A Comparison of Static Analysis and Evolutionary Testing for the Verification of Timing Constraints

Frank Mueller
Humboldt-Universität zu Berlin
Institut für Informatik
Unter den Linden 6
10099 Berlin (Germany)
phone: (+49) (30) 20181-276, fax:-280
e-mail: mueller@informatik.hu-berlin.de

Joachim Wegener
Software Technology (FT3/SM)
Daimler-Benz AG
Alt-Moabit 96a
10559 Berlin, Germany
phone: (+49) (30) 39982-232, fax:-107
Joachim.Wegener@dbag.bln.daimlerbenz.com

Abstract

This paper contrasts two methods to verify timing constraints of real-time applications. The method of static analysis predicts the worst-case and best-case execution times of a task's code by analyzing execution paths and simulating processor characteristics without ever executing the program or requiring the program's input. Evolutionary testing is an iterative testing procedure, which approximates the extreme execution times within several generations. By executing the test object dynamically and measuring the execution times the inputs are guided yielding gradually tighter predictions of the extreme execution times. We examined both approaches on a number of real world examples. The results show that static analysis and evolutionary testing are complementary methods, which together provide upper and lower bounds for both worst-case and best-case execution times.

1. Introduction

For real-time systems the correct system functionality depends on their logical correctness as well as on their temporal correctness. Accordingly, the verification of the temporal behavior is an important activity for the development of real-time systems.

The temporal behavior is generally examined by performing a schedulability analysis to ensure that a task's execution can finish within specified deadlines. The models for schedulability analysis are commonly based on the assumption that the worst-case execution time (WCET) is known. Specifically, the models assume that the WCET must not exceed the task's deadline. The best-case execution time (BCET) may also be used to predict system utilization or ensure that minimum sampling intervals are met.

Techniques of static analysis (SA) can be used in the course of system design in order to assess the execution times of planned tasks as pre-condition for schedulability analysis. Static timing analysis constitutes an analytical method to determine bounds on the WCET and BCET of an application. SA simulates the timing behavior at a cycle level for hardware concepts such as caches and pipelines of a given processor. The approach discussed in this paper uses the method of Static Cache Simulation followed by Path Analysis within a timing analyzer. Timing estimates are calculated without knowledge of the input and without executing the actual application.

Dynamic testing is one of the most important analytical method for assuring the quality of real-time systems. It serves for the verification as well as the validation of systems. An investigation of existing test methods shows that they mostly concentrate on testing the logical correctness. There is a lack of support for testing the temporal system behavior. For that reason, we developed a new approach to test the temporal behavior of real-time systems: Evolutionary Testing (ET). ET searches automatically for test data, which produces extreme execution times in order to check if the timing constraints specified for the system are violated. This search is performed by means of evolutionary computation.

Although SA and ET are usually applied in different phases of system development, both procedures aim at estimating the shortest and longest execution times for a system, which makes a comparison of these two methods very interesting. Both approaches are compared in this paper with the help of several examples.

Chapter 2 offers a general overview of related work on SA as well as on testing. The third chapter describes the tool we employ for SA. Afterwards, chapter 4 introduces

ET. Both approaches have been used to determine the minimum and maximum run times of different systems. Chapter 5 summarizes the obtained results. These are discussed in chapter 6. It will be seen that a combination of SA and ET makes a reliable definition of extreme run times possible. The most important statements are summarized in chapter 7 that also includes a short outlook on future work.

2. Related Work

This section presents an overview of published work in timing analysis for real-time systems followed by a discussion of previous work on testing methods for real-time systems.

2.1. Timing Analysis of Real-Time Systems

Bounding the WCET of programs is a difficult task. Due to the undecidability of the halting problem, static WCET analysis is subject to constraints on the use of programming language constructs and on the underlying operating system. For instance, an upper bound on the number of loop iterations has to be known, indirect calls should not be used, and memory should not be allocated dynamically [25]. Often, recursive functions are also not allowed, although there exist outlines on treating bounded recursion similar to bounded loops [19]. Recent research in the area of predicting the WCET of programs has made a number of advances. Conventional methods for static analysis have been extended from unoptimized programs on simple CISC processors to optimized programs on pipelined RISC processors, and from uncached architectures to instruction caches [1, 16] and data caches [13, 16, 31].

Today, mainly three fundamental models for static timing analysis exist. First, a source-level oriented timing schema propagates times through a tree and handles pipelined RISC processors with first-level split caches [23, 13]. Second, a constraint-based method models architectural aspects, including caches, *via* integer linear programming [16]. Third, our approach uses data-flow analysis to model the cache behavior separate from pipeline simulation, which is handled later in a timing analyzer via path analysis [1, 11, 31]. The first and second approaches use integrated analysis of caches while our approach uses separate analysis. This allows us to deal with multi-level memory hierarchies or unified caches. Another approach using data-flow analysis to modeling caching originally used the same categorizations as our approach but a different data-flow model. Recently, the approach has been generalized to handle a number of data-flow solutions with differing complexity and accuracy [9].

In the presence of caches, non-preemptive scheduling was initially assumed to prevent undeterministic behavior due to the absence of unpredictable context switch points. If context switches occurred at arbitrary points (*e.g.,* in a preemptive system), cache invalidations may occur resulting in unexpected cache misses when the execution of a task is resumed later on. Hardware and software approaches have been proposed to counter this problem but find little use in practice due to a loss of cache performance when caches are partitioned [14, 17]. Recently, attempts have been made to incorporate caching into rate-monotone analysis and response-time analysis [5, 15], which allows WCET predictions for non-preemptive systems to be used in the analysis of preemptively scheduled systems. This approach seems most promising since the information gathered for static timing analysis can be utilized within this extended framework for schedulability analysis.

2.2. Testing Real-Time Systems

Analytical quality assurance plays an important role in ensuring the reliability and correctness of real-time systems, since a number of shortcomings still exist within the development life cycle. In practice, dynamic testing is the most important analytical method for assuring the quality of real-time systems. It is the only method that examines the run-time behavior, based on an execution in the application environment. For embedded systems, testing typically consumes 50% of the overall development effort and budget [8, 29]. It is one of the most complex and time-consuming activities within the development of real-time systems [12]. In comparison with conventional software systems the examination of additional requirements like timeliness, simultaneity, and predictability make the test costly, and technical characteristics like the development in host-target environments, the strong connection with the system environment or the frequent use of parallelism, distribution, and fault-tolerance mechanisms complicate the test.

The aim of testing is to find existing errors in a system and to create confidence in the system's correct behavior by executing the test object with selected inputs. For testing real-time systems, the logical system behavior, as well as the temporal behavior of the systems, need to be examined thoroughly. An investigation of existing test methods shows that a number of proven test methods are available for examining the logical correctness of systems [22, 10]. But there is a lack of support for testing the temporal behavior of systems. Only very few works deal with testing the temporal behavior of real-time systems. Braberman *et al.* have published an approach that is based on modeling the system design with a particular, formally defined SA/SD-RT notation that is translated into high-level timed Petri nets [4]. Out of this formal model a symbolic representation of the temporal behavior is formed, the time reachability tree. Each path from the root of the tree to its leaves represents a poten-

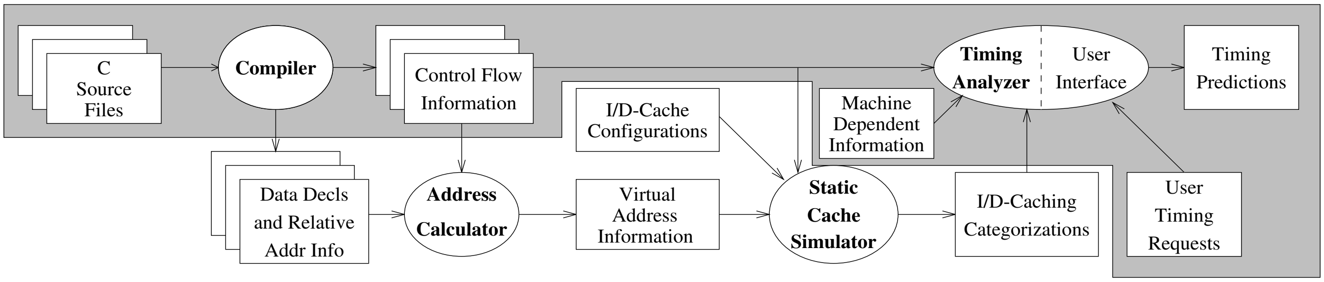

Figure 1. Framework for Timing Predictions

tial test case. The tree already becomes very extensive for small programs so that the number of test cases must be restricted according to different criteria. Results of any practical trial testing of this approach are not reported. Mandrioli *et al.* developed an interactive tool that enables the generation of test cases for real-time systems from formal specifications written in TRIO [18]. The language TRIO extends classical temporal logic to deal explicitly with time measures. At present, however, the applicability of the tool is restricted to small systems whose properties are specified through simple TRIO formulas. Clarke and Lee [6] as well as Dasarathy [7] describe further techniques for verifying timing constraints using timed process algebra or finite-state machines.

All of these approaches demand the use of formal specification techniques. Since the use of formal methods has not yet been generally adopted in industrial practice due to the great expenditure connected with it and the lack of maturity of the existing tools, the testing approaches mentioned have not spread far in industry, particularly since the suitability of these approaches in many cases remains restricted to small systems. Accordingly, there are no specialized methods available at the moment that are suited for testing the temporal behavior of real-time systems. For that reason, testers usually go back to conventional test procedures developed originally for the examination of logical correctness, *e.g.,* systematic black-box or white-box oriented test methods. Since the temporal behavior of complex systems is hard to comprehend and can therefore be examined only insufficiently with traditional test methods, existing test procedures must be supplemented by new methods, which concentrate on determining whether or not the system violates its specified timing constraints. Therefore we examine the applicability of evolutionary testing (ET) to test the temporal behavior of real-time systems.

3. Static Analysis (SA)

Our framework of WCET prediction uses a set of tools as depicted in Figure 1. An optimizing compiler has been modified to emit control-flow information, data information, and the calling structure of functions in addition to regular target code generation. Up to now, the research compiler VPCC/VPO [3] performed this task. We are currently integrating Gnat/Gcc [26, 28] into this environment.

A static cache simulator uses the control-flow information and calling structure in conjunction with the cache configuration to produce instruction and data categorizations, which describe the caching behavior of each instruction and data reference. We currently use a separate analyzer for instruction and data caches since data references require separate preprocessing *via* an address calculator. Current work also includes a single analyzer for unified caches and the handling of secondary caches [20]. The timing analyzer uses these categorizations and the control-flow information to perform a path analysis of the program. It then predicts the BCET and WCET for portions of the program or the entire program, depending on user requests.

In the experiments described in section 5, we chose an architecture without caches for reasons also explained in section 5. Thus, only the portion of the toolset shaded grey in Figure 1 was used in these experiments. Next, we describe the interaction of the various tools of the entire framework. The framework can be retargeted by changing the cache configurations and porting the machine description. However, the largest retargeting overhead constitutes a port of the compiler. Thus, our current efforts to integrate Gnat/Gcc into the framework will greatly improve portability.

3.1. Static Cache Simulation

Static cache simulation provides the means to predict the caching behavior of the instructions and data references of a program/task (see Static Cache Simulator in Figure 1). The addresses of instruction references is obtained from the control-flow information emitted by the compiler. Addresses of data references are calculated by the Address Calculator (see Figure 1) from locating data declarations for global data and obtaining offsets for relative addresses of local data, which are translated into virtual addresses by taking the context of a process into account. For both instruction and data references, the caching behavior is dis-

Category	1st reference	consecutive ref.
Always-hit	hit	hit
Always-miss	miss	miss
First-hit	miss	hit
First-miss	hit	miss

Table 1. Categorizations for Cache References

tinguished by the categories described in Table 1. For each category, the cache behavior of the first reference and consecutive references is distinguished. Consecutive references are strictly due to loops since we distinguish function invocations by their call sites. For data caches, an additional category, called *calculated*, denotes the total number of data cache misses out of all references within a loop for a memory reference.

A program may consist of a number of loops, possibly nested and distributed over several functions. For each loop level, an instruction receives a distinct categorization. The timing analyzer can then derive tight bounds of execution time by inspecting the categorizations for each loop level.

Since instruction categorizations have to be determined by inter-procedural analysis of the entire program, the call graph of the program has to be analyzed. The method of static cache analysis traces the origin of calls within the call graph by distinguishing *function instances*. Since instruction categorizations for a function are specified for each function instance, the timing analyzer can interpret different caching behaviors depending on the calling sequence to yield tighter WCET predictions.

The static cache simulator determines the categories of an instruction based on a novel view of cache memories, using a variation of iterative inter-procedural data-flow analysis (DFA). The following information results from DFA:

- The **abstract cache state** describes which program lines that map into certain cache blocks may potentially be cached within the control flow.
- The **linear cache state** contains the analog information in the (hypothetical) absence of loop.
- The **post-dominator set** describes the program lines certain to still be reached within the control flow.

The above data-flow information can also be reduced with respect to certain subsets, in particular to check if the information is available within a certain loop level. A formal framework for this analysis for instruction and data caches is described in [31]. The data-flow information provides the means to derive the above categories, for example for set-associative instruction caches with multiple levels of associativity. The following categories are derived for each loop level of an instruction for the worst-case cache behavior:

Always-hit: (on spatial locality within the program line) or ((the instruction is in cache in the absence of loops) and ((there are no conflicting instructions in the cache state) or (all conflicts fit into the remaining associativity levels))).

First-hit: (the instruction was a first-hit for inner loops) or (it is potentially cached, even without loops and even for all loop preheaders, it is always executed in the loop, not all conflicts fit into the remaining associativity levels but conflicts within the loop fit into the remaining associativity levels for the loop headers, even when disregarding loops).

First-miss: the instruction was a first-miss for inner loops, it is potentially cached, conflicts do not fit into the remaining associativity levels but the conflicts within the loop do.

Always-miss: This is the conservative assumption for the prediction of worst-case execution time when none of the above conditions apply.

A loop header is an entry block into the loop with at least one predecessor block outside the loop, called the preheader, and at least one predecessor block inside the loop.

3.2. Timing Analysis

The timing analyzer (see Figure 1) calculates the BCET and WCET by constructing a timing tree, traversing paths within each loop level, and propagating the timing information bottom-up within the tree. During the traversal, the timing analyzer has to simulate hardware characteristics (*e.g.,* pipelining) and the instruction categorizations have to be interpreted.

The timing analyzer does not have to take the cache configuration into account. Instead, the instruction categorizations, as introduced above, are used to interpret the caching behavior. The approach of splitting cache analysis via static cache simulation and timing analysis makes the caching aspects completely transparent to the timing analyzer. Solely based on the instruction categorizations, the timing analyzer can derive the WCET by propagating timing predictions bottom-up within the timing tree.

The timing tree represents the calling structure and the loop structure of the entire program. As seen in the context of the static cache simulator, functions are distinguished by their calling paths into function instances. This allows a tighter prediction of the WCET due to the enhanced information about the calling context. Each function instance is regarded as a loop level (with one iteration) and is represented as a node in the timing tree. Regular loops within the program are represented as child nodes of its surrounding function instance (outer-most loops) or as child nodes of another loop that they are nested in.

The timing analyzer determines the BCET and WCET in a bottom-up traversal of the tree. For any node, all possible paths (sequences of basic blocks) within the current loop level have to be analyzed, which will be described in more detail for the WCET. When a child node is encountered along a path, its WCET is already calculated and can simply be added to the WCET of the current path, sometimes with small adjustments. Adjustments are necessary for transitions from first-misses to first-misses and always-misses to first-hits between loop levels [1]. For a loop with n iterations, a fix-point algorithm is used to determine the cumulative WCET of the loop along a sequence of (possibly different) paths. Once a pattern of longest paths has been established, the remaining iterations can be calculated by a closed formula. In practice, most loops have one longest path. Thus, the first iteration is needed to adjust the WCET of child loops along the path, and the second iteration represents the fix-point time for all remaining iterations. The scope of the WCET analysis can such be limited to one loop level at a time, making timing analysis very efficient compared to an exhaustive analysis of all permutations of paths within a program. See [1, 11] for a more detailed description of the timing analyzer and an analog description for the BCET.

4. Evolutionary Testing (ET)

Evolutionary testing is a new testing approach, which combines testing with evolutionary computation. In first experiments the application of ET for examining the temporal behavior of real-time systems achieved promising results. In ten experiments performed ET always achieved better results compared to random testing with respect to effectiveness as well as efficiency. More extreme execution times were found by means of evolutionary computation with a less or equal testing effort than for random testing (see [30] for more details).

4.1. A Brief Introduction to Evolutionary Computation

Evolutionary algorithms represent a class of adaptive search techniques and procedures based on the processes of natural genetics and Darwin's theory of evolution. They are characterized by an iterative procedure and work in parallel on a number of potential solutions, the population of individuals. In every individual, permissible solution values for the variables of the optimization problem are coded. Evolutionary algorithms are particularly suited for problems involving large numbers of variables and complex input domains. Even for non-linear and poorly understood search spaces evolutionary algorithms have been used successfully because of their robustness.

The evolutionary search and optimization process is based on three fundamental principles: selection, recombination, and mutation. The concept of evolutionary algorithms is to evolve successive generations of increasingly better combinations of those parameters, which significantly effect the overall performance of a design. Starting with a selection of good individuals, the evolutionary algorithm achieves the optimum solution by the random exchange of information between these increasingly fit samples (recombination) and the introduction of a probability of independent random change (mutation). The adaptation of the evolutionary algorithm is achieved by the selection and reinsertion procedures since these are based on fitness. The fitness-value is a numerical value, which expresses the performance of an individual with regard to the current optimum. The notion of fitness is essential to the application of evolutionary algorithms; the degree of success in using them may depend critically on the definition of a fitness function that changes neither too rapidly nor too slowly with the design parameters. Figure 2 gives an overview of a typical procedure of evolutionary optimization.

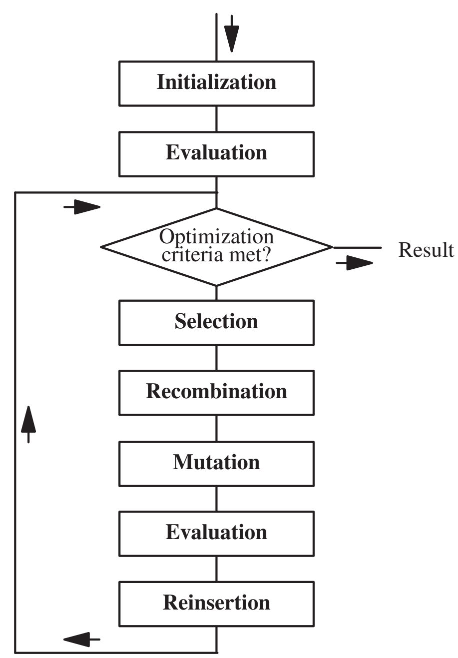

Figure 2. The Process of Evolutionary Computation

At first, a population of guesses to the solution of a problem is initialized, usually at random. Each individual in the population is evaluated by calculating its fitness. The results obtained will range from very poor to good. The remainder of the algorithm is iterated until the optimum is achieved or another stopping condition is fulfilled. Pairs of individuals are selected from the population and are combined in some way to produce a new guess in an analogous way to biological reproduction. Selection and combination algorithms are numerous and vary. A survey can be found in [24].

After recombination the offspring undergoes mutation. Mutation is the occasional random change of a value, which alters some features with unpredictable consequences. Mutation is like a random walk through the search space and is used to maintain diversity in the population and to keep the population from prematurely converging on one local solution. Besides, mutation creates genetic material that may not be present in the current population [27]. Afterwards, the new individuals are evaluated for their fitness and replace those individuals of the original population who have lower fitness values (reinsertion). Thereby a new population of individuals develops, which consists of individuals from the previous generation and newly produced individuals. If the stopping condition remains unfulfilled, the process described will be repeated.

4.2. Applying Evolutionary Computation to Testing Temporal System Behavior

The major objective of testing is to find errors. As described in section 2, real-time systems are tested for their logical correctness by standard testing techniques. The fact that the correctness of real-time systems depends not only on the logical results of computations but also on providing the results at the right time adds an extra dimension to the verification and validation of such systems, namely that their temporal correctness must be checked. The temporal behavior of real-time systems is defective when such computations of input situations exist that violate the specified timing constraints. Normally, a violation means that outputs are produced too early or their computation takes too long. The task of the tester therefore is to find the input situations with the shortest or longest execution times to check if they produce a temporal error. This search for the shortest and longest execution times can be regarded as an optimization problem to which evolutionary computation seems an appropriate solution.

Evolutionary computation enables a totally automated search for extreme execution times. When using evolutionary optimization for determining the shortest and longest execution times, each individual of the population represents a test datum with which the test object is executed. In our experiments the initial population is generated at random. If test data has been obtained by a systematic test, in principle, these could also be used as initial population. Thus, the evolutionary approach benefits from the tester's knowledge of the system under test. For every test datum, the execution time is measured. The execution time determines the fitness of the test datum. If one searches for the WCET, test data with long execution times obtain high fitness values. Conversely, when searching for the BCET, individuals with short execution times obtain high fitness values. Members of the population are selected with regard to their fitness and subjected to combination and mutation to generate new test data.

By means of selection, it is decided what test data are chosen for reproduction. In order to retain the diversity of the population, and to avoid a rapid convergence against local optima, not only the fittest individuals are selected, but also those individuals with low fitness values obtain a chance of recombination. In our experiments stochastic universal sampling [2] was used as selection strategy. For the recombination of test data discrete recombination [21] was applied, a simple exchange of variable values between individuals (see Figure 3). The probability of mutating an individual's variables was set to be inversely proportional to its number of variables. The more dimensions one individual has, the smaller is the mutation probability for each single variable. This mutation rate has been used with success in a multitude of experiments [24, 27]. It is checked if the generated test data are in the input domain of the test object. Then, the individuals produced are also evaluated by executing the test object with them. Afterwards, the new individuals are united with the previous generation to form a new population according to the reinsertion procedures laid down.

In our experiments we applied a reinsertion strategy with a generation gap of 90%. The next generation therefore contained more offspring than parents since 90% of a population's individuals were replaced by offspring. This process repeats itself, starting with selection, until a given stopping condition is reached, *e.g.,* a certain number of generations is reached or an execution time is found, which is outside the

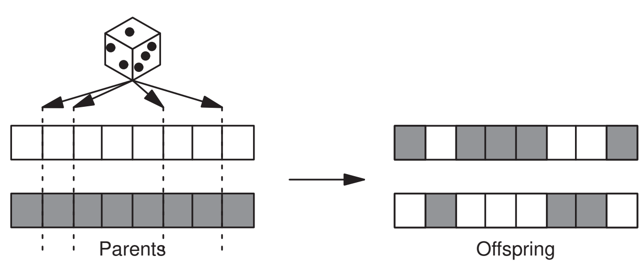

Figure 3. Discrete Recombination with four Randomly Defined Crossover Points

specified timing constraints. In this case, a temporal error is detected. If all the times found meet the timing constraints specified for the system under test, confidence in the temporal correctness of the system is substantiated. In all experiments evolutionary testing was stopped after a predefined number of generations which we have specified according to the complexity of the test objects with respect to their number of input parameters and lines of code (LOC).

5. Verifying Timing Constraints: SA vs. ET

We used SA and ET in five experiments to determine the BCET and WCET of different systems. Except for the last two examples described in this section, all programs tested come from typical real-time systems used in practice. The test programs were chosen since three of them cover different areas within industrial applications of the Daimler Benz company, and the remaining two programs serve as a reference to related work, where these had been used as examples for general-purpose algorithms within real-time applications. The test program also cover a wide range of real-time applications within graphics, transportation, defense, numerical analysis and standard algorithms. Of course, the results are dependent on the hardware/software platform and are generally not directly transferable from one to another since the processor speed and the compiler used directly affect the temporal behavior. All the experiments that are described in the following were carried out on a SPARC-station IPX running under Solaris 2.3 with 40 MHz. The execution times in processor cycles were derived by SA and ET.

We chose a SPARC IPX platform since this architecture does not have any caches. At the current stage of development, the timing analyzer for SA only supports either instruction cache categorizations or data cache categorization. We are working on an extension to support both categorizations at the same time. For ET the execution times were measured using the performance measurement tool Quantify, available from Rational. Quantify performs cycle-level timing though object code instrumentation. Thus, overheads of the operating system were ruled out, and the execution times reported were the same for repeated runs with identical parameters. However, Quantify does not take the effects of caching into account. Thus, we needed an uncached architecture to perform our experiments.

The SA approach utilized the pipeline simulation of the timing analyzer for the experiments. The instruction execution was simulated for a five-stage pipeline with a throughput of one instruction per cycle for most cases, as commonly found in RISC architectures. Load and store instructions caused a stall of two cycles to access memory. Floating point instructions resulted in stalls with varying durations, specified for the best case and worst case of such operations. The timing analyzer calculates a conservative estimate of the number of cycles required for an execution based on path analysis. For the worst case, the estimate is guaranteed to be greater or equal than the actual WCET. Conversely, the estimate is less or equal than the actual BCET.

The library of evolutionary algorithms, which we applied for ET, was a Matlab-based toolbox developed at the Daimler-Benz laboratories by Hartmut Pohlheim. It provides a multitude of different evolutionary operators for selection, recombination, mutation, and reinsertion [24]. For each experiment, the evolutionary algorithms were applied twice; first, to find the longest execution time, and then the shortest. The fitness was set equal to either the execution time measured in processor cycles for the longest path or its reciprocal for the shortest path. The population size was varied for the experiments according to the complexity of the test objects. Pairs of test data were chosen at random and combined using different operators like discrete recombination or double crossover depending on the representation of the individuals. The mutation probability was set inversely proportional to the length of the individuals. There is no means of deciding when an optimum path has been found, and ET was usually allowed to continue for 100 generations before it was stopped.

5.1. Test Objects

The first example is a simple computer graphics function written in C, which checks whether or not a line is covered by a given rectangle with its sides parallel to the axes of the co-ordinate system. The function has two input parameters: the line given by the co-ordinates of both line end points, and the rectangle, which is described using the position of its upper left corner, its width and its height. This amounts to eight atomic input variables altogether. The function has 107 LOC and contains a total of 37 statements in 16 program branches.

The second application comes from the field of railroad control technology. It concerns a safety-critical application that detects discrepancies between the separate channels in a redundant system. It has 389 LOC and 512 different input parameters: 16 binary variables, 384 variables ranging from 0 to 255 and 112 variables with a range of each from 0 to 4095.

The third application concerned comes from the field of defense electronics. It is an application that extracts characteristics from images. A picture matrix is analyzed with regard to its brightness, and the signal-to-noise ratio of its brightest point and its background is established. The defense electronics program has 879 LOC and 843 integer input parameters. The first two input parameters represent the position of a pixel in a window and lie within the range 1..1200 and 1..287 respectively. The remaining 841

Program	Graphics		Railroad		Defense		Matrix		Sort	
Method	best	worst	best	worst	best	worst	best	worst	best	worst
SA	309	2,602	389	23,466	848	71,350	8,411,378	15,357,471	16,003	24,469,014
actual	N/A	N/A	N/A	N/A	N/A	N/A	10,315,619	13,190,619	20,998	11,872,718
ET	457	2,176	508	22,626	9,095	35,226	12,050,569	13,007,019	1,464,577	11,826,117

Table 2. Execution Times $[cycles]$ for Test Programs

parameters define an array of 29 by 29 pixels representing a graphical input located around the specified position; each integer describes the pixel color and lies in the range 0..4095.

The fourth sample program multiplies two integer matrices of size 50 by 50 and stores the result in a third matrix. Only integer parameters in the range between 0 and 8095 are permissible as elements of the matrices. Matrix operations are typical for embedded image processing applications.

The fifth test program performs a sort of an array of 500 integer numbers using the bubblesort algorithm. Arbitrary integer values can be sorted. Sorting operations are common for countless applications within and beyond the area of real-time systems.

5.2. Experiments

For all test objects mentioned the shortest (best) and longest (worst) execution times were determined. The results of the experiments are summarized in Table 2 for the best case and worst case. The first row depicts the results for static analysis and the last row shows the measurements for evolutionary testing. The middle row shows the *actual* shortest and longest execution times for the multiplication of matrices and the bubblesort algorithm that were easily determined by applying a systematic test. Notice that the actual execution times could only be determined with certainty in the absence of caching due to hardware complexities [31]. The other examples of actual real-time systems are so complex with regard to their functionality that their extreme execution times cannot be definitely determined by a systematic test. For applications used in practice this is the normal case.

For the computer graphics example SA calculated a lower bound of 309 processor cycles for the shortest execution time and an upper bound of 2602 cycles for the longest execution time. ET discovered a shortest time of 457 cycles, and a longest time of 2176 cycles within 24 generations. The population size was set to 50. The generation of 76 additional generations with 3800 test data sets does not produce any longer or shorter execution times. Thus the shortest execution times determined vary by 32%, the longest by 16%.

For the railroad technology example the population size for ET was increased to 100 because of the complex input interface of the test object with its more than 500 parameters. Starting from the first generation a continuous improvement up to the 100th generation could be observed for ET. This suggests that ET would find even more extreme execution times if the number of generations was increased. The shortest execution time found by ET so far (508 cycles) is nearly 24% above the 389 cycles computed by SA. The longest execution time determined (22626 cycles) varies only by 4% from the one calculated by SA (23466 cycles). Therefore, the worst-case execution time of this example can already be defined very accurately after 100 generations. It can be guaranteed that the maximum execution time of this task lies between 22626 cycles and 23466 cycles.

The defense electronics program has 843 input parameters. Therefore, the population size in this experiment was also set to 100. For this example, evolutionary algorithms were used to generate pictures surrounding a given position. The number of generations was increased to 300 because of the large range of the variables and the large number of input parameters. Again the longest execution time increased steadily with each new generation and asymptoted towards the current maximum of 35226 cycles when the run was terminated after 300 generations. The fastest execution time was found to be 9095 after 300 generations. Compared to the results achieved by SA significant differences could be observed. The estimates for the extreme execution times calculated by SA are 848 cycles and 71350 cycles. A closer analysis of possible causes for these deviations lead to the possibility that certain instructions were assumed to take different times for their pipeline execution. The instructions in question are multiply and divide, which account for multiple cycles during the execution stage. We are currently trying to isolate these effects for the Quantify tool to allow a proper comparison with SA.

The next example in the table is the multiplication of matrices. Due to its functional simplicity the minimum and maximum run time can very easily be determined by systematic testing because they represent special input situations. The longest execution time of 13190619 cycles results if all elements of both matrices are set to the largest permissible value (8095). The shortest run time of

10315619 cycles results if both matrices are fully initialized with 0. When ET is applied to the multiplication of matrices a single individual is made up of 5000 parameters (2*50*50). The resulting search space is by far the largest of the examples presented here. For each generation with 100 individuals 500000 parameter values have to be generated. Nevertheless, the number of generations for this example was increased to 2000. When searching for the longest execution time, a maximum of 13007019 cycles was found. The evolutionary algorithms had found an execution time that lies only a good 1% below the absolute maximum. The longest execution time that was determined with the help of SA (15357471 cycles) exceeds the absolute maximum by about 16%. The shortest execution time determined by the evolutionary algorithms is 12050569 cycles, which means a deviation of 17% compared to the actual shortest run time. The deviation of SA is nearly similar: the execution time of 8411378 cycles lies about 18% below the actual value.

The last example is the bubblesort algorithm. Again the determination of the extreme run times is very easy with the help of a systematic test. The longest execution time for bubblesort results from the list sorted in reverse and amounts to 11872718 cycles. The shortest run time results, of course, from the sorted list, which leads to an execution time of just 20998 cycles. Once again the longest execution time found by ET (11826117 cycles) comes close to the actual maximum. It deviates by less than 1%. The upper bound (24469014 cycles) for the longest execution time that was determined by SA exceeds the actual one by more than 100%. This overestimation is caused by a deficiency of the algorithm that interpolates the execution time for loops. In particular, two loops are nested with a loop counter of the inner loop whose initial value is dependent on the counter of the outer loop. Currently, the timing analyzer estimates the number of iterations of the inner loop conservatively as n^2, where n is the maximum number of iterations for the outer loop. We are working on a method to handle such loop dependencies to correctly estimate the number of iterations for nested loops. In this case, the inner loop has $\frac{1}{2}n^2$ iterations. As a coarse estimate, 12234507 cycles or half the estimated value should be calculated taking the actual loop overhead into account, *i.e.,* the value would be around 3% off the actual value. Further discussions will refer to this adjusted value. The shortest execution time of the bubblesort algorithm is only insufficiently evaluated by evolutionary optimization (1464577 cycles). Although shorter run times have been continually found over 2000 generations the results are far from the absolute minimum. For that reason current work focuses on a detailed analysis of the bubblesort example and an improvement of the ET results. Also, our main focus was to bound the WCET since this provides the means to verify that deadlines cannot be missed, a very important property of real-time systems. The shortest execution time determined by SA (16003 cycles) differs by 24% from the absolute minimum.

6. Discussion

The measurements of the last section show that the methods of static analysis and ET bound the actual execution times. While SA always estimates the extreme execution times in such a way that the actual run times possible for the system will never exceed them, ET provides only actually occurring execution times. For the worst case, the estimates of SA provide an upper bound while the measurements of ET give a lower bound on the actual time. Conversely, SA's estimates provide a lower bound for the best-case time while ET's measurements constitute an upper bound.

In about half of the experiments, the actual execution times were bounded within $\pm 3\%$ or better with respect to the range of execution times determined by SA. These results are directly applicable to schedulability analysis and provide a high confidence about the range for the actual WCET. In further cases, the variation between the two approaches was about $\pm 10\%$, which may still yield useful results for schedulability analysis. We regard $\pm 10\%$ as a threshold for useful results in the sense that larger deviations between the two methods may not be accurate enough to guarantee enough processor utilization, even though they may be safe. For the multiplication of matrices and the defense example larger variations were detected. This indicates that both approaches need further investigation to improve their precision.

The overhead for estimating the extreme execution times differs for both approaches. ET requires the execution of a test program over many generations with a large number of input data, *i.e.,* the overhead is dependent on the actual execution times of the test object and additional delays caused by the timing. SA requires a test overhead in the order of seconds for the tested programs since one simulation suffices to predict the extreme execution times, *i.e.,* the overhead is independent of the actual execution times. Instead, the overhead depends on the complexity of the combined call graph and control-flow graphs of the entire program and roughly increases quadratically with the program size. SA automatically yields not only timing estimates for the entire application, but also estimates for arbitrary subroutines or portions of the control flow. To obtain corresponding data with ET test objects have to be isolated.

A prerequisite for performing SA is the knowledge of the cycle-level behavior for the target processor that has to be supplied in configuration files. The ET approach works for a wide range of timing methods. On one hand, hardware timers calculating wall-clock time may be used without knowledge of the actual hardware. This method is highly portable but subject to interference with hardware and software components, *e.g.,* caches and operating systems. On the other hand, cycle-level timing information, excluding the instrumentation instructions, may be calculated as part of the program execution, as seen in the above experiments. The portability of this method is constrained by the portability of the instrumentation tool.

In summary, the ET approach cannot provide safe timing guarantees. It measures the actual, running system. ET is universally applicable to arbitrary architectures and requires knowledge about the input specification. The SA approach yields conservative estimates that safely approximate the actual execution times. It requires knowledge about loop frequencies and information about the cycle-level behavior of the actual hardware. New hardware features have to be implemented in the simulator, which limits the portability of SA. If hardware details are not known, only the ET approach can be applied. If the hardware is not available yet but the specification of the hardware has been supplied, only the SA approach will yield results.

We regard the two methods as complementary approaches. Whenever deadlines have to be guaranteed SA should be used to yield safe estimates of the WCET. ET may be used additionally to bound the extreme execution times more precisely. Furthermore, ET may suffice if missed deadlines can be tolerated sporadically. The WCET for schedulability analysis should also be derived from SA for hard real-time environments where soft real-time environments may choose between SA and ET, or even the mean between SA and ET. In general, every real-time system should be tested for its logical as well as temporal correctness. Independent of the methods used during system design, we recommend to apply ET to validate the temporal correctness of systems. The confidence in the application is increased since ET checks for timing violations over many input configurations.

7. Conclusion and Future Work

This work introduced two methods to verify timing constraints of actual real-time applications, namely the method of static analysis and the method of evolutionary testing. Both methods were implemented and evaluated for a number of test programs with respect to their prediction of the worst-case and best-case execution times. The results show that the methods are complimentary in the sense that they bound the actual extreme execution times from opposite ends.

For most of the investigated programs the actual execution times for the best and worst cases could be guaranteed with fairly high precision. They are within $\pm 10\%$ of the mean between the results of both methods relative to the possible execution times determined by static analysis. Less precise results were obtained for few experiments indicating that further improvements for both approaches are necessary to ensure their general applicability. Current work on static analysis includes extensions to handle loop dependencies and integrate the Gnat/Gcc compiler. Current work on evolutionary testing focuses on the development of robust algorithms that reduce the probability of getting caught in local optima. Furthermore, suitable stopping criteria to terminate the test are to be defined. If the program code is available the degree of coverage achieved during evolutionary testing and the observation of the program paths executed could be an interesting aspect for deciding when to stop the test. The most promising criteria seem to be branch and path coverage because of the strong correlation between the program's control flow, the execution of its statements and the resulting execution times. The coverage reached will also be used to assess the test quality when comparing evolutionary testing with systematic functional testing – another area where we want to intensify our research in the future, in order to estimate thoroughly the efficiency of different testing approaches for the examination of real-time systems' temporal behavior.

In comparison, evolutionary testing should be more portable but requires extensive experimentation over many program executions. Static analysis has a lower overhead for the simulation process but requires detailed information of hardware characteristics and extensions to the simulation models for new architectural features. We recommend that the worst-case execution time for schedulability analysis be derived from static analysis for hard real-time environments where soft real-time environments may choose between static analysis and evolutionary testing. Furthermore, we suggest that evolutionary testing be used to increase the confidence in the temporal correctness of the actual, running system.

References

[1] R. Arnold, F. Mueller, D. B. Whalley, and M. Harmon. Bounding worst-case instruction cache performance. In *IEEE Real-Time Systems Symposium*, pages 172–181, Dec. 1994.

[2] J. E. Baker. Reducing bias and inefficiency in the selection algorithm. In *International Conference on Genetic Algorithms and their Application*, July 1987.

[3] M. E. Benitez and J. W. Davidson. A portable global optimizer and linker. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 329–338, June 1988.

[4] V. Braberman, M. Felder, and M. Marre. Testing timing behavior of real-time software. In *International Software Quality Week*, May 1997.

[5] J. V. Busquets-Matraix. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In

IEEE Real-Time Technology and Applications Symposium, pages 204–212, June 1996.

[6] D. Clarke and I. Lee. Testing real-time constraints in a process algebraic setting. In *International Conference on Software Engineering*, Apr. 1995.

[7] B. Dasarathy. Timing constraints of real-time systems: Constructs for expressing them, methods of validating them. *IEEE Transactions on Software Engineering*, 11(1):80–86, 1985.

[8] C. G. Davis. Testing large, real-time software systems. In *Software Testing, Infotech State of the Art Report*, volume 2, pages 85–105, 1979.

[9] C. Ferdinand, F. Martin, and R. Wilhelm. Applying compiler techiniques to cache behavior prediction. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pages 37–46, June 1997.

[10] M. Grochtmann, K., and Grimm. Classification trees for partition testing. *Software Testing, Verification & Reliability*, 3(2):63–82, 1993.

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

[12] W. S. Heath. *Real-Time Software Techniques*. Van Nostrand Reinhold, 1991.

[13] S. Kim, S. Min, and R. Ha. Efficient worst case timing analysis of data caching. In *IEEE Real-Time Technology and Applications Symposium*, June 1996.

[14] D. B. Kirk. SMART (strategic memory allocation for real-time) cache design. In *IEEE Real-Time Systems Symposium*, pages 229–237, Dec. 1989.

[15] C. Lee, J. Hahn, Y. Seo, S. Min, R. Ha, S. Hong, C. Park, M. Lee, and C. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. In *IEEE Real-Time Systems Symposium*, Dec. 1996.

[16] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *IEEE Real-Time Systems Symposium*, pages 254–263, Dec. 1996.

[17] J. Liedke, H. Härtig, and M. Hohmuth. Os-controlled cache predictability for real-time systems. In *IEEE Real-Time Technology and Applications Symposium*, pages 213–223, June 1997.

[18] D. Mandrioli, S. Morasca, and A. Morzenti. Functional test case generation for real-time systems. In *IFIP Working Conference on Dependable Computing for Critical Applications*, pages 29–61, Sept. 1992.

[19] F. Mueller. *Static Cache Simulation and its Applications*. PhD thesis, Dept. of CS, Florida State University, July 1994.

[20] F. Mueller. Timing predictions for multi-level caches. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pages 29–36, June 1997.

[21] Mühlenbein and Schlierkamp-Vossen. Predictive models for the breeder genetic algorithm: I. continuous parameter optimization. *Evolutionary Computation*, 1(1):25 – 49, 1993.

[22] G. J. Myers. *The Art of Software Testing*. John Wiley & Sons, 1979.

[23] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–61, Mar. 1993.

[24] H. Pohlheim. GEATbx: Genetic and evolutionary algorithm toolbox for use with matlab - documentation. Technical report, Technical University Ilmenau, Germany, 1996.

[25] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1(2):159–176, Sept. 1989.

[26] R. M. Stallman. *Using and Porting GNU CC*. Free Software Foundation, Inc., 2.7.2 edition, Nov. 1995.

[27] H.-H. Sthamer. *The Automatic Generation of Software Test Data Using Genetic Algorithms*. PhD thesis, Dept. of Electronics and Information Technology, University of Glamorgan, Wales, UK, 1996.

[28] N. Y. University. The Gnu NYU Ada Translator (GNAT). Available by anonymous FTP from cs.nyu.edu.

[29] J. Wegener and R. Pitschinetz. Tessy - yet another computer-aided software testing tool? In *European International Conference on Software Testing Analysis & Review*, Oct. 1994.

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

[31] R. White, F. Mueller, C. Healy, D. Whalley, and M. Harmon. Timing analysis for data caches and set-associative caches. In *IEEE Real-Time Technology and Applications Symposium*, pages 192–202, June 1997.