Verifying Timing Constrains by means of Evolutionary Testing in a Nondeterministic Environment

J.Z.M. Broeders 1376934

Delft University of Technology,
EWI, Software Technology
J.Z.M.Broeders@student@tudelft.nl

Abstract. Genetic algorithms can be used to automatically find test cases to test the temporal requirements of real-time systems. There is a current trend to use "off the shelf" hardware and software to implement soft real-time systems. These off the shelf products are designed to optimize the average performance which causes nondeterministic behavior. Genetic algorithms normally used to find test cases for testing temporal requirements do not work in a nondeterministic environment. This problem is identified and explained in this paper. Several solutions are proposed and tested. It is concluded that recalculating the fitness, for each individual, for each new generation is the best method when searching for a test case which will produce the WCET for a piece of code under test. The effect of a nondeterministic environment in the automatic search for temporal test cases is only studied for one algorithm (insertion sort) in one specific environment (Pentium 4 with Windows XP). More work is needed to verify the proposed solution for other algorithms and other environments.

1. Introduction

Real-time systems are defined as those systems in which the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are produced [1]. Real-time systems which are useless if a temporal requirement is not met, are called hard real-time systems. There are also a lot of systems with temporal requirements which may be missed, ones in a while. These systems are called soft real-time systems. A typical soft real-time system is a DVD player. Each video frame should be produced within a certain time period but when the DVD player fails to produce the frame on time in exceptional cases most users probably will not notice it.

For every real-time system temporal requirements are defined, either implicitly or explicitly. These temporal requirements must be verified to ensure proper functioning of the real-time system. Outputs should not be produced too early or too late. The execution time of a piece of code often depends on the input values which the code has to process. For these pieces of code with an input dependable execution time it is important to find the input data which causes extreme execution times. Test inputs which cause extreme (best case or worst case) execution times for a certain task in the system are good test cases, because they most likely cause the system to fail a temporal requirement. Therefore it is important to find the test input which produces the best case execution time (BCET) and the test input which produces the worst case execution time (WCET) for each task with temporal requirements in the system.

There are two main methods to find test cases: manual and automatic. In manual testing the tester uses static timing analysis to predict the test inputs which result in the B/WCET for a certain piece of code. When performing static timing analysis the tester takes the code, analyzes the set of possible control flow paths through the code, combine control flow with some (abstract) model of the hardware architecture, and obtains bounds for this combination [2]. This method is very labour intensive and when modern hardware is used the behaviour of this hardware is very difficult or even impossible to predict. Especially for processors which use speculative execution [4] the execution time of a piece of code can become nondeterministic [5]. In automatic testing the test inputs are produced automatically by means of some

algorithm and these test cases are compared using dynamic timing analysis. When this method is used the piece of code under test is executed on the target hardware for each generated test input and the B/WCET is measured. These measurements are used to select the input data with the B/WCET execution time.

A recent (March 2007) overview of methods and tools to find the B/WCET states: “Today, in most parts of industry the common method to estimate execution time bounds is to measure the end-to-end execution time of the task for a subset of the possible executions—test cases. These will in general overestimate the BCET and underestimate the WCET and so are not safe for hard real-time systems.”

The task of automatic testing is to automatically find the input values with especially long or short execution times in order to check whether a temporal error can be produced. The most simple form of automatic testing is random testing. In this case the test cases are generated at random. Of course general search or optimization techniques can be used to improve the randomly found test-cases. Due to selection and loop statements in programs the execution time of such programs is dependent on the input data in a complex, discontinuous, and non-linear manner. When searching for extreme execution times neighborhood search methods like hill climbing can not be used due to the complex search space. Therefore, metaheuristic search methods are used, for example: evolutionary algorithms, simulated annealing or tabu search. In this paper, genetic algorithms [6] are used to generate test data, since their robustness and suitability for this task has already been proven [7][8][9].

General purpose operating systems (GPOSes) like Linux or Microsoft Windows XP that are not really deterministic at all are nevertheless employed in many embedded applications [10]. Many of these nondeterministic systems have (soft) real-time requirements which must be verified.

This paper signals a problem that can occur when genetic algorithms (GAs) are used to search for data to produce worst case execution times in a nondeterministic environment. The nondeterminism can be caused by hardware (e.g. speculative execution), by software (e.g. GPOS), or both. Three solutions for this problem are presented and these solutions are demonstrated for a simple algorithm.

The rest of this paper is organized as follows. In the next (second) section a short overview of the use of GAs to find test cases with the WCET is given. In the third section the problem that occurs when a GA is used in a nondeterministic environment is identified and illustrated with some measurements. Also some solutions are presented in that section. In the fourth section experiments are presented which show how well the presented solutions work in a nondeterministic environment for a simple algorithm which is known to have a high testability by GAs in a deterministic environment. In the fifth section related work is discussed. Finally, in the last section, some conclusions are drawn based on the performed measurements and further research ideas are mentioned.

2. Using Genetic Algorithms to Find Worst Case Execution Times

Genetic algorithms are based loosely on the ideas of biological genetics. They are applied to the problem of automatically finding input data which cause the WCET for a piece of code under test in the following way. A number of guessed solutions are generated by creating input data at random. Each solution (input data) is interpreted as an individual in the GA. The input data is interpreted as a single bit pattern called chromosome and the execution time of the piece of code for that specific input data determines the fitness of the individual. When searching for WCET, individuals which results in a long execution time get a high fitness value and individuals which results in a short execution time get a low fitness value. Pairs of individuals (called parents) are chosen by some selection strategy based on their fitness and their chromosomes are combined in some way to produce new individuals (children) an

analogous way to biological reproduction. Unfortunately, GAs are not guaranteed to generate improved individuals, but the so called schema theorem [6] predicts that the probability of improving over several generations is high.

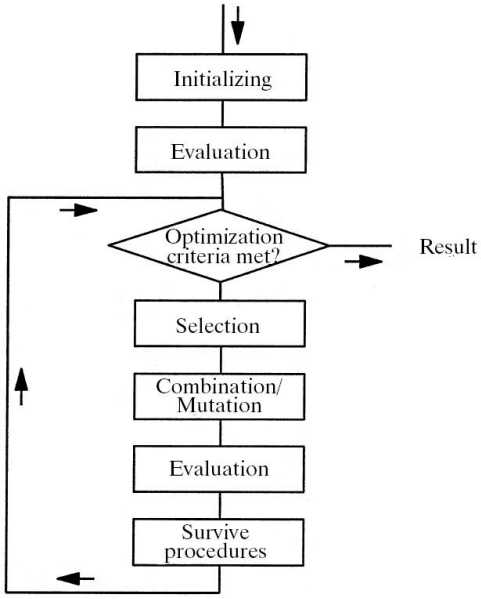

Figure 1. The flow diagram of a typical GA.

Figure 1 which is taken from [7] gives an overview of a typical GA. Each individual in the population is evaluated by calculating its fitness. The remainder of the algorithm is iterated until the optimum is achieved or, when the optimum is not known, until some other stopping criterion is met. Pairs of individuals are selected from the population and new individuals are generated by combining and mutating their chromosomes. The new individuals are evaluated for their fitness, and survivors into the next generation are chosen from the parents and offspring, according to fitness. It is important to maintain a diversity in the population, though, to prevent premature convergence to a suboptimal solution. To understand this paper it is not so important to understand the details of selection, combination and mutation. In the experiments in this paper a population consisting of 20 parents and 20 children is used. Selection is performed with a tournament selector with a tournament size of 3. Combination is performed with multipoint crossover with a probability of 0.5 and mutation is implemented by a bit flip with a probability of 0.01. The interested reader can find more information in [6]. It is however very important to understand the evaluation method and the survival method to understand the problem described in the next section of this paper.

The evaluation method executes a so called fitness function to determine the fitness of a *new* individual. When we use GAs to search for the WCET the measured execution time can simply be used as fitness value. The fitness function which is called with the chromosome as parameter simply executes the piece of code under test with the chromosome as input data and measures the execution time to evaluate the fitness. Preserving good parents is called “elitism” [11] because the elite (a select group of individuals with outstanding fitness) stays alive for a long time.

3. Problems with Using Genetic Algorithms to Find the WCET in a Nondeterministic Environment

It is important to note that the WCET of a piece of code does not include waiting times due to preemption, blocking, or other interference [2]. The WCET measurement must be done in a deterministic environment. When the execution time is measured several times using the same input data the measured execution time must be the same for each measurement. When measuring the execution time with constant input data in a nondeterministic environment the measured execution time will vary. When a nondeterministic GPOS is used (e.g. Windows XP) the execution of the code under test can be interrupted by other tasks which causes great variation in observed execution times. In figure 2 the execution time for a simple insertion sort with a random input of 1024 elements is given for 1000 measurements. The minimal measured value is 1.269 ms and as can be seen from figure 2 most measurements are close to this minimum value. The maximum measured value is 2.838 ms, which is more than 2 times the minimal value and this maximum can only be causes by the operating system which must have blocked the task under test for some time to execute some other task.

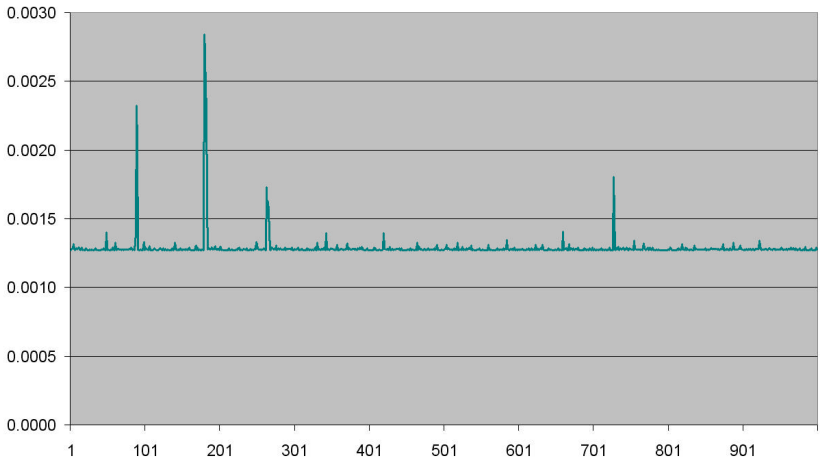

Figure 2. The execution time of insertion sort in second measured 1000 times.

The problem when using a GA to find the WCET in a nondeterministic environment is that some individuals get a very high fitness value which has nothing to do with their actual fitness but is caused by a measurement error due to the nondeterministic behavior of the environment. When elitism is used, these individuals stay alive for ever and cause the search process to stay stuck at this erroneous measured executing time. Figure 3 shows the best WCET of each generation found by a GA for 1000 generations. This seems like a very good result but when the best input data found is repeatedly tested for 1000 tests the average measured execution time is 1.374 ms which is very close to the average value of a 1000 random tests which is 1.287 ms. The “real” execution time of the found input data is far away from the maximum value shown in figure 3: 3.438 ms. This maximum is clearly caused by the nondeterministic environment and it prevents the GA to further explore the search space and find better test cases.

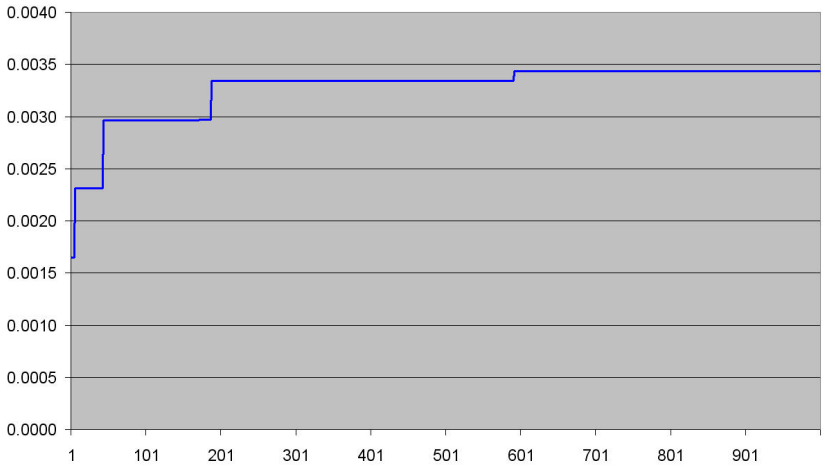

Figure 3. The WCET of insertion sort for 1000 generations of a GA.

The following solutions to this problem are proposed:

- Instead of measuring the execution time of each new individual ones, the execution time can be measured several times for each new individual. The minimal measured value is then used as fitness value because the other, longer execution times, must be caused by the nondeterministic environment. The time to determine the fitness of a new individual, which is the time to measure the execution time of the piece of code under test using the new individual's chromosome as input data, is of course N times higher if we take N samples.
- Instead of only measuring the execution time of each new individual, the execution time of the surviving parents is also measured again. Using this method the elite is formed by the individuals which proof their fitness with each generation. The time to determine the fitness of a new generation normally is the time it takes to determine the fitness of all new individuals (children). Using this method the time to determine the fitness of a new generation is the time it takes to determine the fitness of all individuals (parents + children). If the number of children equals the number of parents, the time to determine the fitness of all individuals in a generation is twice the time needed to determine the fitness of the new individuals in a generation.
- Instead of preserving good parents into the next generation (elitism), the parents can be killed so each generation only consists of newly created children. Using this method individuals with an erroneous measured execution time are not preserved in the population. This method does not influence the time which is needed to determine the fitness in the GA.

4. Measuring the WCET of Insertion Sort in a Nondeterministic Environment

To test the solutions proposed some measurements are performed. Gross [12] identifies a number of properties of programs which determine the testability of the programs by GAs. Programs which contain execution paths which only execute for a specific input value are difficult to test using GAs because the probability of finding this single input value is low. We used a program, insertion sort, which does not contain such paths and has a good testability for GAs according to Gross.

The environment to perform the tests is a PC with an Intel Pentium 4 microprocessor with a clock frequency of 3.2 GHz running the Windows XP operating system. The Intel Pentium 4 uses so called hyper- threading [13] which means that the instruction scheduler, which issues instructions out of order,

intermingles instructions from several threads to create on optimal instruction mix for the best average execution time. This makes the execution time of one specific piece of code nondeterministic because it depends on the other (hyper-threaded) tasks which run at the same time. The execution time of a task running on the Windows XP operating system [14] is nondeterministic because the programmer can not prevent the operating system to block the task under test.

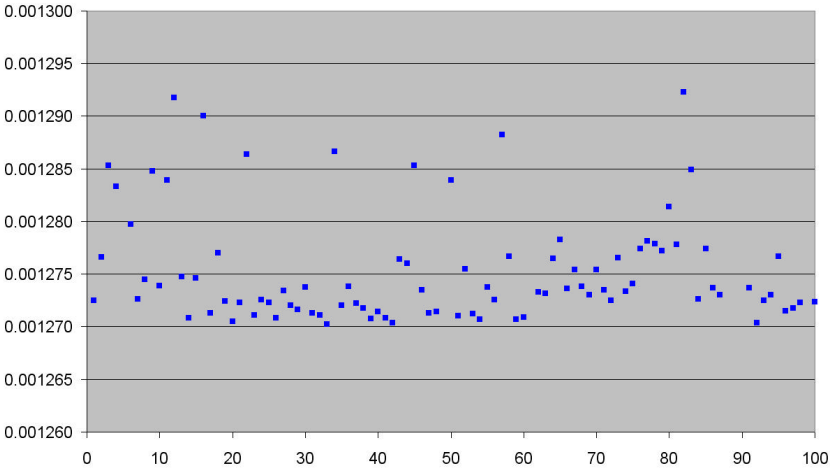

Figure 4. Zoomed in view of figure 2.

The Win32 API provides a high-resolution timer which (on the test system) has the same resolution as the processor clock frequency. This high-resolution timer is used to create figure 2. Figure 4 zooms in on the first 100 measurements and shows that 60% of the measurements are within a boundary of +/- 2.5 µs and 80% are within a boundary of +/- 5 µs. There are only 6 values which are above the border of figure 4 and the highest value has an error of 1050 µs.

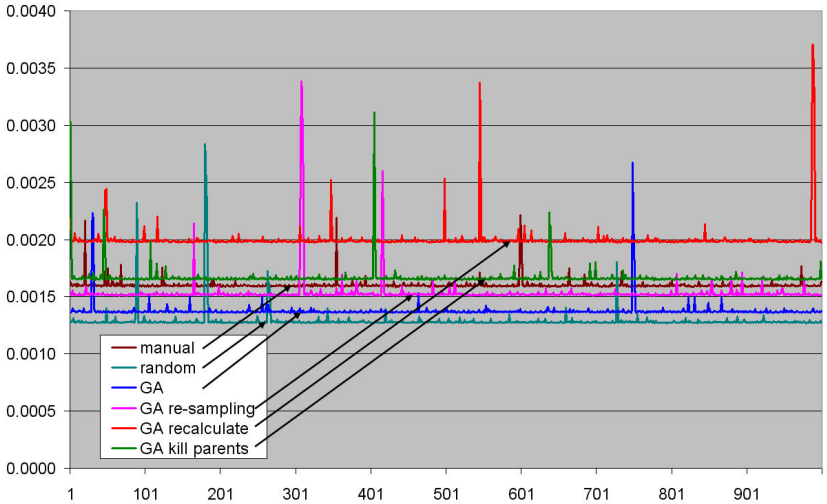

Figure 5. The WCET of insertion sort for different test cases each executed 1000 times.

For each proposed solution the GA to search for the WCET is used to generate 1000 generations starting from a random population. The best individual of the last generation is saved to a file. This file is then used as input data for the test program and the execution time is measured 1000 times (see figure 5). For comparison the execution time of manually determined WCET input data and the execution time of random input data are also shown in figure 5. From a quick static analysis it was concluded that input data which is in the reversed order, will cause the WCET for the insertion sort algorithm. For each generation generated by the GA, the fitness value of the fittest individual is recorded, see figure 6.

Figure 5 shows that all automatically found test cases and the manually found test case all perform better than the random test case. As already explained, the GA normally used for finding the WCET only finds a small improvement compared to the random test case. The test case found by re-sampling every new individual 3 times is the least effective of the three proposed solutions. The test case found by killing the parents after each new generation is generated is slightly better than the manual test case. The test case found by re-sampling all parents each generation clearly is the best test case. Figure 6 can be used to explain this.

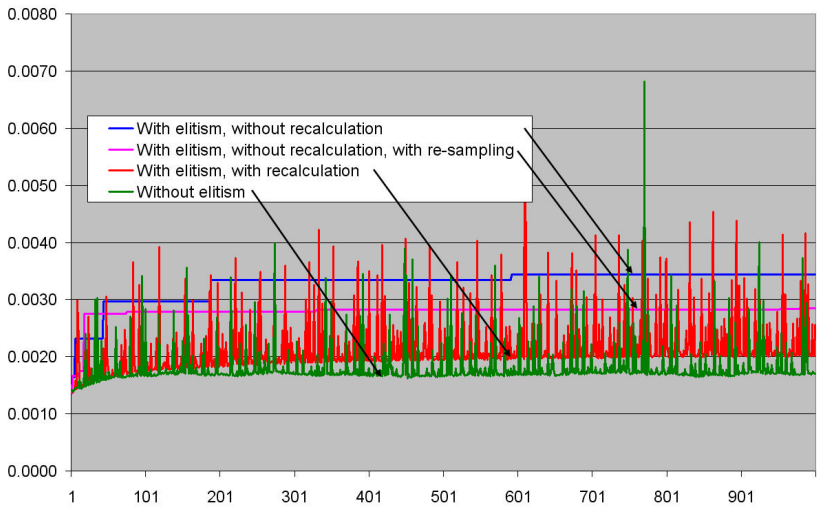

Figure 6. WCET of insertion sort for 1000 generations of different GAs.

Figure 6 shows that the re-sampling solutions suffers from the same problem as the original GA with elitism. Closer examination of the measurements shown in figure 2 shows that the big measurement errors occur in clusters. This means we can not use the re-sample solution with a small re-sample factor. Using the re-sample solution with a high re-sample factor, e.g. 10, is not practical because the GA with re-sampling N times will take about N times as long as a GA without re-sampling. The re-sampling solution is the most expensive solution but produces the least quality test case. The solution which does not use elitism (by killing the parents) follows the solution which recalculates all fitness values for each generations for the first 50 generations, but after about 100 generations, the solution without elitism fails to find better test cases and just wanders around in the solution space. This is probably caused by the inability to remember the best solutions found in the current generation into the next generation. The solution with elitism which calculates all solutions for each generation keeps finding better test cases.

The atomically found test cases with the last two proposed solutions are better than the manual test case. This can be explained as follows. The quick static analysis done to find the input data which is thought to produce the WCET only did a path analysis. This analysis revealed that the insertion sort has

to execute the most instructions when the data is in the reversed order. The static analysis failed to take the specific target platform into consideration. The Pentium 4 uses branch prediction [4][13] and speculatively executes instructions in the predicted direction of the branch. The branch predictor can predict the behavior of the conditions used in the insertion sort very well when the data is in the reversed order. The GA finds test cases which are a lot more difficult to predict. The long pipeline used in the Pentium 4 causes a long branch penalty (extra time needed to recover from a wrong prediction). To run the found test cases less instructions have to be executed than for the manual test case but due to branch penalties the execution of lesser instructions takes more time. This is an example of speculation-caused timing anomalies [17][18].

5. Related Work

A survey of search based software test data generation can be found in [19]. Wegener et al. [20] were the first to experiment with GA to find the WCET for a piece of code. In their experiments [7][21] they used a deterministic method to determine the fitness of an individual. They did not measure the execution time, but instead determined the fitness by establishing the number of cock cycles needed to execute the code fragment. This later value was obtained by using an Object Code Insertion (OCI) technology to count the instructions executed by the software under test. They explicitly state: “Thus the execution times reported were the same for repeated runs with the same input parameters since it was unaffected by overheads such as periodic interrupts by the operating system”. Their experiments show that GAs yield better results than random testing. Also in some cases GAs found better results than manual testing. These results where confirmed by experiments performed by Puschner et al. [8] and Gross [9].

Gross [9] used GAs to determine the WCET of 15 different pieces of code and compared the result to the WCET found by random testing. The fitness function actually measures the execution time of the piece of code under test. He used a real-time operating system and assigns the task which executes the fitness function the highest priority. This causes the measured execution times to be reasonable deterministic. Gross reports: “Repeated timing of the same test case almost always leads to the same execution time in microseconds (+/- 2 microseconds)”. The phrase “almost always” may worry us a little though.

In [22] a GA is used to find extreme response times of protection relay software. The authors make the following remark when discussing the fitness function: “At first, old individuals were not tested again; but later they were. Because of the nondeterminism of the testing system, the tested individuals do not have the same fitness values every time.”

There is a large body of knowledge about using GAs in a noisy environment [23][24][25]. See Beyer [26] for a review of techniques for coping with noisy fitness evaluations. Di Pietro et al. [25] state: “Re-sampling and averaging the fitness of several samples is the noise compensation technique that is usually used by the optimization community (if anything)”. Using this technique each individual is sampled several times and the thereby obtained noisy fitness values are averaged. All research found about noisy fitness functions assumes that the noise is normally distributed with a mean of zero and standard derivation equal to the noise strength. The “noise” caused by the nondeterministic behavior of the environment investigated in this paper has a positive bias and therefore the re-sampling technique did not work.

Table 1. Relative WCET for different test cases for insertion sort.

Author	Random test case	Test case found with GA	Manually found test case
Gross	100%	148%	162%
Broeders	100%	156%	125%

Gross [9] also tested an insertion sort program. In table 1 his results are compared to the results found in this paper. To compare the results all execution times are expressed relatively to the WCET time found by random testing. The table shows that the results found in this paper for a nondeterministic environment are compatible with results found in a deterministic environment. The fact that the manually found test case performs better in the measurements performed by Gross can be explained by the fact that Gross used a Pentium 3 processor which uses less aggressive speculative execution than the Pentium 4 which is used for the measurements in this paper.

6. Conclusions and Further Work

From the described measurements we can conclude that GAs can be used to automatically generate test cases which produce extreme execution times for an insertion sort program in a nondeterministic environment. When searching for a test case which produces the WCET it is important to recalculate the fitness of all individuals for every generation. This recalculation is needed to prevent an erroneous measured execution time to dominate the search.

Of course these results can not be generalized. This paper only investigated the effect of nondeterministic fitness measurements for one algorithm (insertion sort) in one specific environment (Pentium 4 with Windows XP). More work is needed to verify the proposed solution for other algorithms and other environments.

There is a current trend to use “off the shelf” hardware and software to implement soft real-time systems. These “off the shelf” products are designed to optimize the average performance which causes nondeterministic behavior. The temporal requirements for these systems must be tested and test cases which cause extreme execution times must be found. Finding these test cases manually is very time consuming and very difficult therefore it is important to investigate if the technique described in this paper can be used to automatically generate test cases for such soft real-time systems.

References

[1] J. A. Stankovic. *Real-Time Computing.* Byte, pages 155--160, August 1992.

[2] P. Puschner and A. Burns. *A review of worst-case execution-time analysis*. Journal of Real-Time Systems, 18(2/3):115--128, May 2000.

[3] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesin, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Muller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. *The Worst-Case Execution Time Problem - Overview of Methods and Survey of Tools*. MRTC report ISSN 1404-3041 ISRN MDH-MRTC-209/2007-1-SE, March 2007.

[4] J. L. Hennessy and D. A Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[5] Jan Reineke, Björn Wachter, Stefan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. *A Definition and Classification of Timing Anomalies.* 6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

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

[7] Joachim Wegener, Harmen Sthamer, Bryan F. Jones, and David E. Eyres. *Testing real-time systems using genetic algorithms.* Software Quality Journal, 6(2), pp. 127-135, 1997.

[8] Peter P. Puschner and Roman Nossal. *Testing the Results of Static Worst-Case Execution-Time Analysis.* IEEE Real-Time Systems Symposium, pp. 134-143, 1998.

[9] H.-G. Gross. *An Evaluation of Dynamic, Optimization-based Worst-Case Execution Time Analysis.* Proceedings of the International Conference on Information Technology: prospects and Challenges in the 21st Century May23-26, 2003.

[10] Jason McDonald. *Selecting an Embedded RTOS.* eg3.com, June 2007.

[11] Randy I Haupt and Sue Ellen Haupt. *Practical Genetic Algorithms 2nd edition*. John Wiley and Sons, 2004.

[12] H.-G. Gross. *A prediction system for evolutionary testability applied to dynamic execution time analysis.*
Information & Software Technology, 43(14), pp. 855-862, 2001.

[13] http://www.intel.com/products/processor/pentium4/index.htm

[14] http://www.microsoft.com/windows/products/windowsxp/default.mspx

[15] http://msdn2.microsoft.com/en-us/library/ms644900(VS.85).aspx

[16] http://blackboard.tudelft.nl Course: IN4024 Real-Time Systems (2007-2008 Q2) (7121-070802) Additional documents for exercise 5.

[17] Thomas Lundqvist and Per Stenström. *Timing Anomalies in Dynamically Scheduled Microprocessors.* IEEE Real-Time Systems Symposium, pp. 12-21, 1999.

[18] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. *The influence of processor architecture on the design and the results of WCET tools*. Proceedings of the IEEE, 91(7), pp. 1038-1054, 2003.

[19] Phil McMinn. *Search-based software test data generation: a survey*. Softw. Test, Verif. Reliab, 14(2), pp. 105-156, 2004.

[20] J. Wegener, K. Grimm, M. Grochtmann, H. Sthamer, and B. Jones: *Systematic Testing of Real-Time Systems.* Proceedings of the 4th European Conference on Software Testing, Analysis & Review (EuroSTAR 1996), Amsterdam, Netherlands, December 1996.

[21] Joachim Wegener and Matthias Grochtmann, *Verifying Timing Constraints of Real-Time Systems by Means of Evolutionary Testing*. Real-Time Systems, 15(3), pp. 275-298, 1998.

[22] J.T. Alander, T. Mantere, G. Moghadampour, and J. Matila. *Searching protection relay response time extremes using geneticalgorithm-software quality by optimization.* Fourth International Conference on Advances in Power System Control, Operation and Management, pp. 95 - 99 vol.1 Nov 1997.

[23] J. Michael Fitzpatrick and John J. Greffenstette. *Genetic Algorithms in Noisy Environments.* Machine Learning, Vol. 3, pp. 101-120, 1988.

[24] B. L. Miller and D. E. Goldberg, *Genetic Algorithms, Tournament Selection, and the Effects of Noise*. Complex Systems, pp. 193-212, June 1995.

[25] Anthony Di Pietro, Lyndon While and Luigi Barone. *Applying Evolutionary Algorithms to Problems with Noisy, Time-consuming Fitness Functions*. Proceedings of the 2004 IEEE Congress on Evolutionary Computation, pp. 1254-1261, IEEE Press, 20-23 June 2004.

[26] H.-G. Beyer. *Evolutionary algorithms in noisy environments. Theoretical issues and guidelines for practice.* Computer Methods in Applied Mechanics and Engineering 186(2-4), 239–267, 2000.