, , 1–27 ()
© Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

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

JOACHIM WEGENER joachim.wegener@daimlerchrysler.com
DaimlerChrysler AG, Research and Technology, Alt-Moabit 96a, 10559 Berlin, Germany, phone: (+49) (30) 39982-232, fax:-107

FRANK MUELLER mueller@informatik.hu-berlin.de
Humboldt University Berlin, Institut f. Informatik, Unter den Linden 6, 10099 Berlin, Germany, phone: (+49) (30) 2093-3011, fax:-3010

Editor: Wolfgang A. Halang

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.

Keywords: Real-time systems, timing analysis, static timing analysis, testing, genetic algorithms, evolutionary testing.

1. Introduction

For real-time systems, the correct system functionality depends on temporal correctness as well as on logical correctness. Accordingly, the verification of temporal behavior is an important aspect in the development of real-time systems. The most important development phases for the verification of temporal behavior are system design and testing.

In the course of system design, 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 to ensure that minimum sampling intervals are met. The tighter the bounds are to the actual best or worst case, the higher can a system be utilized and, as a consequence, the lower are the hardware costs of such a system.

Techniques of static analysis (SA) can be used during 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 of a task's code at a cycle level for hardware concepts such as caches and pipelines of a given processor.

Dynamic testing (in the following referred to as "testing") is an inevitable part of the verification and validation process for real-time systems. Testing is aimed at finding errors in the system under test and building up confidence in its correct behavior by executing the test object on the target system with selected input values. Since a complete test, i.e. a test with all possible input situations, usually cannot be carried out interesting test data has to be selected according to special criteria. An investigation of existing test methods shows that they concentrate mostly on testing logical correctness; there is a lack of support for testing the temporal system behavior. For this reason, DaimlerChrysler Research and Technology developed a new approach to testing the temporal behavior of real-time systems: Evolutionary Testing (ET) [46]. ET searches automatically for test data that 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.

Previous work deals with detailed comparisons of ET, random testing, and systematic testing. The application of ET achieved promising results: In all experiments performed, ET obtained better results than random testing with respect to effectiveness and efficiency [49, 46]. More extreme execution times were found by means of ET with a less or equal testing effort than for random testing. Similar results were reported by Puschner and Nossal [40]. Compared with systematic testing, ET seems to be better for complex test objects where the temporal behavior is difficult to assess by systematic testing [13, 48].

In this work we concentrate on the comparison of ET and SA in order to further evaluate the suitability of ET for testing the temporal behavior of real-time systems. Although ET and SA 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. These approaches are compared in this paper by means of several examples and in much greater detail than in our previous work [32].

Section 2 offers a general overview of related work on SA as well as on testing. The third section describes the tool we employ for SA. Section 4 introduces ET. Section 5 summarizes the results obtained, which are further discussed in Section 6. The most important statements are summarized in Section 7, which 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 [39]. Often, recursive functions are also not allowed, although there exist outlines on treating bounded recursion similar to bounded loops [28].

Our static analysis framework handles call and return sequences by distinguishing the *instances* of functions, which differ for different calling sequences of the same function. This approach covers non-recursive programs. Recursive programs require a distinction between the first and any subsequent recursive invocation as separate instances. In addition, a lower and upper bound on the recursion depth has to be specified through pragmas or interactively. Our framework also relies on the compiler to infer loop bounds from the program source. Programming constructs of for-loops often contain information about the iteration space but while-loops may contain data-dependent termination conditions. In the former case, compiler analysis of the intermediate code reveals the loop bounds and the timing analyzer handles dependencies between loop counters of nested loops. In the latter case, the programmer needs to supply loop bounds (as a pragma or interactively). The framework currently does not handle data dependencies that are due to branch correlation. It should be feasible to use recent methods to find correlations within a function [33] or even interprocedurally [4] to guide the path selection of the static analyzer.

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 [25, 14], and from uncached architectures to instruction caches [31, 22, 17] and data caches [41, 19, 23, 50].

Today, mainly four fundamental models for static timing analysis exist (see Table 1). First, a source-level oriented timing schema propagates the WCET/BCET through a tree of timing nodes and handles pipelined RISC processors with first-

Table 1. Classification of Static Timing Analysis Work

analysis technique	integrated			separate
level of analysis	timing schema	hardware simulator	constraint based	data-flow based
source code oriented	[37, 25, 19]		[22, 23]	
intermediate code				[14, 15, 50, 31]
object code		[26]		[44], [11]

level split caches [37, 25, 19]. Second, a hardware simulator can be extended to handle known and unknown values in order to simulate an execution with unknown inputs on the cycle level for a given executable [26]. Third, a constraint-based method models architectural aspects, including caches, *via* integer linear programming [23]. Forth, 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 [14, 50, 31]. The first, second and third approaches use integrated analysis of caches while our approach uses separate analysis. Separate analysis allows us to deal with multi-level memory hierarchies or unified caches [30]. Another approach using data-flow analysis to model caching originally used the same categorizations as our approach but a different data-flow model. Recently, this latter approach has been generalized to handle a number of data-flow solutions with differing complexity and accuracy [11] and has been combined with integer linear programming to model architectural characteristics [44].

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 and thus unexpected cache misses may occur 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 [20, 52, 29, 24]. Recently, attempts have been made to incorporate caching into rate-monotone analysis and response-time analysis [6, 7, 21], which allows WCET predictions for non-preemptive systems to be used in the analysis of preemptively scheduled systems. This seems most promising since the information gathered for static timing analysis can be utilized within this extended framework for schedulability analysis. Thus, the results of this paper for static analysis are applicable to both preemptive and non-preemptive systems. However, for preemptive systems the overhead of operating system calls would need to be reflected in the model to account for delays due to resource allocation and synchronization between tasks.

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. Testing typically consumes 50% of the overall development effort and budget [10, 47]. It is one of the most complex and time-consuming activities within the development of real-time systems [16]. In comparison to conventional software systems, the examination of additional requirements like timeliness, simultaneity, and predictability makes testing costly. Also, 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. Nevertheless, testing is an inevitable part of the verification and validation process

of real-time systems. It is the only method that examines run-time behavior, based on an execution in the application environment. Testing is the only means to consider dynamic aspects, which are especially important to rule out malfunctioning of real-time systems, *e.g.,* the dynamic memory allocated during program execution, the synchronization of parallel processes, and the coordination with the underlying real-time operating system.

The aim of testing is to find existing errors in a system and to establish 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 *e.g.,* black-box test methods like the classification-tree method [12] or white-box test methods like branch, path or condition coverage testing [35]. But there is a lack of support for testing the temporal behavior of systems. Only very few studies deal with testing the temporal behavior of real-time systems. Braberman *et al.* [5] 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. 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 potential 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.* [27] have developed an interactive tool that enables the generation of test cases for real-time systems from formal specifications written in TRIO. 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 [8] as well as Dasarathy [9] 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 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 suited for testing the temporal behavior of typical real-time systems at the moment. 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 [46, 48], 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 approach to static timing analysis can be characterized as a intermediate code-based analysis that corresponds to the actual instructions of the architecture but abstracts from the instruction representation for the sake of portability. Thus, code portions of a real-time task (or of a task's subset of code) are analyzed for a given architecture (currently the SPARC), optionally with instruction and data caches with parameters such as cache size, line size and associativity. The analysis uses closed formulae to derive cycle-level estimates of the WCET and BCET.

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 [42, 45] into this environment.

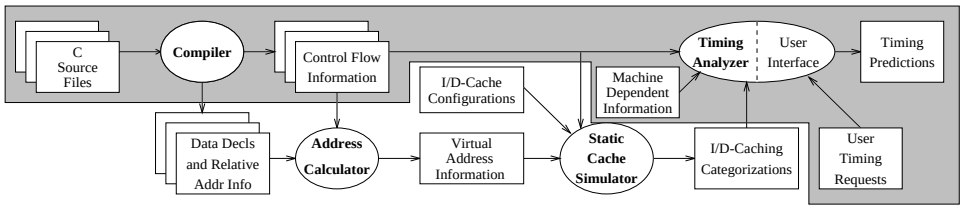

Figure 1. Framework for Timing Predictions

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 [30]. 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 that section. Thus, only the portion of the tool set 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. We proceed with a description of the major components of the timing analysis framework, regardless whether they are utilized or not in this study, to provide the reader with a comprehensive description of the toolset that could not be achieved by merely describing parts of it.

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 distinguished by the categories described in Table 2. 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.

Table 2. Categorizations for Cache References

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

A program may contain 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 associated with given cache blocks may potentially be cached along some control flow path to the given program point.

- The **linear cache state** contains the analogous information assuming the (hypothetical) absence of loop.

- The **dominator cache state** indicates the program lines that are certain to be traversed in the control flow before the given program point, regardless of the actual execution path.

- The **post-dominator set** describes the program lines that are certain to be reached along every future control flow path.

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 is described in [31] for instruction and in [50] for data caches. The data-flow information provides the means to derive the above categories, for example for set-associative instruction caches with multiple levels of associativity. In the following, the term a line l *is potentially cached* or *in a cache state* refer to a line being a member of the abstract cache state of any or of a certain associativity level, respectively. A *conflicting instruction* belongs to a line m that maps into the same cache line as l. Conflicts *fit into* the remaining associativity levels of the cache if their number is less than the associativity level. An instruction is *always executed* if present in the post-dominator of a loop entry. Conflicts *in a loop* refer to conflicts within the subset of lines contained in a certain loop level. Hence, 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 line of the instruction dominates the path and
(the instruction is potentially in some cache cache state) and
((there are no conflicting instructions in this 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, it dominates the path, it is always executed in the loop, not all conflicts fit into the remaining associativity levels but the conflicts at loop entry plus those inside the loop prior to l at level for the loop fit into the remaining associativity levels).

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 (outermost loops), or as child nodes of an outer loop for nested loops.

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. The WCET of a node consists of the WCET of any of its children and its own WCET. The latter is determined by simulating the pipeline of the processor's architecture for the instructions contained within the current node (but not its children). The pipeline analysis allows instructions to execute concurrently at different stages within the pipeline, it detects pipeline stalls and keeps track of the elapsed number of processor cycles along a sequence of instructions. Different execution paths within a node (loop or function) are considered as long as they may contribute towards the WCET, *i.e.*, the lengths of shorter paths are only calculated once and then dismissed. The calculation is performed in a bottom-up traversal. For leaf nodes, this calculation can proceed right away since there are no dependents. 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 [14]. For a loop with n iterations, a fixed-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 fixed-point time for all remaining iterations. The scope of the WCET analysis

can 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 [14, 15] for a more detailed description of the timing analyzer and an analogous description for the BCET.

4. Evolutionary Testing (ET)

As described in Section 2, real-time systems need to be tested in view to their logical as well as their temporal correctness. It is the major objective of testing to find errors. The temporal behavior of real-time systems is defective if computations of input situations exist which violate the specified timing constraints. As a rule, a violation means that outputs are produced too early or their computation takes too long. The tester's task, therefore, is to find the input situations with the shortest or longest execution times in order to check if these produce a temporal error.

The idea of ET is to look for input situations for which computation through the test object leads to a violation of the specified timing constraints. The evolutionary test is applied in order to execute a search for the input situations with the minimum and maximum execution times. This search is taken as an optimization problem which is processed with the support of evolutionary computation. It is not necessary that the evolutionary test finds the input situations with the actually longest and shortest execution times. To call it a successful test, it is sufficient to find input situations, which violate the specified timing constraints. In this case, an error is found.

Evolutionary testing enables a totally automated examination of temporal behavior. The test especially benefits from the fact that test evaluation concerning temporal behavior is usually trivial. Contrary to logical behavior, the same timing constraints apply to large numbers of input situations whereas for examining logical correctness expected values for output parameters have to be determined according to the selected test data. However, because each test — with the exception of a complete test — is a spot check, even the most extensive tests can not rule out that there might be other input situations with longer or shorter execution times than those that have been found.

4.1. A Brief Introduction to Evolutionary Computation

Evolutionary computation forms the generic term for direct, probabilistic search and optimization algorithms gleaned from the model of biological evolution. The main representatives of this computational paradigm, the genetic algorithms, evolution strategies, and the evolutionary programming, which were developed independently, can be seen as instances of a generalized evolutionary algorithm [1]. Due to an increasing exchange of ideas between these techniques, the differences between them are diminishing. Thus, we simply adopt the term evolutionary computation in this paper.

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 a large number 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 that 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.

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 [38].

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 [43].

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.

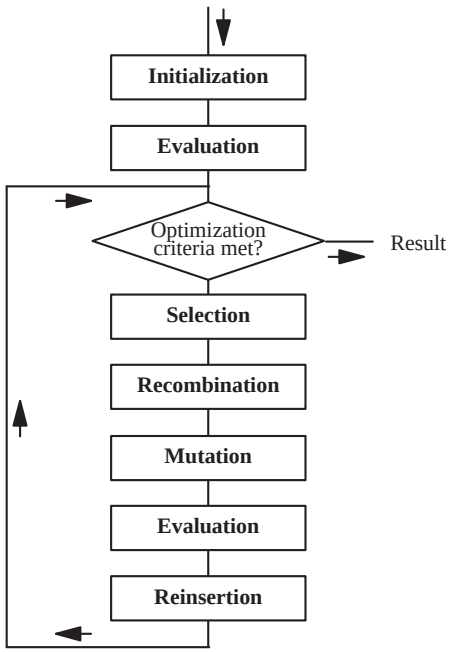

Figure 2. The Process of Evolutionary Computation

4.2. Applying Evolutionary Computation to Testing Temporal System Behavior

In contrast to other optimization methods, evolutionary algorithms are particularly suitable 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. With the exception of very simple real-time systems, the temporal behavior always forms a very complex multi-dimensional search space with many plateaus and discontinuities, since the execution times for several input data executing the same program path are identical whereas the execution of different program paths leads to irregular changes of the execution times. Figure 3 represents a simple two-dimensional variation plot of such a search space. The diagram shows the different execution times for a bubblesort algorithm that sorts a list with 5 elements. The second and third list element are varied within the range of 0..10 while the other elements remain constant. This small example already shows different plateaus of the same execution time and several local optima. More complex examples can be found in [48].

When using evolutionary testing for determining the shortest and longest execution times of test objects, each individual of the population represents a test datum with which the system under test is executed. In our experiments the initial population is generated at random. If test data has been obtained by a systematic test, in

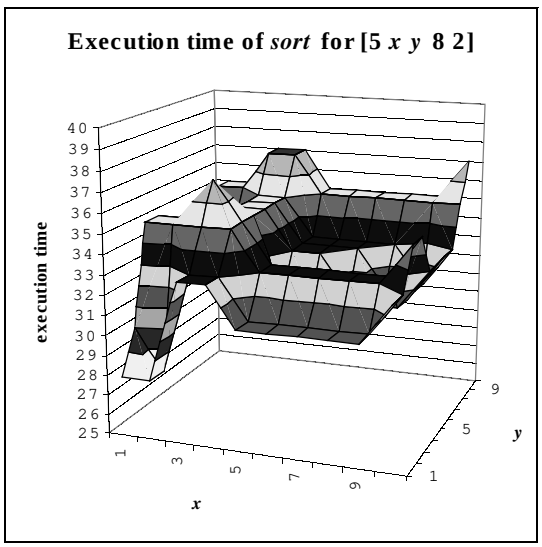

Figure 3. Execution Time Distribution for Sort

principle, these could also be used as an 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 to determine its fitness value. In our experiments, the population is sorted in accordance with the execution times measured. The fitness assigned to each individual depends only on its position in the individuals rank and not on the actual execution time. This so-called rank-based fitness assignment overcomes some problems, which apply to the proportional fitness assignment, and behaves in a more robust manner [51]. 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 which test data are chosen for reproduction. In order to retain the diversity of the population and to avoid a rapid convergence towards a local optimum, not only the fittest individuals are selected but some individuals with low fitness values to allow broader recombination. In our experiments, stochastic universal sampling [2] was used as selection strategy. The individuals are mapped onto contiguous segments of a line, in such a way that each individual's segment is equal in size to its fitness. Equally spaced pointers, which correspond in number to the individuals that are to be selected, are placed over the line. If $Npointer$ is the number of individuals to be selected, then the distance between the pointers is $Length_of_line/Npointer$. The position of the first pointer is given by a randomly generated number in the range of $[0, Length_of_line/Npointer]$ as depicted in Figure 4 (also see [38]).

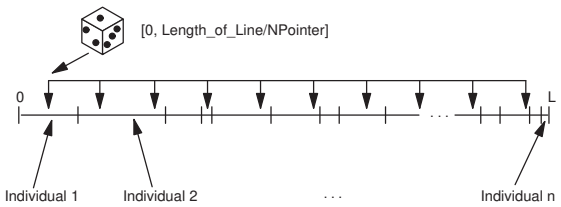

Figure 4. Stochastic Universal Sampling

For the recombination of test data, discrete recombination [34] was applied, a simple exchange of variable values between individuals (see Figure 5). For each variable, the parent that contributes its variable to the offspring is chosen randomly with equal probability.

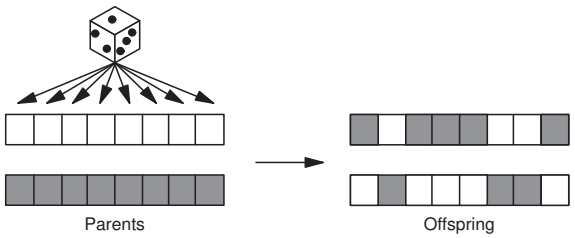

Figure 5. Discrete Recombination

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 the mutation probability for each single variable. Correspondingly, for each variable the mutation probability is $1/number_of_variables$. This mutation rate has been used with success in a multitude of experiments [38, 43].

Afterwards, it has to be checked if the generated test data are in the input domain of the test object. Invalid data are replaced by additionally generated offspring. The individuals produced will then also be evaluated by executing the test object with them and measuring the execution times. The new individuals are united with the previous generation to form a new population according to the reinsertion procedures laid down. 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 specified timing constraints. In this case, a temporal error is detected and the test is successful. 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. However, in this case it becomes con-

siderably more difficult to decide when the test should be terminated in order to establish sufficient confidence in the temporal correctness of the test object. Therefore, DaimlerChrysler Research and Technology and the Stanford University are currently working on developing suitable stopping criteria (see [36]).

In all experiments presented in the following section, evolutionary testing was stopped after a predefined number of generations, specified according to the complexity of the test objects with respect to 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. Three of the test programs come from systems that have to meet hard timing constraints and are thus time critical. They cover different areas within industrial applications of the DaimlerChrysler company. Two of these applications are additionally safety critical. The remaining two programs serve as a reference to related work, where they had been used as examples for general-purpose algorithms within real-time applications. The test programs 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 the other, since the processor speed and the compiler used directly affect the temporal behavior. All experiments described in the following were carried out on a SPARCstation IPX running under Solaris 2.3 at 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, execution times were measured using the performance measurement tool Quantify, available from Rational. Quantify performs cycle-level timing through 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 of varying duration, 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, for the best case 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 DaimlerChrysler laboratories by Hartmut Pohlheim. It provides a multitude of different evolutionary operators for selection, recombination, mutation, and reinsertion [38]. For each experiment, the evolutionary algorithms were applied twice: first, to find the longest execution time, and then the shortest. The execution times of every generated test datum were measured with Quantify in order to determine its fitness. There is no means of deciding when an optimal execution time has been found, and ET was usually allowed to continue for 100 generations before it was stopped. The population size was varied according to the complexity of the test objects.

5.1. Test Objects

The first example is a computer graphics function, 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, described by 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 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 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 Tables 3 and 4 for the best case and worst case. The first row depicts the results for static analysis. The measurements for evolutionary testing are depicted in the second row in Table 3 and in the last row in Table 4. The middle row in Table 4 shows the *actual* shortest and longest execution times for the multiplication of matrices and the bubblesort algorithm, which were easily determined by applying a systematic test. The difference between the SA and ET results in Table 3 are summarized in the last row as percent values. In Table 4, percent differences are listed relative to the actual value. Notice that the actual execution times could only be determined with certainty in the absence of caching due to hardware complexities [50, 31]. The examples of actual real-time systems in Table 3 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 generally is the case.

Table 3. Execution Times [*cycles*] and % difference for Industrial Test Programs

Program	Graphics		Railroad		Defense	
Method	best	worst	best	worst	best	worst
SA	309	2,602	389	23,466	848	71,350
ET	457	2,176	508	22,626	9,095	35,226
difference	32%	16%	24%	4%	N/A	N/A

For the computer graphics example, the shortest execution times determined vary by 32%, the longest by 16% between SA and ET. ET discovered the extremes within 24 generations with a population size of 50 individuals. The creation of additional 76 generations with 3800 test data sets did not produce any longer or shorter execution times.

For the railroad control technology example the population size for ET was increased to 100 individuals because of the complex input of the test object. The shortest and longest execution time differ by about 24% and 4%, respectively. Therefore, the worst-case execution time of this example can already be defined very accurately after 100 generations.

The defense electronics program has 843 input parameters. Therefore, the population size in this experiment was also set to 100. The number of generations was increased to 300 because of the large range of the variables and the large number of input parameters. The longest execution time is about half of the value calculated by SA. The fastest execution time found after 300 generations differs significantly from the results obtained by SA. A closer analysis of potential 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. Hence, the current numbers cannot be directly compared. We are currently trying to isolate these effects for the Quantify tool to allow a proper comparison with SA.

Table 4. Execution Times [*cycles*] and % difference from actual time for Standard Benchmarks

Program	Matrix				Sort			
Method	best		worst		best		worst	
SA	8,411,378	18%	15,357,471	16%	16,003	24%	12,268,014	3%
actual	10,315,619		13,190,619		20,998		11,872,718	
ET	12,050,569	17%	13,007,019	1%	20,998	0%	11,826,117	1%

The first example in Table 4 represents 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 results if all elements of both matrices are set to the largest permissible value (8095). The shortest run time 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 500,000 parameter values have to be generated. For this reason, the number of generations for this example was increased to 800. The values for SA and ET differ by about 17% from the actual measurements, except for the worst-case time of ET, which was within 1% of 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. Conversely, the shortest run time results, of course, from the sorted list. When searching for the longest execution times, the evolutionary test was limited to 500 generations with 300 individuals. The prediction of the shortest execution time by SA differed by 24% from the actual value. The remaining values found by SA and ET are within 3% or less of the actual measurements. In fact, for ET the actual shortest execution time was found after 746 generations. Figure 6 shows the successive reduction in execution times found up to the 746th generation, where the actual minimal time is found. ET manages the generation of a sorted list merely on the basis of the execution times analysis. The shortest execution time determined by SA differs by 24% from the absolute minimum.

While Figure 6 represents the development of the shortest execution times found over generations, Figure 7 gives an insight into the structure of the individuals. The variable values of all 300 individuals of the generations 20, 300 and 1000 are displayed. The list elements 1-500 are indicated on the x-axis, the 300 different

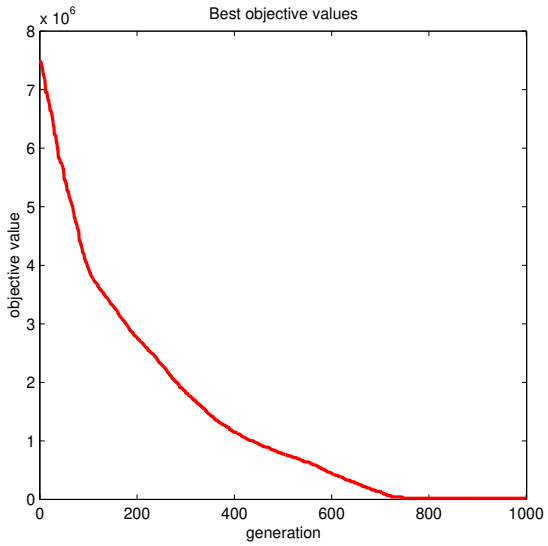

Figure 6. BCET over 1000 Generations for Sort

values for each list element are represented on the y-axis. The population in generation 20 does not yet show any structure. The variable values of all list elements vary in the valid range. This changes in generation 300. Here the solution begins to emerge. All individuals of the population begin to approach a sorted list. The deviation of the variable values is already relatively low for the first 160 elements. For the remaining list elements it is more pronounced. In generation 1000, the entire population has converged against the sorted list. Isolated deviations from the sorted list result from mutation of single individuals.

Since the bubblesort algorithm is an ordering problem, another mutation-operator was used than in the other examples, which can be considered as pure parameter optimizations. The mutation is not implemented by the accidental change of single variable values but is founded on the exchanging of two variables of an individual that have been determined randomly. If this particular mutation operator was not applied and if the example sort was only considered as a parameter optimization problem without knowing its functionality, the sorted list could only be found as result for the shortest execution time after about 3000 generations. On the other hand, the number of generations before finding the BCET can be reduced by the additional use of particular recombination operators for ordering problems. The efficiency of ET is consequently determined by the suitability of operators chosen for the test object. For complex test objects that show elements of ordering as well as parameter optimization problems extended evolutionary algorithms will be combined in subpopulations implementing different search strategies [48].

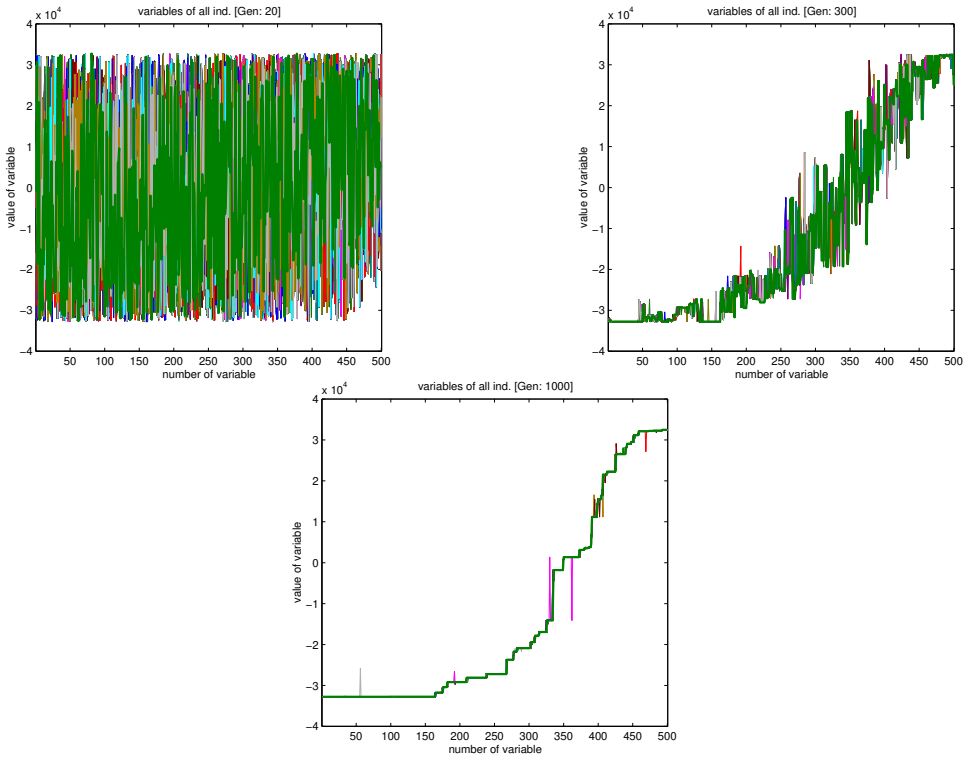

Figure 7. Visualization of Change in Data over Generations for Sort

6. Discussion

SA is a well-established method for the verification of timing constraints used during system design whereas ET is a new method for the examination of temporal behavior. At present, ET is the only test method specialized in testing temporal behavior that does not depend on the application of formal development methods. ET is usually applied in the testing phase. Although SA and ET are generally used in different phases of system development, a comparison of both methods is interesting because they both aim at estimating the worst-case and best-case execution times for systems.

The experiments presented in this paper show that evolutionary testing is comparable to static analyses with respect to the accuracy of estimating the longest and shortest execution time. For those sample programs whose actual worst-case and best-case execution times were known, it could even be shown that the extreme execution times found by evolutionary testing are closer to the actual execution times than the timing predictions of static analysis. These promising results illustrate that ET should be able to detect temporal errors in real-world applications and it

can thus contribute significantly to ensuring quality and dependability of real-time systems.

Our experiments also showed that an early combination of ET and SA is helpful during system design in order to predict execution times of tasks as precisely as possible. The measurements of the last section show that SA and ET bound the actual execution times from opposite ends.

In about half of the experiments, the actual execution times were bounded within ±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 ±10%, which may still yield useful results for schedulability analysis. For the multiplication of matrices and the defense example larger variations were detected, indicating that both approaches need further investigation to improve their precision.

Figure 8 gives an example of the bounds determined for the multiplication of matrices. The actual WCET is bounded below and above by ET and SA, respectively, and vice versa for the BCET.

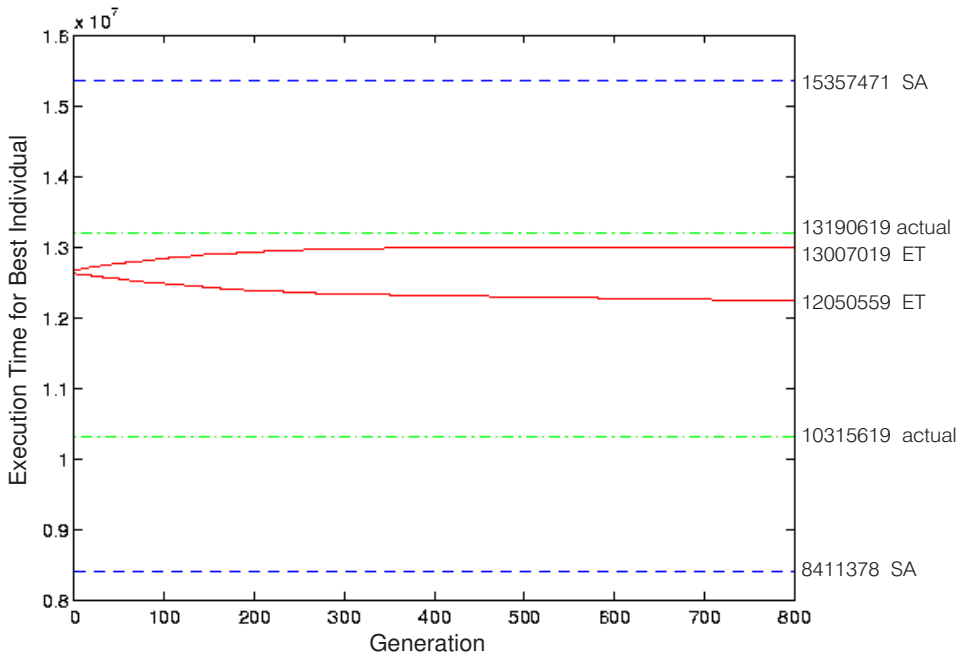

Figure 8. Extreme Execution Times Bounded by SA and ET for Matrix

The Figure also shows that ET stops making essential progress after only 350 generations and yielding a close worst-case estimate while the fitness value indicates that a close approximation of the best case is impossible without changing search parameters (*e.g.*, population size) or even search methods.

6.1. Further Qualities of SA and ET

The executed experiments illustrate some further qualities of SA and ET that will be compared in the following section.

The overhead for estimating the extreme execution times differs for both approaches (see Table 5). ET requires the execution of the system under test 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, additional test objects have to be isolated and tested separately.

Since test objects for ET are considered as black boxes, ET is universally applicable in all test phases for testing temporal behavior (unit test, task test, integration test and system test), independent of the programming language used, the internal architecture (sequential, parallel, distributed) of the system under test and the characteristics of the underlying target hardware (caching, pipelining). The only prerequisites for ET are the availability of the target hardware and the input specification of the test object. ET 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. The SA approach requires knowledge about loop frequencies and information about cache configurations. In addition, the cycle-level behavior of the actual hardware that has to be supplied in configuration files. New hardware features have to be implemented in the simulator, which limits the portability of SA. Some features with a small impact are currently not covered, *e.g.,* DMA.

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.

In summary, the ET approach cannot provide safe timing guarantees. Since it measures the actual, running system and tries to achieve the optimum solution by the random exchange of information between increasingly fit samples (combination) and the introduction of independent random change (mutation), it is not predictable if and when certain input situations will be found, which might be especially important for the run-time behavior of the system under test. Therefore, we cannot prove that the timings found for the three real-world examples are the

Table 5. Comparison of Properties of SA and ET

property	Static Analysis	Evolutionary Testing
timing	safe	actually observed
overhead	program size & structure	test execution
time program parts	included (for free)	separate timing
portability	compiler, cycle info	actual timing method
interferences	smaller (e.g., DMA)	larger (caches, OS)
prerequisites	loop bounds, hardware specs	input specification
test method	white box	black box

longest and shortest possible values. The SA approach yields conservative estimates that safely approximate the actual execution times. However, it is not possible to indicate how close the timing predictions determined by SA are to the actual execution times. Furthermore, it is difficult to verify the correctness of the models that the SA toolset uses, *i.e.,* the approach depends on the availability of detailed and correct data about the cycle-level behavior of the actual hardware. We therefore 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 in system design to bound the extreme execution times more precisely. ET may also suffice if missed deadlines can be tolerated sporadically. Thus, for hard real-time environments the WCET for schedulability analysis should be derived from SA whereas soft real-time environments may choose between SA and ET, or even the mean between SA and ET (see Table 6). The results of ET may also be used to adjust the results of SA if, for example, the WCET derived by SA is overly pessimistic and viable methods exist to safely quantify the amount of pessimism. 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. We think that both methods have potential for further improvement.

Table 6. Recommendation for Deriving Extreme Execution Times

Design Phase		
methods	hard real-time	soft real-time
static analysis	mandatory	optional
evolutionary testing	optional	sufficient
Testing Phase		
evolutionary testing	mandatory	optional

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. Static analysis predicts the worst-case and best-case execution times by analyzing execution paths and simulating processor characteristics without executing the program. Evolutionary testing is a new test method that approximates the extreme execution times within several iterations. By executing the test object with selected test data and measuring the execution times, the inputs are guided yielding gradually tighter predictions of the extreme execution times with each iteration.

Both methods were implemented and evaluated for a number of test programs with respect to their prediction of the extreme execution times. The experiments showed that the accuracy of timings determined by evolutionary testing is of the same magnitude as that of static analysis. It was shown for those sample programs whose longest and shortest execution times were known, that the extreme execution times determined by evolutionary testing were closer to the actual execution times for the worst and best case than the timing predictions of static analysis.

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.

If static analyses are already supplemented by evolutionary testing during system design, very good predictions of the extreme execution times of tasks can be determined, because these methods are complementary 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. Therefore, we recommend that the worst-case execution time for schedulability analysis be derived from static analysis for hard real-time environments. Evolutionary testing may be used additionally to bound the extreme execution times more precisely. Soft real-time environments may choose between static analysis and evolutionary testing. Furthermore, we suggest the general use of evolutionary testing in the testing phase to find errors in the temporal behavior of the actual, running system and to increase the confidence in its temporal correctness.

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 integrate the Gnat/Gcc compiler. Current work on evolutionary testing focuses on the development of suitable stopping criteria to terminate the test. One possibility is to measure the convergence state of the population [36]. Additionally, 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 seems to be path coverage because of the strong correlation between the program's control flow, the execution of its statements and the resulting execution times. Coverage information can also be used to control the evolutionary test during test execution (grey-box testing). For example, survival of individuals, who cover a new program path, can be guaranteed for several generations in order to examine whether passing the new program path leads to longer or shorter execution times. A suitable instrumentation of test objects could furthermore direct the evolutionary test to find new program branches or paths (see [18]).

Acknowledgments

Chris Healy provided support for the timing analyzer.

References

1. T. Baeck. *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, 1996.
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. R. Bodik, R. Gupta, and M. Soffa. Interprocedural conditional branch elimination. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 146–158, June 1997.
5. V. Braberman, M. Felder, and M. Marre. Testing timing behavior of real-time software. In *International Software Quality Week*, May 1997.
6. 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.
7. J. V. Busquets-Matraix. Hybrid instruction cache partitioning for preemptive real-time systems. In *EuroMicro Workshop on Real-Time Systems*, pages 56–63, June 1997.
8. D. Clarke and I. Lee. Testing real-time constraints in a process algebraic setting. In *International Conference on Software Engineering*, April 1995.
9. 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.
10. C. G. Davis. Testing large, real-time software systems. In *Software Testing, Infotech State of the Art Report*, volume 2, pages 85–105, 1979.
11. C. Ferdinand and R. Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2/3):131–181, November 1999.
12. M. Grochtmann, K., and Grimm. Classification trees for partition testing. *Software Testing, Verification & Reliability*, 3(2):63–82, 1993.
13. M. Grochtmann and J. Wegener. Evolutionary testing of temporal correctness. In *Quality Week Europe*, November 1998.
14. C. A. Healy, R. D. Arnold, F. Mueller, D. Whalley, and M. G. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1):53–70, January 1999.
15. 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, December 1995.
16. W. S. Heath. *Real-Time Software Techniques*. Van Nostrand Reinhold, 1991.

17. Y. Hur, Y. H. Bae, S.-S. Lim, B.-D. Rhee, S. L. Min, C. Y. Park, M. Lee, H. Shin, and C. S. Kim. Worst case timing analysis of RISC processors: R3000/R3010 case study. In *IEEE Real-Time Systems Symposium*, pages 308–319, December 1995.
18. B. Jones, D. Eyres, and H. Sthamer. A strategy for using genetic algorithms to automate branch and fault-based testing. *The Computer Journal*, 41(2):98–107, 1998.
19. 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.
20. D. B. Kirk. SMART (strategic memory allocation for real-time) cache design. In *IEEE Real-Time Systems Symposium*, pages 229–237, December 1989.
21. 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*, December 1996.
22. Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *IEEE Real-Time Systems Symposium*, pages 298–397, December 1995.
23. 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, December 1996.
24. 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.
25. S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, and C. S. Kim. An accurate worst case timing analysis for RISC processors. In *IEEE Real-Time Systems Symposium*, pages 97–108, December 1994.
26. T. Lundquvist and P. Stenström. An integrated path and timing analysis method based on cycle-level sympolib execution. *Real-Time Systems*, 17(2/3):183–208, November 1999.
27. 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, September 1992.
28. F. Mueller. *Static Cache Simulation and its Applications*. PhD thesis, Dept. of CS, Florida State University, July 1994.
29. F. Mueller. Compiler support for software-based cache partitioning. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pages 137–145, June 1995.
30. 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.
31. F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2/3):209–239, May 2000.
32. F. Mueller and J. Wegener. A comparison of static analysis and evolutionary testing for the verification of timing constraints. In *IEEE Real-Time Technology and Applications Symposium*, pages 179–188, June 1998.
33. F. Mueller and D. B. Whalley. Avoiding conditional branches by code replication. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 56–66, June 1995.
34. Mühlenbein and Schlierkamp-Vossen. Predictive models for the breeder genetic algorithm: I. continuous parameter optimization. *Evolutionary Computation*, 1(1):25 – 49, 1993.
35. G. J. Myers. *The Art of Software Testing*. John Wiley & Sons, 1979.
36. M. O'Sullivan, S. Voessner, and J. Wegener. Testing temporal correctness of real-time systems – a new approach using genetic algorithms and cluster analysis. In *European International Conference on Software Testing, Analysis, and Review*, 1998.
37. C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–61, March 1993.
38. H. Pohlheim. GEATbx: Genetic and evolutionary algorithm toolbox for use with matlab - documentation. Technical report, Technical University Ilmenau, Germany, 1996.
39. P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1(2):159–176, September 1989.

40. P. Puschner and R. Nossa. Testing the results of static worst-case execution time analysis. In *IEEE Real-Time Systems Symposium*, pages 134–143, December 1998.
41. J. Rawat. Static analysis of cache analysis for real-time programming. Master's thesis, Iowa State University, 1995.
42. Richard M. Stallman. *Using and Porting GNU CC*. Free Software Foundation, Inc., 2.7.2 edition, November 1995.
43. 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.
44. H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise wcet prediction by separated cache and path analyses. *Real-Time Systems*, 18(2/3):149–171, May 2000.
45. New York University. The Gnu NYU Ada Translator (GNAT). Available by anonymous FTP from cs.nyu.edu.
46. J. Wegener and M. Grochtmann. Verifying timing constraints of real-time systems by means of evolutionary testing. *Real-Time Systems*, 15:275–298, 1998.
47. J. Wegener and R. Pitschinetz. Tessy - yet another computer-aided software testing tool? In *European International Conference on Software Testing Analysis & Review*, October 1994.
48. J. Wegener, H. Pohlheim, and H. Sthamer. Testing the temporal behavior of real-time tasks using extended evolutionary algorithms. In *European International Conference on Software Testing, Analysis, and Review*, November 1999.
49. 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.
50. R. T. White, F. Mueller, C. Healy, D. Whalley, and M. G. Harmon. Timing analysis for data and wrap-around fill caches. *Real-Time Systems*, 17(2/3):209–233, November 1999.
51. D. Whitley. The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In *International Conference on Genetic Algorithms*, pages 116–121, 1989.
52. A. Wolfe. Software-based cache partitioning for real-time applications. In *Workshop on Responsive Computer Systems*, 1993.