# Evolutionary Testing for Timing Analysis of Parallel Embedded Software

Muhammad Waqar Aziz and Syed Abdul Baqi Shah
Science and Technology Unit, Umm Al-Qura University, Kingdom of Saudi Arabia

**Abstract:** *Embedded real-time software must be verified for their timing correctness where knowledge about the Worst-Case Execution Time (WCET) is the building block of such verification. The WCET of embedded software can be estimated using either static analysis or measurement-based analysis. Previously, the WCET research assumes sequential code running on single-core platforms. However, as computation is steadily moving towards using a combination of parallel programming and multicore hardware, necessary research in WCET analysis should be taken into account. While focusing on the measurement-based analysis, the aim of this research is to find the WCET of parallel embedded software by generating the test-data using search algorithms. In this paper, the use of a meta-heuristic optimizing search technique-Genetic Algorithm is demonstrated, to automatically generate such test-data. The search-based optimization used yielded the input vectors of the parallel embedded software that cause maximal execution times. These execution times can be either the WCET of the parallel embedded software or very close to it. The process was evaluated in terms of its scalability, safety and applicability. The generated test-data showed improvements over randomly generated data.*

**Keywords:** *Embedded real-time software, worst-case execution-time analysis, measurement-based analysis, end-to-end testing, genetic algorithm, parallel computing.*

*Received May 24, 2016; accepted February 3, 2017*

## 1. Introduction

Parallel software is now increasingly used in embedded systems to fulfil the growing demands of high performance real-time applications and to optimally utilize the available multicore hardware [18]. This consequently has put special demands on the design and testing of Parallel Embedded Software (PES) [14]. Generally, the temporal testing of embedded systems is performed through timing analysis, where Worst-Case Execution Time (WCET) is a central parameter [26]. The worst-case bounds can be derived using static timing analysis [10] applied on software and hardware models of the system, without executing the program and requiring any inputs. However, it is very difficult to develop and analyze these models for parallel systems. For instance, the inter-thread interferences among shared resources, e.g., L2 caches are hard to analyze statically [28]. Alternatively, dynamic analysis or measurement-based methods can be used for better estimating the program execution time. Measurement-based methods execute a task or its parts, to measure its execution time, on the given hardware or simulator using a large set of inputs [26]. These methods provide either an evidence of the successful operation of the software or highlight the errors that cannot be detected by static analysis. Therefore, measurement-based methods are widely used in the industry and are well suited for soft real-time systems, which is the target of this research work.

A clear disadvantage of measurement-based methods is that these methods cannot guarantee safe (not underestimated) WCET, until a significantly large number of appropriate input values are exercised [1]. To give a reliable guarantee that the WCET has been encountered, the worst-case inputs[1] of a task need to be known. Generally worst-case inputs are unknown and hard to derive [26], due to the extremely large input domain. However, if searching the worst-case inputs from the set of all possible inputs is considered as an optimization problem, a meta-heuristic optimizing search technique [16] such as a Genetic Algorithm (GA) can be utilized to automatically search the required data. To the best of our knowledge, there is a general lacking in search-based test data generation research for PES, as most of the research efforts of WCET analysis for multicore systems are focused on performance enhancing hardware features [12, 15, 21], and application [3, 17, 20] or programming models [8, 29, 30]. Therefore, the use of meta-heuristic optimizing search technique to generate the worst-case inputs for WCET analysis of PES needs to be explored.

The objective of this work is to use evolutionary testing coupled with the notion of WCET coverage to yield long execution times that are the WCET or close to it with appropriate levels of confidence. Evolutionary testing [24] is an iterative testing process that automatically searches for the test data, which produce long execution times. The inputs are guided by executing the test object dynamically and measuring the execution times, yielding gradually

[1]Worst-case inputs refer to such inputs of a program which produce the maximum execution time.

tighter predictions of the long execution times [24]. In this paper, the well-known form of evolutionary algorithms, i.e., a GA is used to automatically find the worst-case inputs of PES. The use of GA is supposed to search the inputs which produce long execution times in each generation, leading towards a data set of worst-case inputs. We have applied the steps of GA to generate test data that is ample for producing long execution times, which can be regarded as the suitable candidates for being the WCET. To stop testing, the WCET coverage metrics are also proposed in this work, which are used as the stopping criteria when a satisfactory level of confidence about the obtained estimations is encountered.

While applying GA, the end-to-end execution time of PES was considered as the fitness function. This is in accordance with the current industry practice of performing end-to-end testing and applying some engineering wisdom to derive the WCET. For automatic test-data generation, the PES was executed repeatedly with sets of inputs for a number of generations. The process started by generating *k* random input vectors initially (first generation) and obtaining their timing information through measurement. Then, the generated *k* random inputs with the timing information were used to produce the next *k* inputs (second generation) and so on. The number of generations was guided by the testing coverage metrics, proposed in this work. The ParMiBench benchmark suite [11] was selected as an example PES, the details are provided in the next section. The end-to-end execution time was measured in number of processor clock cycles. It was obtained by gathering the execution traces of the parallel program using Gem5 architecture simulator [2] as the execution platform.

It is important to note that the application of meta-heuristic search technique GA allows the classification of this work as a search-based test data generation method. Therefore, it should not be confused with static timing analysis methods, which use program flow analysis and micro-architectural modeling for estimating WCET of parallel programs. Consequently, analyzing the hardware related issues such as the contention effects caused due to access of shared hardware resources, e.g., the cache hierarchies, the memory bus and the memory controller, are out of the scope of this work.

The layout of the paper is as follows: section 2 provides the background of GA and of the example PES used for experimentation. Section 3 describes the process how evolutionary testing is performed using GA, in this work, to approach the problem of WCET analysis. Section 4 reports the experimental setup and the results obtained from the experiment. The evaluation of this work is provided in section 5. Section 6 describes related work in measurement-based WCET analysis of PES. Section 7 contains concluding remarks and directions for the future work.

## 2. Background

### 2.1. Genetic Algorithm

GA uses the concept of natural evolution to reach the desired solution from a given huge search space. It mimics the process of natural selection and chooses the best from one generation to produce the next generation and attempts to reach the solution much faster than otherwise. In contrast to other search-based optimization algorithms, such as Hill Climbing and Simulated Annealing, GA is a global search approach that considers many solutions in the search space at once. Thus, it avoids convergence towards local optima. This section introduces the steps of GA, as also applied in this research.

In GA jargon, individuals or chromosomes (refer to potential solutions in the search space) are usually random guesses to the solution of a problem. Collectively, the current set of individuals under consideration forms the current population. To apply GA, an initial population of individuals needs to be defined, which is then evolved across a number of generations. Care should be given to maintain diversity in the population so that premature convergence towards a sub-optimal solution can be prevented. The fitness of the individuals is a problem-dependent value that specifies the goodness of an individual in solving the problem at hand. The selection of an individual for the next generation depends on its fitness value, i.e., each individual in the population is evaluated by calculating its fitness. In other words, the fitness value is used to select the best of any generation to 'mate' them in order to produce the new generation. The fitness value is generated in GA through fitness function. A selection strategy is applied, to the individuals of a population in a given generation to decide which ones are allowed to proceed to the next generation.

The evolution of the population involves the exchange of genetic material between the individuals through crossover operation. Traditionally, this is achieved by choosing a point along two bit strings at random and swapping the tails. As a result, each individual is evolved into two offspring which then go into a mutation process. Mutation is a random change in the genetic material of a single individual value, which is performed in GA to scape a local optimum. In this way, the mutation operation diversifies the search into new areas of the search space. Mutation is achieved by picking a bit at random and flipping its value. Similarly, the next generation of the population is chosen and the new individuals are evaluated for fitness. Like this, the fitness function guides the search to promising areas of the search space. This cycle is iterated, until the optimum is achieved or a stopping criterion is fulfilled such as some fixed number of fitness evaluations is exhausted.

## 2.2. ParMiBench Benchmark Suite

ParMiBench [11] suite is designed to evaluate the performance of embedded multi-core systems. It includes benchmarks from various domains of the embedded applications such as control and automation, networks, offices, and security. ParMiBench was selected as it is an open source parallel version of a subset of MiBench [9]-a well establish benchmark suite, many of whose benchmarks appear to be suitable candidates for WCET analysis [6]. The coarse-grained task decomposition, used in ParMiBench, reduces the synchronization and communication among threads. Still the benchmark is a decent representative of PES, especially in the scarcity of any publicly available application, which is both embedded and parallel. Conversely, considering an application that is rich in parallelism (if any exist for embedded system), is very hard to analyze even for the domain experts.

We have selected the Stringsearch benchmark from the suite, which is related to searching a token (string placed in pattern file) from a text file. This selection is because string search benchmark fulfills the requirements needed to apply a search-based optimization technique: Representation (i.e., it is capable of being encoded to allow manipulation by the search algorithm) and fitness function (i.e., it allows the definition of a problem-specific fitness function that guides the search). However, the string search is inappropriate to be used for evolutionary testing with its existing data set, as it contains duplicate values. Since generating new pattern files from this data was useless, we developed a new data set (consisting of new pattern and text files) for Stringsearch.

## 2.3. WCET Analysis Using Meta-Heuristic Search

This work focuses on automatic test data generation for PES, which yield long execution times that are the WCET or close to it. In this regard, the use of a GA is proposed to heuristically search the inputs that will cause the PES to execute for the longest period of time. To this end, the steps of GA, as explained in the previous section, were applied to string search benchmark-an example PES. The process, displayed in Figure 1, is applied as follows.

In the initialization step, an initial population was generated by randomly picking up 50% tokens from the newly created text file, while the remaining tokens were randomly generated. This was done to get an unbiased population, which genetically represents the solution domain. The initial population consisted of one hundred pattern files (individuals). The fitness of each individual in the initial population was evaluated in the end-to-end calculation step. In the first generation, the end-to-end time was calculated for all the individuals present in the initial population. However, in all the rest of generations, it was calculated for the new individuals of the next generation only. The end-to-end execution time was calculated automatically by a Java application, as depicted in Figure 2, using the execution traces obtained from the Gem5 simulator by executing the parallel program. The end-to-end execution time was considered due to the inconsistent execution information of the individual threads, we observed across several runs.

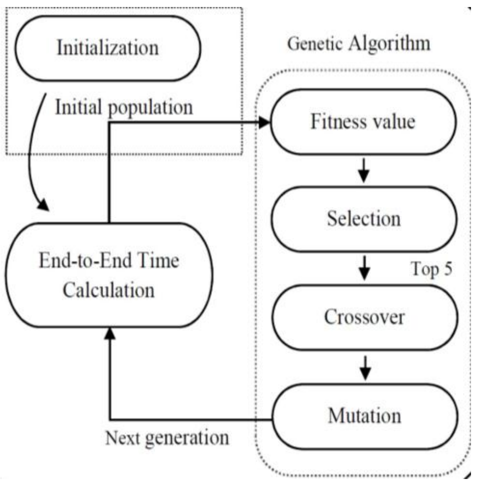

Figure 1. GA based test data generation for timing analysis of parallel embedded software.

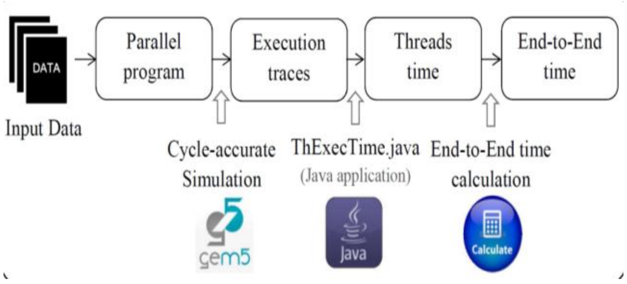

Figure 2. The process of calculating the end-to-end execution time.

As already mentioned, the execution time of the PES was used as a fitness value, in this work. To reveal longer execution times, the search attempts to maximize the fitness function. It means that the longer the end-to-end execution time, the higher would be the fitness of an individual to be selected for the next generation. As a result, the inputs with higher fitness values were selected in each generation, eventually leading towards inputs which cause long execution times. It is worth mentioning here that cache hits or misses, thread conflicts or any other parameters were not considered to evaluate the fitness. Because considering these parameters is out of scope of this research and is a huge research in its own, that requires more effort and time. In addition, Gem5 simulator does not provide much information about these parameters, which can be useful at this level.

During the selection step, only five individuals were selected, as selecting more individuals would take extra time in the remaining steps and also in calculating their fitness in the next generation. A rank based selection was performed by sorting the individuals based on their fitness values. To produce a new generation, a crossover was made among the top five selected individuals, by merging two lists of tokens picked up from randomly selected pattern files. This crossover resulted in ten new individuals. In crossover, one part of a file was concatenated with another part of the second file, where the size of selected parts may not be the same, e.g., 2 tokens picked from one file were concatenated with 62 tokens from the other file. This selection was based on cutting one individual at a random location and concatenating it with the remaining part of the second individual cut at the same location. The mutation process was applied to the results of crossover to produce and further improve the next generation. It was done by randomly picking a token from a pattern file and replacing its any letter with another random character. This process was repeated until the defined percentage of mutation was achieved. In this way, the next generation of 10 individuals was produced. The above process was repeated for a set of ten individuals, which was reproduced after each generation, until the stopping criteria were met.

With the above modeled parameters, a Java application was developed that implemented GA in the experiment. The implementation of GA, as used in this work, is presented in Algorithm 1. The evolution continues until an optimum is found that solves the problem adequately, or a stopping condition has reached (e.g., a certain number of generations is reached). In this work, we developed coverage metrics to stop testing; by which one can state to a satisfactory level of confidence that the WCET has been covered or that the probability of the WCET having not been covered is infinitely small. The coverage metrics were defined in terms of:

1. *Data saturation*: there is no optimization in the given population for a fixed number of generations (e.g., for 10 generations).
2. *Threshold value*: an initially defined threshold value has reached.
3. *Best individual saturation*: when the best individual in the population ceases to improve for a certain number of generations.
4. *Number of generations*: In case criterion (1) and (3) do not meet, the evolutionary testing would be stopped after a predefined number of generations. This number of generations was decided based on the time taken by the available hardware to produce one generation and the available time for the experiment.

## 3. Experimentation

### 3.1. Experimental Setup

We used a state-of-the-art computer architecture simulator (Gem5), to model a multicore system. The configuration of Gem5 used in this experiment included four cores of ARM detailed architecture (ARMv7-A ISA based) with default size of L2 cache (2MB) and 256 MB of Memory. Gem5 was selected as it is a modular platform and a cycle-accurate simulator that provides full-system simulation to execute a program in the operating system environment, which is our research interest. In this work, ARM embedded Linux (AEL) was used as the guest operating system contained in a disk image. In addition, Gem5 supports several commercial Instruction Set Architectures (just as ARM, ALPHA), CPU types, cache levels, memory and other components, which make it more powerful than other similar simulators, such as SimpleScalar.

*Algorithm 1: Implementation of GA for generating test data to exercise long execution times close to WCET*

```
1:  Max_Gen = Maximum number of generations of GA to be considered
2:  Num_Top_Fits = Number of top fits to be picked during selection
3:  For Gen:0 to Max_Gen
        // To run a loop for a given number of generations.
4:  Step 1: Read Fitness Values Table
        // A function call to read the fitness values and
        // their respective input file numbers from a file
5:  Step 2: Select Num_Top_Fits Values
        // To pick up top 'num_top_fits' values from the fitness values table
        // Selector Function takes a table of fitness values with their respective pattern file numbers and uses a 'Sorter' method to perform 'Sorting' w.r.t. fitness value.
        // 'Sorter', then returns a table of only top N values, specified as top fits. Sorter Function performs sorting of an unsorted vector string based upon the specified column.
6:  For i:0 to Num_Top_Fits
        // Read the input files with top fitness, one by one, in a loop
7:  Step 3: Read respective input files
8:  End
9:  Step 4: Perform CrossOver
        // Perform 'Cross Over' to produce a new generation of Genomes.
        // This method performs 'Cross Over' amongst the top selected pattern files
10: Step 5: Perform Mutation (ftn) to get a new Generation
        // Perform 'Mutation' and the new generation is ready for use.
11: Store the new generation to input test-data files
        // Create a new directory to save the newly generated chromosomes into test-data files.
        // Write the newly generated chromosomes into a files using a 'for' loop.
12: Run Simulation to compute fitness
        // Call a System Command Runner class to start the simulation
13: End
```

Moreover, Gem5 is widely used by the computer architecture research community to run the simulations. We have used it to execute the benchmarks for collecting the time-stamped execution traces.

In order to perform the WCET analysis using GA, a simulation needs to be executed hundreds of times, where each simulation takes tens of minutes to complete, making it too long to repeat this simulation sequentially. To solve this problem, we considered to run multiple simulations in parallel, each using a separate instance of Gem5 simulator. We had Xeon processor with twelve logical cores and enough memory (48GB) available in our machine to accommodate multiple instances of simulator running in parallel. Therefore, 6, 8, 10, and 12 instances of full system simulation were tested to run in parallel. It turned out that 10 instances in parallel gave better timing trade-off than other options. However, by running 10 instances of Gem5 in parallel increased the complexity of the experimental setup over its sequential execution, in terms of involved shell scripts, disk images and output directories.

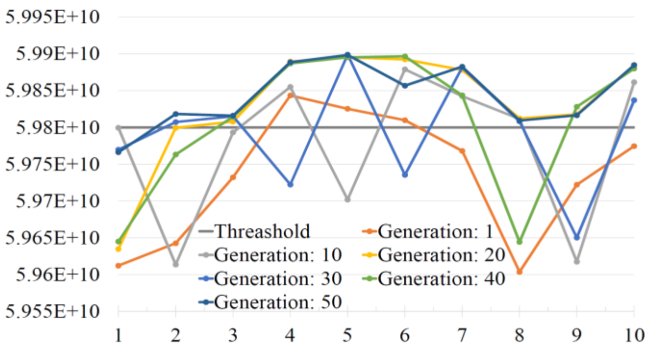

Figure 3. End-to-End execution times in different generations for the input pattern files with five characters long tokens.

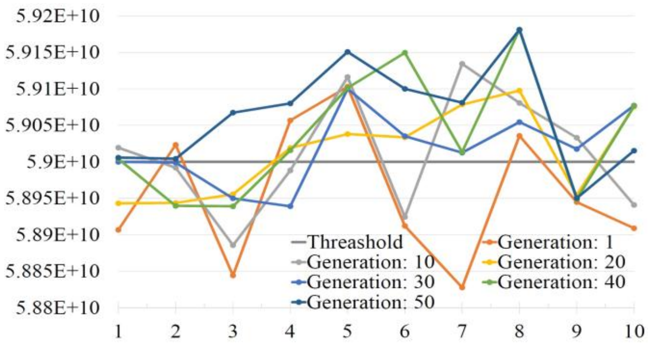

Figure 4. End-to-End execution times in different generations for the input pattern files with ten characters long tokens.

## 3.2. Experimental Results

To observe the effects of applying GA using visual representation, graphs of the calculated end-to-end times were plotted for each generation. In the graphs (Figures 3, 4, 5, and 6), the length of the execution time is represented on the vertical axis in terms of CPU ticks, whereas the ten individuals of each generation are represented horizontally, as shown in Figure 3. Following the proposed coverage metrics criteria, the evolution testing was performed for fifty generations. This was due the absence of data and best individual saturations. A threshold value was defined to analyze the improvement in the inputs over all generations. The improvement was measured by counting the number of individuals (input pattern files, in this case) taking greater or equal time than the defined threshold value. From the plotted graph (Figure 3), it can be observed that by using GA an overall improvement was achieved in the measured end-to-end time. For instance, in first generation there are only three individuals which caused the program to execute for a period longer than the defined threshold. In comparison, the number of individuals increased to nine in 50th generation. This increase in the number of individuals, after applying several generations, proved that the generated inputs were taking longer times. A sudden decrease was also observed in some of the values, as can be seen in Figure 3. This is due to the very nature of GA where results can degrade even after reaching to an improved position. However, an overall improvement is achieved across 50 generations.

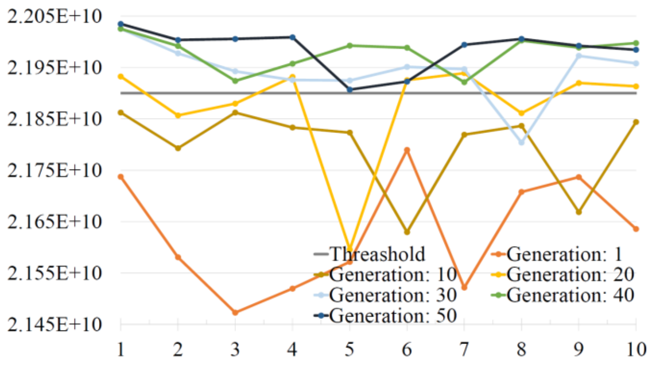

Figure 5. End-to-End execution times in different generations for the input files of Dijkstra benchmark.

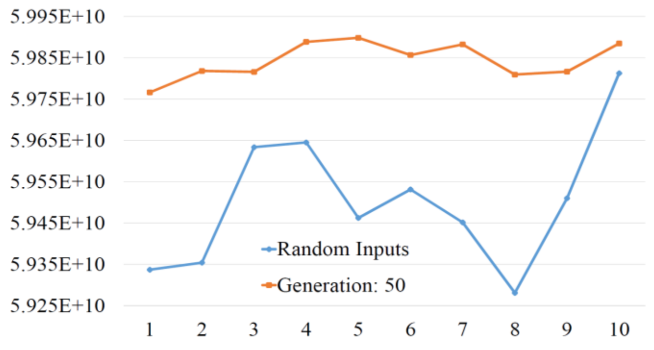

Figure 6. Comparison of randomly generated inputs with the inputs generated through GA.

## 4. Evaluation

The presented work was evaluated in terms of its scalability, safety, applicability, and improvements over the randomly generated inputs. The details are provided below.

### 4.1. Scalability

In order to evaluate the scalability of the proposed method, the experiment was repeated for the same benchmark with inputs of different lengths. To this end, the complete process was re-performed for tokens with a length of 10 characters; compared to the original experiment where five characters long input tokens were used. The result produced, in different generations of 10 characters length token, is presented in Figure 4. It can be observed that the number of individuals increased the threshold value with the number of generations. Although the number of files slightly increased and decreased due to the nature of GA, an overall steady increase in the number of files was observed after 50 generations.

### 4.2. Safety

To evaluate the safety, the measurement-based derive facts were compared with the static analysis of the Stringsearch benchmark. For that the flow analysis of the Stringsearch was performed to statically analyze the inputs that would maximize the number of run-time steps of the program. The analysis focused the testing of particular sections of the code (e.g., loops) in order to trigger long execution times. One obvious reason for analyzing loops was that an embedded program spends most of its time in loops. More importantly, loop bound analysis is the most focused areas within the flow analysis research. Thus, the iterations of the specified loop based on the input data were tested. In static flow analysis, it was found that the number of loop steps decreased with the increase in the length of a token. This observation was in accordance with the difference felt in the ranges of the execution times for 5 and 10 characters during the measurement-based experiment (see, e.g., Figures 3 and 4).

### 4.3. Applicability

To further demonstrate the applicability of the proposed method, it was also applied to another benchmark of ParMiBench, i.e., Dijkstra. Dijkstra benchmark is related to finding the shortest path between nodes of a graph. This application and the improvements in the inputs, in terms of taking more time, proved that the proposed method is general enough to apply to other PES.

### 4.4. Improvements

During the experiment performed, a significant improvement in the execution times was not observed from one generation to the next. However, there was a substantial difference in the execution times of the worst-case inputs generated by using the proposed method, compared to those generated randomly. This difference can be seen in Figure 6, where the randomly generated inputs are compared with the worst-case generated inputs obtained after 50th generation.

## 5. Related Work and Discussion

Most of WCET-analysis research is performed for sequential software and single-core hardware. Recently, research on WCET analysis of sequential code running on multi-core processors has been a main focus. The work that has been done in this area so far can be divided into two parts:

1. Static hardware modeling for WCET analysis of multicore architectures [7, 27, 28, 31].
2. Design of analyzable multi-core computers that favor timing predictability over performance [5, 19, 22, 23].

Some research on parallel applications running on multi-core architectures is also available. For instance, the problem of analyzing the timing behavior of non-sequential software on a multi-core architecture is highlighted in [21].

In relation to this work, we found no related work to review, which deals with test data generation for parallel applications running on multi-core architectures using evolutionary testing. Previously, search-based techniques have been used to generate the input test-data [24, 25] and evolutionary search (more specifically GA) has been employed [4, 13] to find long execution times. In [24], a comparison between static analysis and evolutionary testing is presented, with the merits and demerits of each approach. However, all of these works target sequential real-time programs running on single core hardware. In contrast, we have used GA to generate test data for WCET analysis of parallel programs running on multi-core hardware. The fitness function used in this work considered the end-to-end time of a program, which was calculated from the execution times of individual threads (see Figure 2). This also included the overall thread execution and interaction time. This consideration has helped us to produce appropriate lists of tokens by using GA that maximized the fitness value.

Although heuristic based algorithms, in general, do not guarantee to generate improved guesses, some improvements have been observed in our case. It is therefore claimed that the proposed method has a high probability of finding the worst-case inputs of the programs which lie in the subset where GA will improve. It is important to mention here that the measurement-based methods generally do not guarantee safety, therefore these methods should be complemented with static timing analysis, if safety is required, e.g., for hard real-time systems. However, finding an absolute safe bound on the execution time is not required for most of the real-time systems which are soft in their majority. Moreover, the measurement-based methods can guarantee safety if the appropriate test data is used [24]. In this work, as the worst-case test data has been generated, a safe upper bound on the WCET can be guaranteed.

## 6. Conclusions

In this paper, the worst-case execution time of parallel embedded software is determined by generating the test-data. A genetic algorithm was used to heuristically search the inputs from a huge search space that would cause the parallel program to execute for the longest period of time. The input vectors were evolved using the largest end-to-end execution time as the fitness function. The results of this evolutionary testing showed that GA has significantly improved the execution times of inputs, i.e., those inputs were generated that would lead towards the WCET of parallel embedded software. The measurement-based analysis, performed in this work, is sufficient not only for soft real-time systems, but also for safety-critical systems as a safe upper bound on the WCET is guaranteed due to the generation of the appropriate test data. The process was demonstrated by its application in a parallel embedded benchmark suite-*ParMiBench*, where it was evaluated in terms of its scalability, safety and applicability. In the future, we aim to use a real-life, real-time application to evaluate our work. It is planned to consider a richer multi-objective fitness function in the future that might include thread conflicts, cache misses/hits and cache sizes.

## References

[1] Betts A., Bernat G., Kirner R., Puschner P., and Wenzel I., "WCET Coverage For Pipelines," Real Time Systems Research Group-University of York and Institute of Computer Engineering-Vienna University of Technology, Technical Report, 2006.

[2] Binkert N., Beckmann B., Black G., Reinhardt S., Saidi A., Basu A., Hestness J., Hower D., Krishna T., Sardashti S., Sen R., Sewell K., Shoaib M., Vaish N., Hill M., and Wood D., "The Gem5 Simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1-7, 2011.

[3] Ding Y. and Zhang W., "Multicore-Aware Code Co-Positioning to Reduce WCET on Dual-Core Processors with Shared Instruction Caches," *Journal of Computing Science and Engineering*, vol. 6, no. 1, pp. 12-25, 2012.

[4] Gross H., "An Evaluation of Dynamic, Optimisation-Based Worst-Case Execution Time Analysis," *in Proceedings of the International Conference on Information Technology: Prospects and Challenges in the 21st Century*, Kathmandu, 2003.

[5] Guan N., Stigge M., Yi W., and Yu G., "Cache-Aware Scheduling and Analysis for Multicores," *in Proceedings of the 17th ACM International Conference on Embedded Software*, Grenoble, pp. 245-254, 2009.

[6] Gustafsson J., Betts A., Ermedahl A., and Lisper B., "The Mälardalen WCET Benchmarks: Past, Present and Future," *in Proceedings of 10th Workshop on Worst-Case Execution Time Analysis*, Brussels, 2010.

[7] Gustavsson A., Ermedahl A., Lisper B., and Pettersson P., "Towards WCET analysis of Multicore Architectures Using Uppaal," *in Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, Dagstuhl, pp. 101-112, 2010.

[8] Gustavsson A., Gustafsson J., and Lisper B., "Toward Static Timing Analysis of Parallel Software," *in Proceedings of 12th International Workshop on Worst-Case Execution Time Analysis*, Dagstuhl, 2012.

[9] Guthaus M., Ringenberg J., Ernst D., Austin T., Mudge T., and Brown R., "Mibench: A Free, Commercially Representative Embedded Benchmark Suite," *in Proceedings of the 4th Annual IEEE International Workshop on Workload Characterization*, Austin, pp. 3-14, 2001.

[10] Heckmann R. and Ferdinand C., "Worst-Case Execution Time Prediction By Static Program Analysis," *in Proceedings of 18th International Parallel and Distributed Processing Symposium*, Santa Fe, pp. 26-30, 2004.

[11] Iqbal S., Liang Y., and Grahn H., "ParMibench-an Open-Source Benchmark for Embedded Multiprocessor Systems," *IEEE Computer Architecture Letters*, vol. 9, no. 2, pp. 45-48, 2010.

[12] Kästner D., Schlickling M., Pister M., Cullmann C., Gebhard G., Heckmann R., and Ferdinand C., "Meeting Real-Time Requirements with Multi-Core Processors," *in Proceedings of International Conference on Computer Safety, Reliability and Security, Berlin*, pp. 117-131, 2012.

[13] Khan U. and Bate I., "WCET Analysis of Modern Processors Using Multi-Criteria Optimization," *in Proceedings of 1st International Symposium on Search Based Software Engineering*, Windsor, pp. 103-112, 2009.

[14] Koziolek H., Becker S., Happe J., Tuma P., and Gooijer T., "Towards Software Performance Engineering for Multicore and Manycore Systems," *ACM SIGMETRICS Performance Evaluation Review*, vol. 41, no. 3, pp. 2-11, 2014.

[15] Liang Y., Ding H., Mitra T., Roychoudhury A., Li Y., and Suhendra V., "Timing Analysis of

Concurrent Programs Running on Shared Cache Multi-Cores," *Real-Time Systems*, vol. 48, no. 6, pp. 638-680, 2012.

[16] Mansour N., Awad M., and El-Fakih K., "Incremental Genetic Algorithm," *The International Arab Journal of Information Technology*, vol. 3, no. 1, pp. 42-47, 2006.

[17] Ozaktas H., Rochange C., and Sainrat P., "Automatic WCET Analysis of Real-Time Parallel Applications," *in Proceedings of 13th Workshop on Worst-Case Execution Time Analysis*, Paris, pp. 11-20, 2013.

[18] Pinho L., Quinones E., Bertogna M., Marongiu A., Carlos J., Scordino C., and Ramponi M., "P-Socrates: A Parallel Software Framework for Time-Critical Many-Core Systems," *in Proceedings of 17th Euromicro Conference on Digital System Design*, Verona, pp. 214-221, 2014.

[19] Pitter C. and Schoeberl M., "A Real-Time Java Chip-Multiprocessor," *ACM Transactions on Embedded Computing Systems*, vol. 10, no. 1, 2010.

[20] Potop-Butucaru D. and Puaut I., "Integrated Worst-Case Response Time Evaluation of Multicore Non-Preemptive Applications," PhD Dissertation, Institut National de Recherche en Informatique et en Automatique, 2013.

[21] Rochange C., Bonenfant A., Sainrat P., Gerdes M., Wolf J., Ungerer T., Petrov Z., and Mikulu F., "WCET Analysis of A Parallel 3D Multigrid Solver Executed on the Merasa Multi-Core," *in Proceedings of OASIcs-OpenAccess Series in Informatics, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik*, Dagstuhl, pp. 90-100, 2010.

[22] Rosen J., Andrei A., Eles P., and Peng Z., "Bus Access Optimization for Predictable Implementation Of Real-Time Applications on Multiprocessor Systems-on-Chip," *in Proceedings of 28th IEEE International Real-Time Systems Symposium*, Tucson, pp. 49-60, 2007.

[23] Ungerer T., Cazorla F., Casse H., Uhrig S., Guliashvili I., Houston M., Kluge F., Metzlaff S., Mische J., Sainrat P., Bernat G., Petrov Z., Rochange C., Quinones E., and Gerdes M., "Merasa: Multicore Execution of Hard Real-Time Applications Supporting Analyzability," *IEEE Micro*, vol. 30, no. 5, pp. 66-75, 2010.

[24] Wegener J. and Mueller F., "A Comparison of Static Analysis and Evolutionary Testing for the Verification of Timing Constraints," *Real-Time Systems*, vol. 21, no. 3, pp. 241-268, 2001.

[25] Wegener J., Sthamer H., Jones B., and Eyres D., "Testing Real-Time Systems Using Genetic Algorithms," *Software Quality Journal*, vol. 6, no. 2, pp. 127-135, 1997.

[26] Wilhelm R., Engblom J., Ermedahl A., Holsti N., Thesing S., Whalley D., Bernat G., Ferdinand C., Heckmann R., Mitra T., Mueller F., Puaut I., Puschner P., Staschulat J., and Stenstrom P., "The Worst-Case Execution-Time Problem-Overview of Methods and Survey of Tools," *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 3, 2008.

[27] Wu L. and Zhang W., "Bounding Worst-Case Execution Time for Multicore Processors Through Model Checking," *in Proceedings of 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, Stockholm, pp. 17-20, 2010.

[28] Yan J. and Zhang W., "WCET Analysis for Multi-Core Processors with Shared L2 Instruction Caches," *IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 80-89, 2008.

[29] Yip E., Roop P., Biglari-Abhari M., and Girault A., "Programming and Timing Analysis of Parallel Programs on Multicores," *in Proceedings of 13th International Conference on Application of Concurrency to System Design*, Barcelona, pp. 160-169, 2013.

[30] Yip E., Roop P., and Biglari-Abhari M., "Predictable Parallel Programming Using PRET-C," Report, University of Auckland, Faculty of Engineering, 2010.

[31] Zhang W. and Yan J., "Accurately Estimating Worst-Case Execution Time For Multi-Core Processors With Shared Direct-Mapped Instruction Caches," *in Proceedings of 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Beijing, pp. 455-463, 2009.

**Muhammad Waqar Aziz** is an Assistant Professor in Umm Al-Qura University, Makkah, Saudi Arabia. He received his Ph.D (Computer Science) from Universiti Teknologi Malaysia, Malaysia in 2013, MS-Software Engineering from City University of Science and Technology, Pakistan in 2009 and MSc- Computer Science from University of Peshawar, Pakistan in 2001. Previously, he has almost eight years of teaching experience as Lecturer at Institute of Management Studies, University of Peshawar. He published more than 20 research articles in indexed and well reputed journals and Conference Proceedings. The research areas of his interest are Embedded Real-Time Software modeling, verification and development and smart environment development.

**Syed Abdul Baqi Shah** is a Lecturer at Science and Technology Unit, Umm Al Qura University, Makkah, Saudi Arabia. He received a B.S. degree in Electronic Engineering from International Islamic University, Islamabad, Pakistan in 2007. He completed his MS in Information and Mechatronics from Gwnangju Institute of Science and Technology, Republic of Korea in 2010. His research interests include timing analysis of real-time systems and applications, as well as design and implementation of low-power embedded system.