Finally, the characteristics of the test setup are discussed.

3.1 Heuristic Generation of Test Data

It is obvious that, due to the large number of input data sets, an exhaustive application of all data sets is impossible. Thus, a method must be used to heuristically derive test data sets that lead to a long or even better the worst-case execution time of the task under test. Stochastic optimization methods like Genetic Algorithms (GA) [8], Simulated Annealing [10], or Tabu Search [3] are perfectly suited for this purpose. They facilitate a goal-oriented optimization without the danger of being stuck in local optima like traditional hill climbing. The major advantage of GAs over the other two methods is that it exploits different parts of the search space at the same time, a feature that is referred to as the *inherent parallelism* of GAs. Because of this characteristic, GAs allow to quickly narrow down the search space to promising regions. Neighborhood search methods like Simulated Annealing and Tabu Search, on the other hand, are superior for systematic evaluation of the search space.

For the first implementation we selected GAs as test data generator. Alternatively, we consider to enhance the test data generator with a neighborhood search for future implementations. In this case the GA identifies the regions of long execution times, which are then exploited by the neighborhood search.

3.2 Introduction to Genetic Algorithms

Genetic Algorithms (GA) [8] are a stochastic method used to solve NP-hard optimization problems in many fields. GAs constitute a so-called "uninformed" search strategy. This term refers to the fact that the algorithm itself does not have any knowledge on the problem it solves. The problem specific knowledge is entirely incorporated into the *fitness function*, which calculates a basic *fitness value* of a candidate solution produced by the GA; additional constraints can be incorporated into the function by so-called *penalty terms*. A penalty term reduces the fitness of a solution provided that it violates a constraint. The resulting, overall fitness value is used as a feedback to the algorithm.

A problem to be solved by a GA has to be *encoded* into bit strings. Each bit string, called a *gene* encodes one part of the information on the problem solution. Genes are grouped to *chromosomes*; one or more chromosomes form an *individual*. Each individual encodes a complete solution to the given problem. The quality of this solution, i.e., the fitness of the individual is assessed by the fitness function. A certain number of individuals form a *population*.

The GA imitates the process of evolution of nature by taking one population as parent generation and creating an offspring generation. The algorithm selects the best, i.e., fittest, individuals of the parent population. Here the fitness of an individual comes into play. The better the fitness value of an individual, the greater is the probability that it is selected for reproduction. After the selection the GA *mutates* some of the genes of the selected individuals by flipping bits according to a given mutation rate and performs *cross-over* between the individuals by swapping parts of chromosomes. As a result of this process a new population, a child generation, is created, which is then evaluated. The whole process iterates until a solution of sufficient quality is found.

While the GA is capable of generating new solutions to a problem that has been properly encoded, the derivation of an encoding scheme and the selection of a fitness function are up to the user. The major difference between conventional optimization techniques like the various branch-and-bound approaches and GAs is the way a solution is generated. Whereas with conventional methods the user must provide the knowledge on how to construct a solution, it is sufficient to know how to assess a given solution with GAs. The "construction" of new solutions is autonomously done by the GA.

For a comprehensive description of GAs and their function the reader is referred to [24, 4]. The GA-library that forms the basis of the GA used in the test environment is presented in [19].

3.3 GA-Adaptation for WCET Testing

In the following we will concentrate on the "intelligence" of the GA, i.e., the problem encoding and the fitness function, which have to be defined by the user. Both have to be adapted such that the GA can operate as an input data generator for the actual test driver.

3.3.1 Problem Encoding

The GA is dedicated to produce sets of test data, which are handed to the task under test. Hence the individuals must represent the input data space of the tested task.

One gene is used for each input data item of the task. The value range of the gene must reflect the value range of the input data. However, regardless of the actual data type of an input data item, the GA is only capable of handling integer values. If the input data item has a different type an appropriate mapping must be applied. The following table lists possible input data types and their encoding as integer genes.

Table 1. Encoding of various data types

Data Type	Encoding
Integer	1 gene
Char	1 gene with value range 0 to 255
String(n)	n character genes
Fixed point number	2 genes, 1 representing integer part, 1 representing fractional part
Floating point number	3 genes, 1 representing integer part of mantissa, 1 fractional part of mantissa, 1 representing exponent

Complex data types like arrays or structures are encoded as sets of the above basic types.

The next step in the encoding hierarchy are chromosomes. There is no general rule how to arrange the input data to form chromosomes. The simplest way is to devise one chromosome, which comprises all input data items. Depending on the shape of the particular input space more sophisticated encoding schemes can be applied. In the matrix multiplication example, which will be presented in Section 4, the input space is made up of two equally sized matrices. In this case we constructed a GA that uses two chromosomes, one for each matrix.

The set of all chromosomes forms one individual. Each individual thus represents one complete set of input data for the task to be tested.

3.3.2 Fitness Function

The evolutionary process inherent to GAs creates new individuals in each generation. Each of these individuals is handed to the fitness function, which conducts the assessment of its quality. The fitness value of an individual is fed back to the GA to guide the further evolution.

For the purpose of deriving execution time bounds, the fitness of an individual (i.e., of a set of input data) is given by the *execution time* that the task under test needs to process the input data. The larger the execution time for a set of input data, the better is the fitness value of this data set. Hence the fitness of a set of input data is determined by executing the task under test with the given input data and measuring its execution time. The GA tries to maximize the fitness value, thus eventually arriving at the WCET or at an execution time close to the WCET.

Note that there is a fundamental difference between conventional applications of GAs for optimization and the test strategy presented in this paper. When normally applying a GA, the primary interest is in the individual that yielded the optimal performance. This individual represents the optimal solution to the given problem. The actual performance of the solution is only of secondary importance. In the GA-application shown in this paper, however, the significance of the individual and its performance is reversed. As the whole test setup is dedicated to assessing the correctness of WCET bounds, the main concern here is the performance of an individual, since it stands for the execution time of the tested task with the given input data. The set of input data items that yield this execution time, i.e., the individual itself, is only of importance if we are interested in more details of the scenario that caused the worst case. Such detailed information is, for example, helpful if we have to investigate why the results of static WCET analysis did not hold against the measured times.

3.3.3 Stopping Criterion

Two types of stopping criteria are common for GAs, (1) a certain fitness threshold is reached, or (2) a maximum number of generations, that has been specified by the user, has been reached. For our purpose we have replaced the second criterion by stopping after a maximum number of individuals, because this can be easier related to the size of the input space.

At first glance criterion 1 seems well suited for our purpose. Since the setup aims at assessing the analytically derived WCET-bounds, stopping the test when the bound is reached appears to be a reasonable stopping criterion. If, however, the analytically derived WCET bound is smaller than the actual WCET of the task, the test would never discover this failure.

To levy this difficulty we suggest a stopping criterion that is made up of the two basic criteria. First, we apply the maximum number of individuals criterion. We choose the number of individuals in relation to the size of the input space, i.e., the larger the input space the greater the maximum number of individuals. Second, the fitness threshold is set to a value slightly beyond the analytically derived WCET bound, i.e., $threshold = WCET_{analytic} + \epsilon$, where ϵ is a small positive number. Using a fitness threshold that is greater than the WCET bound derived from static analysis causes the GA to stop if the analytically-derived WCET bound is exceeded, i.e., the GA discovered a failure.

3.4 Properties of the Test Setup

Goal-Directed Test Data Generation. Compared to random testing the GA-based strategy operates in a goal-directed manner. While in the beginning it tests a variety of different input patterns, the evolutionary process directs the testing into promising directions that are likely to produce longer execution times. This guidance is inherent to the GA, which tries to maximize the fitness of individuals.

Code Inspection Not Required. The inherent guidance, which has just been mentioned, also relieves the user from evaluating the task under test in order to exploit input data yielding longer execution time. While other methods of test data generation force the user to carefully inspect the task to find such data sets, the GA-based approach does not require any code review. Promising sets of input data are discovered by "chance" and are automatically explored further.

Portability. While static WCET analysis needs a special compiler and exact knowledge about the timing characteristics of the target hardware (instruction timing, memory access times, etc.), the GA test strategy, like the other dynamic test-generation methods, is independent from the used programming language and compiler. To adapt it to a specific target hardware, a relatively small number of changes is needed. The test driver has to be ported to the new target and the table of the GA that describes the basic data types has to be adapted for the new target environment. The rest of the software remains unchanged.

4 Experiments

We carried out extensive simulations to demonstrate that the GA-based test approach is well-suited for testing WCET bounds. The simulations replaced the target machine part of the original MC68000 test setup, i.e., the tasks were executed in a simulation environment on a workstation.

By using a simulation environment we could focus our observations on that part of the task execution times that are not constant for every execution (e.g., for the matrix multiplication task, see below, we concentrated on the execution time differences caused by the variable number of steps of multiply instructions). Thus, we did not have to carry the constant parts of the task execution time through our evaluations, which simplified the evaluations. Only for the program with the most complex structure (Heapsort) we measured the execution times in CPU clock cycles.

4.1 Test Techniques

We evaluated the results of the GA-based test method against the results of three other methods for the assessment of execution times.

Random Testing. The GA for test data generation is replaced by a random data generator. Like GA-based testing, random testing is a black box test method. Random testing starts with the same knowledge about the program under test as GA-controlled testing, i.e., it needs the same information from the user (task interface description and termination criterion). It therefore serves as a kind of benchmark for the GA-based test strategy.

Best Effort Data Generation. This step, which was conducted before we started the GA-based tests, aimed at finding the real worst-case execution time for every program configuration under test. Best effort data generation is not a test generation strategy, but comprises the sum of our efforts to generate worst-case input data. Depending on the complexity of the task under test we used different methods, like complete or partial systematic test data generation, data generation based on code inspection, and trial and error for execution time assessment. Yet, we could not be sure that the results of the best effort analysis really yielded the WCET of the tasks (see below).

Static WCET Analysis. The GA-based test method has been devised to test the results of static WCET analysis. Therefore we had, of course, to perform static WCET analysis for each task [22, 20]. The results of static WCET analysis serve as further reference values for judging the quality of the GA-based test method.

Since both GA-based Testing and Random Testing use stochastic methods for data generation, the results of single tests with these test approaches are of limited value. To reduce this stochastic uncertainty, we evaluated each test configuration ten times for the GA-based and the random test.

4.2 Algorithms

We used seven programs with diverse execution-time characteristics for our evaluations. To get a broad range of different problem complexities for the GA, we tested the programs with various input spaces. The programs were executed in a number of configurations with diverse numbers of input elements and with different value ranges for the individual input elements.

Heapsort. This test program implements the `Heapsort` algorithm. The input to the program consists of n integers that have to be sorted by the algorithm. We tested `Heapsort` for three different array sizes, with n being 10, 25, and 50, respectively. For all configurations the input data items were in the range $[0..5n)$.

Matmul. `Matmul`[2] multiplies two matrices. Its input data space consists of two n by n integer matrices, i.e., $2 * n^2$ data elements. `Matmul` was tested with the following array sizes and value ranges for the array elements:

	array size	*value range*
1	5×5	0..255
2	5×5	0..65535
3	10×10	0..255

[2] The use of `Matmul` and `Matcnt` was inspired by the use of equally-named programs in [6]. The programs used in our work have a similar functionality as the programs in [6]. They are, however, not identical with the programs documented there.

Matcnt. `Matcnt` counts the number of elements of a matrix that have values in the upper half of the given value range. `Matcnt` assumes one n by n matrix as input. The configurations tested for `Matcnt` were the same as for `Matmul`.

DES. DES (data encryption standard) is an implementation of the algorithm that UNIX systems use for password encoding [1]. DES has two input parameters, an eight-character password and a two-character "salt"-string.

SimXT1, SimXT2, SimXT3. The three *SimXT* programs were implemented to test how the GA behaves for programs with "malignant" execution-time characteristics. The three algorithms have similar execution patterns. Their execution times depend on two input parameters, x and y, as follows: $exec\_time = C1 * ones(x) + C2 * x + C3 * ones(y) + C4 * y$, where the function *ones* counts the number of ones in the binary representation of its argument and $C1$ to $C4$ are constants that differ between *SimXT*-versions. The constants $C1$ to $C4$ and the ranges for the input data were chosen such that the configurations yielded problems of diverse difficulty for the test-generation GA.

	$C1$	$C2$	$C3$	$C4$
SimXT1	15	2	10	1
SimXT2	3	5	4	1
SimXT3	1	2	1	4

For each of the three simulation programs the input value ranges were [0..99] in configuration 1 and [0..129] in configuration 2.

As discussed in Section 3 the maximum number of individuals has to be defined for each test run. This termination criterion heavily influences the duration of the test and the probability that a faulty WCET bound can be discovered. Table 2 lists the number of individuals used as stopping criterion in the experiments.

Table 2. Maximum number of individuals applied as stopping criterion

Program	No. of Individuals
Heapsort	20000000
Matmul, Matcnt	4000000
DES	100000
SimXT1, SimXT2, SimXT3	10000

4.3 Results

Table 3 lists the results of our experiments. Columns 1 and 2 display the names of the tested programs and the configuration numbers. Columns 3 to 5 show the results of the GA-based tests. *Max.* is the global maximum of the execution times measured in all tests, *Avg.* is the average of the maxima measured in the ten test series, and *StdDev.* is the standard deviation of these maxima. Columns 6 and 7 display the results of the random tests. The meanings of *Max.* and *Avg.* are the same as for the GA-based tests. Columns 8 and 9 present the results from the best-effort data generation and from static WCET analysis.

4.3.1 Discussion of Results

Size of the Input Space: The GA-based method yields good WCET-estimates even for input-data spaces that are large compared to the number of tested individuals. This is best illustrated by the results of `Matcnt` and `Heapsort`. For `Matcnt` the GA returned the real WCET in all configurations, and even for `Heapsort` with 50 elements the average of the maxima derived by the GA-method underestimates the WCET bound from static analysis only by approximately 5%. Of further interest are the results of `Matmul`. The results suggest that the size of the input space of configuration 1 is at the limit of the GA's capabilities for obtaining the WCET. While the maximum and average result of configuration 1 are within 0.1% of the real WCET, the other configurations are much farther away from the optimum. Better results could be obtained only by drastically increasing the number of tested individuals.

GA-based Testing vs. Random Testing: For all programs except for *SimXT3*, GA-based testing performed equal or better than random testing (the results of the *SimXT* programs will be discussed in detail in a dedicated section below). The difference of the results of GA-based testing and random testing strongly depends on the relation of the size of the input space to the number of test-data sets generated in each test. If the number of test-data sets is large relative to the size of the input space, random testing yields similar results as GA-based testing (see configuration 1 of `Heapsort` and `Matcnt`, configurations 1 and 2). For large input-data spaces the goal-directed, GA-based approach clearly outperforms the random method (see `Heapsort`, configurations 2 and 3, and `Matmul`).

GA-based Testing and Static Analysis: The GA-based tests did not invalidate any of the WCET bounds from static analysis. Nevertheless, the fact that the results of the GA-based tests come very close to, or in many test cases hit the actual WCETs lets us assume

Table 3. Results of the execution-time tests. Except for `Heapsort` all numbers are given in abstract time units. For `Heapsort` the execution time is measured in CPU cycles.

		Results of GA Tests			Random		Best Eff.	WCET
Program	Conf.	Max.	Avg.	StdDev.	Max.	Avg.	Max.	Bound
Heapsort	1	10906	10906	0	10906	10906	10906	11610
Heapsort	2	41798	41554	282.5	37214	36554	41796	43320
Heapsort	3	106294	104307	2960.7	84396	83411	106963	109818
Matmul	1	1000	999	3.2	770	753	1000	1000
Matmul	2	1865	1853	13.4	1285	1259	2000	2000
Matmul	3	7880	7756	74.1	5340	5266	8000	8000
Matcnt	1	25	25	0	25	24	25	25
Matcnt	2	25	25	0	25	24	25	25
Matcnt	3	100	100	0	76	74	100	100
DES		62	60	1.7	59	56	62	63
SimXT1	1	438	438	0	438	433	438	438
SimXT1	2	559	559	0	559	553	559	559
SimXT2	1	632	631.6	1.26	632	630	632	632
SimXT2	2	817	816.2	2.53	817	813	817	817
SimXT3	1	608	601	1.89	608	606	608	608
SimXT3	2	785	782	1.41	785	783	785	785

that the GA-based method would detect an erroneous under-estimation of the WCET with high probability.

GA-based Testing vs. Best Effort Data Generation: As mentioned, the purpose of best-effort data generation was to complement static analysis and to approximate the real WCET of programs as good as possible. For all programs but `Heapsort` this method yielded the WCET and confirmed the result of static analysis. Comparing the best-effort and the GA-based method, the most impressive observation for us is definitely the result of `Heapsort`, configuration 2: The execution-time maximum measured in the GA-based tests was two CPU cycles greater than the execution time of the best-effort solution, i.e., the GA identified a better approximation to the real WCET than we had found in the extensive best-effort assessment.

Influence of Input Data on the GA Behavior

The quality of the test result of the GA-based approach largely depends on how the tested program uses its input data items. In order to understand this influence one has to know an important property of GAs. GAs allow a fast convergence towards optimal values—in this case large execution times—if different individuals yield different fitness values. If, on the other hand, a large number of individuals have the same fitness, the algorithm "tumbles" around, creating new individuals without increasing or even decreasing the fitness.

Of particular interest for the convergence is on the one hand the size of the input data space, while on the other hand one has to distinguish between programs that operate on the actual value of an input data item and programs that take into account the distribution of values. Programs with small input spaces that depend on the actual value of an input data item in general exhibit a fast convergence. For this type of program each set of input data out of a small range results in different fitness values. Hence the GA can quickly arrive at a high quality execution time bound. An example for a program of this category is `SimXT1`.

The opposite end of the range is marked by programs that have large input spaces and take into account mainly the input value distribution. In these cases many sets of input data out of a large space are alike and have the same or similar execution times. The GA is not guided into promising directions and takes a long time to explore the input space. An example for the latter category is `Heapsort`. If `Heapsort` sorts 10 numbers between 0 and 50 each, there are numerous test cases that yield approximately equal results. Consider, the case where the first nine number are 0, 1, 2, 3, 4, 5, 6, 7, and 8. In principle all test cases with the tenth input parameter in the range [9..50] are the same, yet they increase the input space.

4.3.2 Discussion of Malignant Test Scenarios

This section gives a closer look at the results for the *SimXT* programs. Although the principal structure of these programs is the same, the quality of the results derived by the GA differs: While the GA found the optimal solution in all tests for *SimXT1*, it failed to find the WCETs for a number of executions of *SimXT2* and *SimXT3* (see Table 3).

To get more details about the behavior of the GA for the three programs, we ran a further series of tests. This time we did not limit the maximum number of individuals tested, since we wanted to find out how many individuals were needed to obtain the WCET for each of the programs and configurations. Table 4 displays the results of the 10 tests for each configuration. The figures show the number of individuals the GA had to generate, on the average, until it found a solution (`Avg`) and the standard deviation of these numbers (`StdDev`).

Table 4. Number of individuals required for deriving the execution time bound

		No. of Evaluations	
Program	Conf.	Avg	StdDev
SimXT1	1	3050	2404.6
SimXT1	2	3411	2197.4
SimXT2	1	13638	22279.1
SimXT2	2	4104	3755.6
SimXT3	1	61306	64953.0
SimXT3	2	522910	530038.5

For `SimXT1` and the second version of `SimXT2` the GA needs a small number of evaluations to find the WCET. The first configuration of `SimXT2` as well as both configurations of `SimXT3`, however, require a large number of evaluations to arrive at the WCET. These numbers are even greater than the actual size of the input data space, which indicates that the GA has problems with these scenarios.

Table 5 shows the input data that yield the WCET for all configurations of the *SimXT* programs. First, note that for the configurations requiring a large number of evaluations the input values producing the WCET are at the upper bounds of the respective input spaces. Second, note that the worst-case input values are always slightly greater than a "big" power of 2 (129 = *128* + 1) or greater than a sum of "big" powers of 2 (99 = *64* + *32* + 3). A "big" power of 2 means that within the allowed input space this number is represented by one of the most significant bits.

Table 5. Input values that lead to the WCET

		Parameter Values	
Program	Conf.	x	y
SimXT1	1	95	95
SimXT1	2	127	127
SimXT2	1	*99*	95
SimXT2	2	127	127
SimXT3	1	*99*	*99*
SimXT3	2	127	*129*

The first observation implies that the GA cannot approach the solutions (99 and 129, respectively) from greater values, as greater values are outside the input space. The second observation indicates that a large number of bits has to be flipped in order to move from input values that are only little smaller than the worst-case inputs to the worst-case input values themselves. For example, seven bits have to be flipped to change the inputs from 127 to 129. Changes that affect a large number of bits at the same time are, however, very unlikely. Furthermore, the input values that have similar bit patterns as the worst-case input values yield significantly smaller execution times than the worst-case inputs. The input values that are composed of big powers of 2 form local maxima for the search of the GA that attract the GA and reduce the survival probability of input values with bit patterns similar to the worst-case patterns. Thus the GA gets stuck and takes long time to construct the worst-case inputs.

5 Summary and Conclusion

In this paper we introduced a black-box test strategy for testing the WCET bounds computed from static WCET analysis. This test strategy relies on a Genetic Algorithm (GA) for the goal-oriented generation of test cases: By means of its feedback mechanism the GA effectively guides test-data generation into directions that have a high chance to yield the real WCET of the program under test.

The experiments carried out to evaluate the proposed test method have shown that test data generation based on GAs is well suited for WCET tests. The results achieved by the GA-based approach compare favorably to execution time bounds derived from static WCET analysis and similar techniques and outperform random testing by far. In particular for programs with large input spaces the GA-based method creates test cases of significantly better quality than random testing, i.e., the execution times of the GA-generated test data are significantly closer to the WCET than the execution time of the random test data.

In the future we will further investigate into problems of GA-based testing that we identified in our experiments. We will focus on problems that the GA has with certain execution-time distributions of the input-data space and with the bit-coding of input data. A special "adaptive" mutation rate will be applied to overcome the problems of the GA with large plateaus of equal fitness function values. Since the tested task represents the fitness function and its execution time the fitness value, strict monotony of the function cannot be assumed. Furthermore, we plan to develop mechanisms that detect anomalies in the behavior of the GA and identify special problem cases.

Acknowledgements

We would like to thank Christian Ebner, Richard Gerber, our shepherd Michael Gonzalez Harbour, and the anonymous reviewers for their valuable comments on earlier versions of the paper.

References

[1] S. Garfinkel and G. Spafford. *Practical UNIX Security*. O'Reilly and Associates Inc., Sebastopol, CA, 1991.

[2] R. L. Glass. Real-time: The "lost world" of software debugging and testing. *Comm. of the ACM*, 23(5):264–271, May 1980.

[3] F. Glover, E. Taillard, and D. de Werra. User's guide to tabu search. *Annals of Operations Research*, 41, 1993.

[4] D. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

[5] H. Gomaa. Software development of real-time systems. *Communications of the ACM*, 29(7):657–668, July 1986.

[6] C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *Proc. 16th Real-Time Systems Symposium*, pages 288–297, Dec. 1995.

[7] C. R. Hill. A real-time microprocessor debugging technique. *ACM SIGSOFT Software Engineering Notes*, 8(4):145–148, Aug. 1983. Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symp. on High-Level Debugging.

[8] J. Holland. *Adaptation in Natural and Artificial Systems*. The Univ. of Michigan Press, Ann Arbor, 1975.

[9] B. Jones, H. Sthamer, and D. Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, 11(5):299–306, 1996.

[10] S. Kirkpatrick, C. Gelatt, and M. Veechi. Optimization by simulated annealing. *Science*, 220:671–680, May 1983.

[11] E. Kligerman and A. Stoyenko. Real-time euclid: A language for reliable real-time systems. *IEEE Transactions on Software Eng.*, SE-12(9):941–949, Sep. 1986.

[12] L. Ko, C. Healy, E. Ratliff, and M. Harmon. Supporting the specification and analysis of timing constraints. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pages 170–178, June 1996.

[13] H. Kopetz. *Real-Time Systems*. Kluwer, 1997.

[14] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, Aug. 1990.

[15] Y. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *Proc. 17th Real-Time Systems Symposium*, Dec. 1996.

[16] G. McGraw, C. Michael, and M. Schatz. Generating software test data by evolution. Technical Report RSTR-018-97-01, Reliable Software Technologies Corporation, 21515 Ridgetop Circle, Suite 250, Sterling, VA 20166, Feb. 1998.

[17] W. Miller and D. Spooner. Automatic generation of floating point test data. *IEEE Transactions on Software Engineering*, SE-2(3):223–226, Mar. 1976.

[18] F. Müller and J. Wegener. A comparison of static analysis and evolutionary testing for the verification of timing constraints. In *Proc. of the 4th Real-Time Technology and Applications Symposium*, pages 144–154, Denver, CO, USA, June 1998.

[19] R. Nossal and T. Galla. Solving NP-Complete Problems in Real-Time System Design by Multichromosome Genetic Algorithms. In *Proceedings of the SIGPLAN 1997 Workshop on Languages, Compilers, and Tools for Real-Time Systems*, pages 68–76. ACM SIGPLAN, June 1997.

[20] P. Puschner. Worst-Case Execution Time Analysis at Low Cost. *Control Engineering Practice*, 6(1):129–135, Jan. 1998.

[21] P. Puschner and C. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Real-Time Systems*, 1(2):159–176, Sep. 1989.

[22] P. Puschner and A. Schedl. Computing Maximum Task Execution Times — A Graph-Based Approach. *Real-Time Systems*, 13(1):67–91, July 1997.

[23] W. Schütz. *The Testability of Distributed Real-Time Systems*. Kluwer, Boston, MA, USA, 1993.

[24] M. Srinivas and L. Patnaik. Genetic Algorithms: A Survey. *IEEE Computer*, pages 17–26, June 1994.

[25] N. Tracey, J. Clark, and K. Mander. The way forward for unifying dynamic test case generation: The optimization-based approach. In *Proc. of the 1998 IFIP International Workshop on Dependable Computing and Its Applications*, pages 169–180, Johannesburg, South Africa, 1998.

[26] J. Wegener and M. Grochtmann. Verifying timing constraints of real-time systems by means of evolutionary testing. *Real-Time Systems*, to appear.

[27] J. Wegener, H. Sthamer, B. Jones, and D. Eyres. Testing real-time systems using genetic algorithms. *Software Quality Journal*, 6(2):127–135, June 1997.