Test-Data Generation for Testing Parallel Real-Time Systems

Muhammad Waqar Aziz, Syed Baqi Shah

Test-data Generation for Testing Parallel Real-Time Systems

Muhammad Waqar Aziz and Syed Abdul Baqi Shah

Science and Technology Unit, Umm Al-Qura University, Kingdom of Saudi Arabia
mwaziz,sashah@uqu.edu.sa

Abstract. The Worst-Case Execution Time (WCET) of real-time systems is mainly influenced by the program design, its execution environment and the input data. To cover the last factor in the context of WCET estimation, the objective of this work is to generate the test-data that maximize the execution times of the parallel real-time systems. In this paper, a test-data generation technique is proposed that uses Genetic Algorithms to automatically generate the input data, to be used for testing of parallel real-time systems. The proposed technique was applied to a parallel embedded application – *Stringsearch*. The result was an analysis that took as input the parallel program and generated the test-data that cause maximal execution times. The generated test-data showed improvements by exercising long execution times in comparison to randomly generated input data.

Keywords: Test-data generation, Genetic Algorithm, Real-time systems, Measurement-based analysis, Worst-Case Execution Time analysis, End-to-end testing

1 Introduction

With the wide use of multicore processors in desktop computers and embedded systems, and the growing demands of high performance real-time applications, it is expected that multicore processors will be increasingly used in real-time systems [26]. The deployment of real-time applications on multicore platforms with tens or hundreds of cores may become a reality very soon [3]. This demands special methods and techniques for the design and testing of future multicore embedded real-time systems, where the previous research mostly assumes sequential code running on single-core platforms.

The fundamental property to guarantee the performance of a real-time system is its Worst-Case Execution Time (WCET) testing. The worst-case bounds can be derived either by using static timing analysis [11], or by measuring the programs execution time on a given hardware or simulator using a set of inputs [24]. The static timing analysis methods applied on hardware and software models of the system are very difficult to apply for parallel systems. For instance, the inter-thread interferences among shared resources, e.g., L2 caches are hard

to analyze statically [26]. In contrast, measurement-based methods can be used for better estimating the execution time of parallel systems and therefore are widely used in the industry.

The problem of WCET testing is to find the test-data that causes execution of the longest path of the program, and thus causes the longest execution time. But to do so, a complete test with all possible inputs, generally cannot be carried out. Similarly, exhausting all the possible program paths, for a given input, is usually infeasible. To handle these problems, evolutionary testing can be utilized that automatically searches the test-data to estimate the WCET. For instance, if searching the worst-case inputs from the set of all possible inputs is considered as an optimization problem, Genetic Algorithms (GA)[1] can be utilized to automatically search the required test-data.

The current research efforts of WCET analysis for multicore systems are focused on performance enhancing hardware features [19, 12, 15], and application [16, 18, 4] or programming level [27, 28, 9]. However, there is lack of research in test-data generation for WCET estimation of parallel real-time systems executing on multicore architectures. To fill this gap, this work proposes a Search-Based Software Engineering (SBSE) technique to generate test-data for testing parallel real-time systems. The proposed technique uses GA to sub-optimally evolve the worst-case inputs, with the objective to find those inputs that will cause the program to take longest execution time.

The proposed technique is applied by measuring the end-to-end execution time of the *ParMiBench* benchmark suite [14]. *ParMiBench* is an open source parallel version of a subset of *MiBench* benchmark suite [10] – many of whose benchmarks appear to be suitable candidates for WCET analysis [7]. *ParMiBench* benchmark suite, actually designed to evaluate the performance of embedded multi-core systems, is implemented using C language and POSIX threads to achieve parallelism and supports Unix/Linux based platforms [14]. The end-to-end time was measured by gathering the execution traces of the parallel program using the Gem5 simulator [2].

This article is organized as follows. Section 2 explains the methodology followed in this work. Section 3 describes the method to approach the problem of test-data generation along with the details of the actions performed. Section 4 reports the experimental setup and the results obtained from the experiment. The evaluation of this work is provided in Sect. 5. Section 6 describes related work in measurement-based WCET analysis for parallel real-time systems. Section 7 contains concluding remarks and directions for future work.

2 Methodology

The *ParMiBench* benchmark suite, used in this work, is a set of embedded parallel benchmarks from various domains of the embedded applications which include, control and automation, networks, offices, and security. However, we have selected the *Stringsearch* benchmark from the suite that is related to searching

a *token* (strings stored in a *pattern file*) from a *text file*, due to the following reasons:

- In this work, the benchmarks are seen as a black box that consumes input and generates timing information. However, for the testing process to be effective and the testing results to be meaningful, we need benchmarks with a rich input space.
- the current Gem5 framework forces to execute the benchmarks from within a disk image in the full system mode. This requires the benchmarks to have input parameters that can be easily fed, e.g., via command line.

In the initial experiment, the simulation was executed with the existing data set of *Stringsearch* benchmark. However, the initial experiment revealed that the text and pattern files in the existing data set consist of repeated contents. Consequently, any new pattern file generated, with tokens picked up randomly from the text file, had tokens that exist in every line of the file regardless of which line they have been picked. Thus, the existing data set of the benchmark was inappropriate to allow evolutionary testing, as the data set should not contain duplicate values. Therefore, a new text file with dissimilar tokens was generated that consists of 102400 random characters (file size $\approx$ 100 KB), and a pattern file containing 64 tokens each of five characters length. The token length was kept fixed (i.e., five characters, similar to the given benchmark) to observe the execution time variations unassociated with the input length. Additionally, fixed size inputs allow the easy alignment of parts during crossover operation.

In this work, Gem5 architecture simulator was used to measure the end-to-end execution time of the parallel benchmark. Gem5 was selected as it provides full-system simulation to execute a program in the operating system environment, with support of several commercial Instruction Set Architectures (just as ARM, ALPHA). To get the execution traces of our interest, we performed some tweaking to the simulator. We applied a Kernel patch and made some custom modifications to the simulator to get the execution traces of the benchmark.[1] These traces were then used to calculate the end-to-end execution time of the benchmark. Instead of manual calculation, the end-to-end execution time was calculated automatically by a Java application.

3 Proposed test-data generation technique

The technique proposed in this work for test-data generation, is based on SBSE approach, namely GA. GA 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. The steps of the proposed test-data generation technique, as depicted in Fig. 1, are described is below:

[1] Interested readers can visit our technical report for more configuration details, http://bit.ly/1JqheNS.

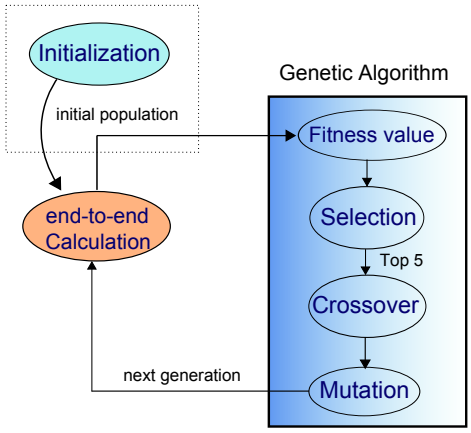


Fig. 1: GA based test-data generation technique proposed for parallel real-time systems

3.1 Define the Initial Population

For applying GA, an initial population of individuals need to be defined which is evolved across a number of generations. Individuals are usually random guesses to the solution of a problem. Care should be given to maintain diversity in the population so that premature convergence towards a sub-optimal solution can be prevented. For instance, in case of *Stringsearch*, a good candidate for the best individual produced by the GA in the last generation is the input to the program where all tokens to be searched do not exist in the text file.

To achieve this, an intuition-based selection was made to start from a point where 50% of tokens exist in the text and 50% do not. An initial population of one hundred pattern files, to be used in GA, was generated using the following ways:

a) By randomly picking up tokens from this newly generated text file
b) By randomly generating totally new tokens

Consequently, each file contained a set of tokens gathered from a mixture of 50% randomly generated tokens and 50% tokens picked up from the text. Thus, the chances of each token within a pattern file are equally likely to be in the text or not, making it a fair distribution to start with.

To conclude, there are (n*k)/2 tokens that exist in the text, and are randomly distributed among the total n*k tokens in n files (considering n pattern files, each having k patterns). This allows to give the 50-50 found/not found distribution without making individuals in the population biased. This set of one hundred pattern files collectively formed the genetic representation of the solution domain. Thus, the initial population in this experiment consists of one hundred chromosomes and 64 genes of each chromosome, in GA terminology.

3.2 End-to-End Time Calculation

In this work, it is proposed to use the end-to-end execution time of parallel program as *fitness function*. The end-to-end time was calculated from the traces obtained from the Gem5 simulator by executing the parallel application. Each trace was generated by detecting the starting and ending points of a thread execution. A Java application was written to automatically calculate the end-to-end time from the obtained execution traces. In the initialization phase, the end-to-end time was calculated for all the 100 files present in the initial population. The simulation was run one hundred times to generate one hundred traces, i.e., one time for each pattern file. However, in all the other iterations it was calculated for the new individuals of the next generation only.

3.3 Applying Genetic Algorithm

GA uses the concept of natural evolution to reach the desired solution from a given huge search space. The main idea of using GA, in this work, is to execute the program with sets of inputs throughout a number of generations. The process starts with generating k random vectors initially (first generation) and obtain their timing information. Then, the generated k random inputs with the timing information are used to produce the next k inputs (second generation). This process is repeated for n generations. The details of the steps of GA as followed in the proposed technique are given below.

1. **Calculate the Fitness Value**

   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. The fitness value is used to select the best of any generation to 'mate' them in order to produce the new generation.
   As already mentioned, the end-to-end time is considered as a fitness value, in this work. Thus, the longer the end-to-end time, the higher would be the fitness. The fitness value is generated in GA through fitness function. It means that calculating the fitness function requires the execution of the program to produce traces. The time taken by *Stringsearch* to execute each pattern file was considered as the fitness value of that pattern file. The use of GA is suppose to search the inputs with higher fitness values in each generation. In this way, the program execution using GA would lead towards inputs having larger fitness values.
   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 a huge research in its own and requires more effort and time. In addition, Gem5 simulator does not provide much information about these parameters which can be useful at this level.

2. **Select the Individuals**

   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. To make a rank based selection, the individual need to be sorted based on their fitness values. It is proposed to select only five individuals, as selecting more individuals would take more time in the remaining steps and also in calculating their fitness in next generation.
   In this experiment, the pattern files were sorted based on their fitness values. The top five pattern files were selected as chromosomes for the next generation. Top five files were selected because selecting 50 files, for instance, will be a big enough number as it will make 100 files for the next generation which will take too long to calculate their fitness.

3. **Crossover**

   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. To produce new files, a crossover was made amongst the top five selected pattern files, by merging two lists of tokens picked up from randomly selected pattern files. This crossover resulted in ten new chromosomes. 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 is based on cutting one chromosome at a random location and concatenating it with the remaining part of the second chromosome cut at the same location.

4. **Mutation**

   The evolution of the population also involves the alteration of the genetic material of a single individual through mutation operation. Mutation is achieved by picking a bit at random and flipping its value. Mutation 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 desired percentage of mutation was achieved. In this way, the next generation of 10 chromosomes was produced.

The above process was repeated for a set of ten chromosomes, which was reproduced after each generation. The evolution continues until a good-enough individual that solves the problem adequately is found, or until a maximum number of generations is reached. Opting for the latter case, the whole process was repeated for 50 generations, as the probability of improving over several generations is high.

With the above modeled parameters, a Java application was developed that implemented GA in the experiment. The implementation used the given pattern files composed of tokens without converting them into binary strings. This simplified the complex problem of coding and decoding of inputs for this specific case.

4 Experimentation

4.1 Experimental Setup

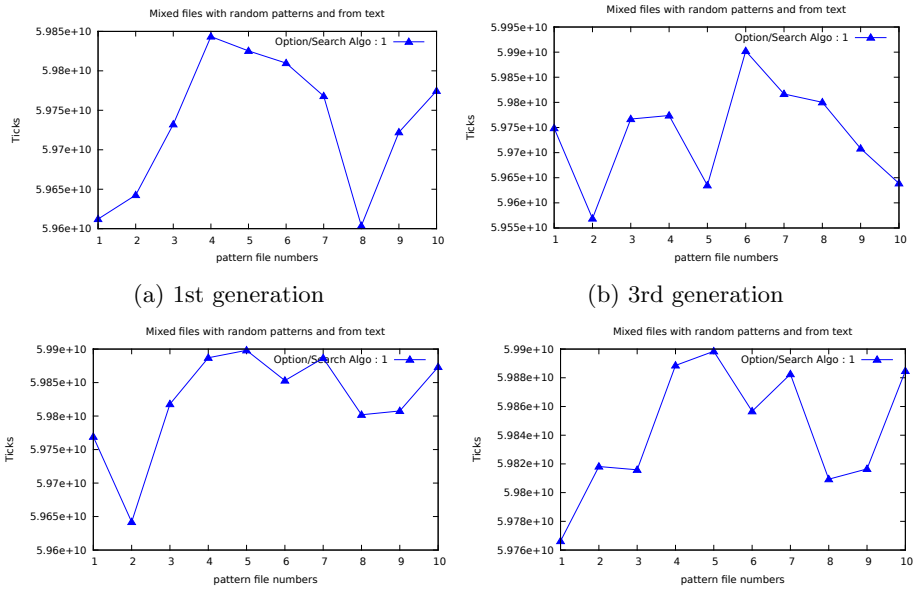
The Gem5 configurations, as used in this experiment, included four cores of ARM detailed architecture as specified by Gem5 (ARMv7-A ISA based) with default size of L2 cache (2MB) and 256 MB of Memory. The disk image contained ARM embedded Linux (AEL) as the guest operating system for the simulator. We used Gem5 to run the simulations and execute benchmarks for collecting the kernel level thread traces and computing the end-to-end times. 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.

4.2 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 each graph, the length of the execution time is represented on the vertical axis in terms of CPU ticks, whereas the ten pattern files of each generation are represented horizontally. For example, in first generation the highest and the lowest values were found as 59843433000 and 59603617500 ticks for 4th and 8th pattern files respectively, as shown in Fig. 2a. Some other results produced during fifty generations are depicted in the remaining parts of Fig. 2.

From the plotted graphs, i.e., the measured end-to-end times, it is observed that by using GA an overall improvement can be achieved in the fitness values corresponding to the pattern files. For instance, in first and third generations (Fig. 2a and 2b) there are only three pattern files which caused the program to execute for a period longer than 5.98e+10 ticks. In comparison, the number of pattern files, which execute longer than this number of ticks, is increased to eight pattern files in 25th generation and nine pattern files in 50th generation (Fig. 2c and 2d). This increase in the number of input files after applying several generations shows that the inputs are taking longer time. This is a clear indication that the proposed technique has improved the inputs, i.e., those inputs are generated that cause the longest time to execute.

Table 1 shows the WCET measured during the experiments in different generations for five character length input token. A threshold value of 5.98e+10 was defined to analyze the improvement in the inputs over all generations. The number of files taking greater or equal time than the threshold value was counted, as represented by Above (Th.5.98) column in Table 1.



(a) 1st generation

(b) 3rd generation

(c) 25th generation

(d) 50th generation

Fig. 2: End-to-End times (fitness values) in different generations for the pattern files with five characters long tokens

An increasing number of input files above the threshold can be observed, in Table 1, except for 30th generation. 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.

5 Evaluation

In order to evaluate the scalability of the proposed technique, the experiment was repeated with other input files as given with the benchmark. 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 input token was used. Some of the graphs, representing different generations produced using 10 characters length are depicted in Fig. 3. The WCET measured for input tokens of 10 characters length are displayed in Table 2. The threshold value, in this case, was defined as 5.9e+10 with the same purpose of analyzing the inputs over all generations.

From Table 2, it can be observed that the number of input files increased with the number of generations. Although the number of files slightly increased and decreased due to GA, an overall steady increase in the number of files was observed after 50th generations.

Table 1: Measured WCET across different generations for 5 characters length token

Input token size	Generation No.	WCET	Above (Th.5.98)
5 characters	1	59843433000	3
	10	59878900000	5
	20	59895367000	8
	30	59898941500	6
	40	59896569500	7
	50	59898461000	9

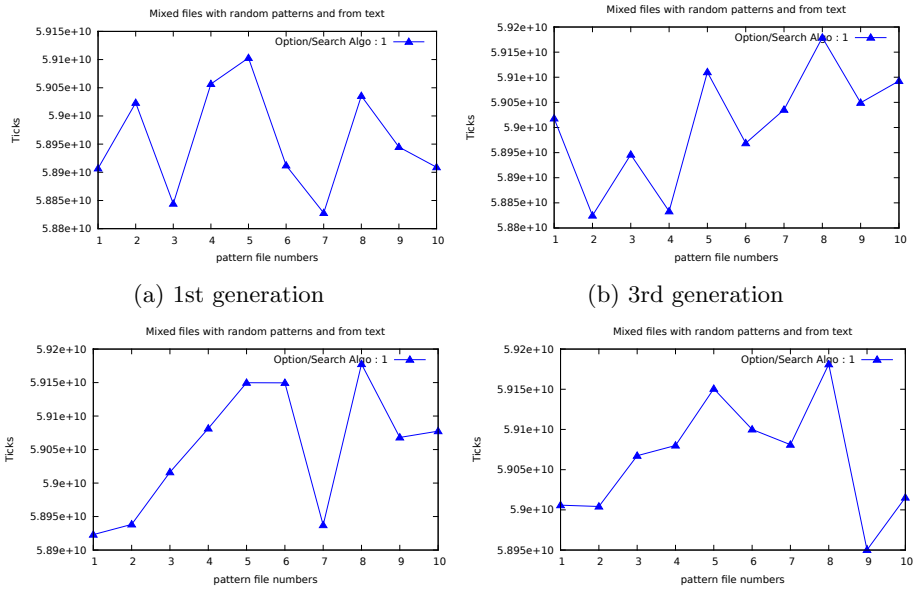
Table 2: Measured WCET across different generations for 10 characters length token

Input token size	Generation No.	WCET	Above (Th.5.90)
10 characters	1	59102869500	4
	10	59134277000	5
	20	59097647500	6
	30	59100034000	6
	40	59181076000	7
	50	59181090000	9

6 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 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 on multi-core architectures [8, 25, 26, 29], and (2) design of analyzable multi-core computers that favor timing predictability over performance [6, 17, 20, 21]. In relation to this work, research on parallel applications running on multi-core architectures is very limited. For instance, Rochange et al. [19] highlights the problem of analyzing the timing behavior of non-sequential software on a multi-core architecture. They report a manual analysis of a parallel application, which determines the synchronization and communication between its executing threads.

In contrast, we have used GA to heuristically search the input from a huge search space of tokens that would cause the program to execute for the longest period of time. It was observed that the execution time of the search tends to decrease with the increase in size of the input text (see e.g., WCET of 5 and 10 characters). In general, the proposed technique is applicable to any parallel real-time system where optimization is needed. This further requires that the system under consideration can be genetically represented and has a fitness function for its evaluation. However, the proposed technique should be complemented with



(a) 1st generation

(b) 3rd generation

(c) 25th generation

(d) 50th generation

Fig. 3: End-to-End times (fitness values) in different generations for the pattern files with ten character long tokens

static timing analysis if *safety* is required, i.e., to ensure that the obtained results are close to the actual WCET of the considered system.

Evolutionary search (more specifically GA) has been employed in the literature [5, 13, 22] to find long execution times of real-time programs. Although, research on using GA for testing real-time systems dates back to 90s [23], it has not been used for WCET analysis of parallel programs running on multi-core hardware, to the best of our knowledge. The fitness function used in this work considered the end-to-end time for the execution of a program as the fitness value. This consideration has helped us to produce good enough results by using GA that maximized the fitness value.

7 Conclusion

In this paper, a measurement-based technique is proposed for automatic test-data generation for parallel real-time systems running on a multicore architecture. The technique uses Genetic Algorithm to generate test data that maximize the execution times of the parallel application. It evolves input vectors that cause long execution times of the program using evolutionary testing. The results of the experiment showed a significant improvement in the execution times of input files after applying the proposed technique. Thus, the aim of producing large execution times, which are either the WCET or close to it, was achieved.

In the future, we aim to use a real-life, real-time application to evaluate our work. In case of publicly-unavailability of such applications, the method can be tested with other benchmarks of *ParMiBench*. Moreover, the static timing analysis is planned to be performed to evaluate the safeness and tightness of the proposed technique. Although, the end-to-end time is considered as an output to the fitness function, it is also planned to consider a richer multi-objective fitness function in future that might include thread conflicts, cache misses/hits and cache sizes.

Acknowledgments. This research is funded by the program of strategic technologies of the National Science, Technology, and Innovation plan in Saudi Arabia (Grant No. 11-INF1705-10). We acknowledge KACST (King Abdulaziz City for Science and Technology) for funding this research and STU (Science and Technology Unit), Umm Al-Qura University, Makkah for providing necessary support.

References

1. Berg, C., Engblom, J., Wilhelm, R.: Requirements for and design of a processor with predictable timing. In: Design of Systems with Predictable Behaviour (2004)
2. Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., et al.: The gem5 simulator. ACM SIGARCH Computer Architecture News 39(2), 1–7 (2011)
3. Calandrino, J.M., Anderson, J.H., Baumberger, D.P.: A hybrid real-time scheduling approach for large-scale multicore platforms. In: Real-Time Systems, 2007. ECRTS'07. 19th Euromicro Conference on. pp. 247–258. IEEE (2007)
4. Ding, Y., Zhang, W.: Multicore-aware code co-positioning to reduce wcet on dual-core processors with shared instruction caches. JCSE 6(1), 12–25 (2012)
5. Gross, H.G.: 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, Nepal (2003)
6. Guan, N., Stigge, M., Yi, W., Yu, G.: Cache-aware scheduling and analysis for multicores. In: Proceedings of the seventh ACM international conference on Embedded software. pp. 245–254. ACM (2009)
7. Gustafsson, J., Betts, A., Ermedahl, A., Lisper, B.: The mälardalen wcet benchmarks: Past, present and future. In: OASIcs-OpenAccess Series in Informatics. vol. 15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2010)
8. Gustavsson, A., Ermedahl, A., Lisper, B., Pettersson, P.: Towards wcet analysis of multicore architectures using uppaal. In: OASIcs-OpenAccess Series in Informatics. vol. 15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2010)
9. Gustavsson, A., Gustafsson, J., Lisper, B.: Toward static timing analysis of parallel software. In: OASIcs-OpenAccess Series in Informatics. vol. 23. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2012)
10. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: Mibench: A free, commercially representative embedded benchmark suite. In: Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on. pp. 3–14. IEEE (2001)

11. Heckmann, R., Ferdinand, C.: Worst-case execution time prediction by static program analysis. In: In 18th International Parallel and Distributed Processing Symposium (IPDPS 2004, pages 26–30. IEEE Computer Society (2004)
12. Kästner, D., Schlickling, M., Pister, M., Cullmann, C., Gebhard, G., Heckmann, R., Ferdinand, C.: Meeting real-time requirements with multi-core processors. In: Computer Safety, Reliability, and Security, pp. 117–131. Springer (2012)
13. Khan, U., Bate, I.: Wcet analysis of modern processors using multi-criteria optimisation. In: Search Based Software Engineering, 2009 1st International Symposium on. pp. 103–112. IEEE (2009)
14. Liang, Y., Iqbal, S.M.Z.: OpenMPBench-An Open-Source Benchmark for Multiprocessor Based Embedded Systems. Ph.D. thesis, Master thesis report MCS-2010: 02, School of Computing, Blekinge Institute of Technology, Sweden (2010)
15. Liang, Y., Ding, H., Mitra, T., Roychoudhury, A., Li, Y., Suhendra, V.: Timing analysis of concurrent programs running on shared cache multi-cores. Real-Time Systems 48(6), 638–680 (2012)
16. Ozaktas, H., Rochange, C., Sainrat, P.: Automatic wcet analysis of real-time parallel applications. In: OASIcs-OpenAccess Series in Informatics. vol. 30. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2013)
17. Pitter, C., Schoeberl, M.: A real-time java chip-multiprocessor. ACM Transactions on Embedded Computing Systems (TECS) 10(1), 9 (2010)
18. Potop-Butucaru, D., Puaut, I., et al.: Integrated worst-case response time evaluation of multicore non-preemptive applications (2013)
19. Rochange, C., Bonenfant, A., Sainrat, P., Gerdes, M., Wolf, J., Ungerer, T., Petrov, Z., Mikulu, F.: Wcet analysis of a parallel 3d multigrid solver executed on the merasa multi-core. In: OASIcs-OpenAccess Series in Informatics. vol. 15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2010)
20. Rosen, J., Andrei, A., Eles, P., Peng, Z.: Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In: Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International. pp. 49–60. IEEE (2007)
21. Supercomputing, B.: Merasa: Multicore execution of hard real-time applications supporting analyzability (2010)
22. Wegener, J., Mueller, F.: A comparison of static analysis and evolutionary testing for the verification of timing constraints. Real-Time Systems 21(3), 241–268 (2001)
23. Wegener, J., Sthamer, H., Jones, B.F., Eyres, D.E.: Testing real-time systems using genetic algorithms. Software Quality Journal 6(2), 127–135 (1997)
24. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., et al.: The worst-case execution-time problemoverview of methods and survey of tools. ACM Transactions on Embedded Computing Systems (TECS) 7(3), 36 (2008)
25. Wu, L., Zhang, W.: Bounding worst-case execution time for multicore processors through model checking. In: Proc. 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS10), Work-in-Progress Session. pp. 17–20 (2010)
26. Yan, J., Zhang, W.: Wcet analysis for multi-core processors with shared l2 instruction caches. In: Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS'08. IEEE. pp. 80–89. IEEE (2008)
27. Yip, E., Roop, P.S., Biglari-Abhari, M.: Predictable Parallel Programming Using PRET-C. University of Auckland, Faculty of Engineering (2010)

28. Yip, E., Roop, P.S., Biglari-Abhari, M., Girault, A.: Programming and timing analysis of parallel programs on multicores. In: Application of Concurrency to System Design (ACSD), 2013 13th International Conference on. pp. 160–169. IEEE (2013)
29. Zhang, W., Yan, J.: Accurately estimating worst-case execution time for multi-core processors with shared direct-mapped instruction caches. In: Embedded and Real-Time Computing Systems and Applications, 2009. RTCSA'09. 15th IEEE International Conference on. pp. 455–463. IEEE (2009)