Proc. of The Seventh Intl. Conf. On Advances In Computing, Control And Networking - ACCN 2017

ISBN: 978-1-63248-134-4 doi: 10.15224/ 978-1-63248-134-4-19

On the Computation of the WCET for LEON3 and MIPS Processors

Bruno Green, Guilherme Debom, Letícia Bolzani Poehls, Fabian Vargas, Celso Maciel da Costa

***Abstract*— In the last two decades, the worst-case execution time (WCET) bound computation was a topic mainly related with hard real-time systems for aerospace and defense applications. Recently, however, it has become crucial in other domains dealing with timing guarantees. This includes among others, the automotive industry, where V2V and V2X applications for autonomous vehicles are demanding not only fault-tolerant responsiveness, but also the guarantee that timing deadlines will not be violated. In this sense, it is a mandatory condition to have an accurate determination of the WCET parameter in order to guarantee the hard-real time response of these critical systems to the environment. We propose in this paper a method to bound WCET for workloads running in the LEON3 and MIPS architectures. This approach performs a static timing analysis of the application code and based on the IPET technique, bounds the WCET. In addition to the use of the IPET technique, the proposed approach also takes advantage of commercial, open-source tools such as GraphViz and lp_solve to produce a WCET upper bound.**

***Keywords*— Worst-Case Execution Time (WCET), LEON3 Processor, MIPS Processor, Implicity Path Enumeration Technique (IPET), Prediction of critical code execution time, Static Timing Analysis.**

I. Introduction

A real-time computer system is a computer system where the *correctness* of the system behavior depends not only on the *logical* results of the computations, but also on the *physical time* when these results are produced. If a result has utility even after the deadline has passed, the deadline is classified as *soft*, otherwise it is *firm*. However, if severe consequences could result if a firm deadline is missed, then the deadline is called *hard* [2]. Fig. 1 depicts the basic notions concerning timing analysis of systems.

A task typically shows a certain variation of execution times depending on the input data or different behavior of the environment. The longest response time is called the *worst-case execution time* (WCET). In most cases, the state space is too large to exhaustively explore all possible executions and thereby determine the exact WCET. It is worth noting that while in the last decades WCET bound was a topic mainly related with hard real-time systems (such as aerospace and military), recently it has become crucial in other domains dealing with timing guarantees. This includes among others, the automotive industry, mobile communication and high-performance computing. In this sense, it is a mandatory condition to have an accurate determination of the WCET parameter in order to guarantee the hard-real time response of these critical systems to the environment [2].

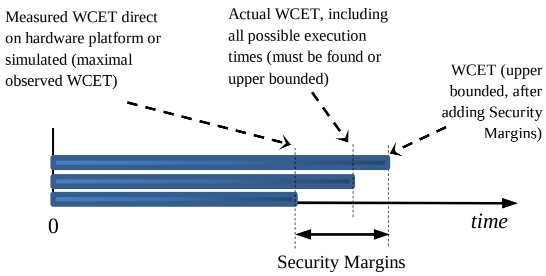

Measured WCET direct on hardware platform or simulated (maximal observed WCET)
Actual WCET, including all possible execution times (must be found or upper bounded)
WCET (upper bounded, after adding Security Margins)
0
time
Security Margins

Fig. 1. Basic notions concerning timing analysis of systems.

In most parts of industry, the common method to estimate execution time bound is to measure the *end-to-end* execution time of the task for some set of inputs (test cases) on the target hardware or on a clock cycle-accurate simulator. This determines the *maximal observed execution time*. This will, in general, underestimate the WCET and so is not safe for hard real-time systems. This method is often called *dynamic timing analysis*. In contrast to this method, there is the *static timing analysis*, which is the preferred method used by academia. This method does not rely on executing code on real hardware or on a simulator. Rather, it takes the task code itself, most often together with some annotations, constructs a control-flow graph (CFG) of the workload and analyzes the set of all possible paths through the CFG. Next, this technique combines control-flow analysis with (abstract) models of the processor architecture (e.g., pipeline, cache memory and bus-access policy models) in order to obtain the WCET bound for the workload.

A lot of research has been carried out within the area of WCET analysis [2]. However, each task is, traditionally, analyzed in isolation as if it was running on a monoprocessor system. Consequently, it is assumed that memory accesses over the bus take constant amount of time to process. For multiprocessor systems with a shared communication infrastructure, however, transfer times depend on the bus load and are therefore no longer constant, causing the traditional methods to produce *incorrect* results [2]. As response to this specific need, several approaches dealing with WCET

Bruno Green, Letícia B. Poehls and Fabian Vargas are with the Catholic University – PUCRS. Electrical Engineering Dept. Av. Ipiranga 6681, 90619-900, Porto Alegre, Brazil.
Guilherme Debom and Celso M. da Costa are with the Universidade Estadual do Rio Grande do Sul (UERGS). R. Santa Maria 2300, 92500-000, Guaiba, Brazil.

prediction in multicore platforms have been proposed [3,4,5,6].

In [3], authors proposed a technique to achieve predictability of tasks running in multiprocessor systems. The approach is based on the simultaneous analysis of the critical task running in a given core with the shared-bus scheduling process, in order to bound the WCET for that task. In order to calculate the whole WCET of such task, the analysis needs to be aware of the TDMA bus, taking into account that cores must only be granted the bus during their assigned time slots.

In [4], authors proposed a unified WCET static timing analysis approach for multicore processors. This work is based on models of cache and shared bus, which interact with other basic micro-architectural models (e.g. pipeline and branch predictor unit). Each processor core is analyzed at a time by taking care of the inter-core conflicts generated by all other cores. In this multicore scenario, it is assumed a TDMA shared bus based on round robin arbitration policy, where a fixed length bus time slot is assigned to each core. They also assume fully separated caches and buses for instruction and data. Therefore, the data references do not interfere with the instruction references. This work only models the effect of instruction caches. Since it is considered only instruction caches, the cache miss penalty (computed from cache analysis) directly affects the instruction fetch (IF) stage of the pipeline. Finally, authors consider the LRU cache replacement policy.

In [5], authors proposed a method to bound WCET for workloads running in a multicore architecture where each core has a local L1 cache and all cores use a shared bus to access the off-chip memory. They modeled the local cache behavior of a program running on a dedicated core. Then, based on the cache model, they constructed a Timed Automaton (TA) to model when the programs access the shared bus. Examples for TDMA and FCFS buses were analyzed.

In contrast to [3,4,5] that are approaches based on static timing analysis, in [6] authors described a project called “Merasa: Multicore Execution of Hard Real-Time Applications Supporting Analyzability”. This work aimed at developing multicore processor design (described in SystemC) for hard real-time embedded systems and a technique to guarantee the analyzability and timing predictability of every feature provided by the processor. Publications presented results for a quad-core version of this processor, where each core consists of two pipelines and implements the TriCore (Infineon) instruction set. Each core provides up to four thread slots (separate instruction windows and register sets per thread), which allows simultaneous execution of one hard real-time task and three non-hard real-time tasks. The processor architecture contains one inter-core bus arbiter, which arbitrates requests from different cores, and four intra-core bus arbiters (one per core) that arbitrate among thread requests from the same core. The processor shared memory can suffer from both intra- and inter-core interferences. To avoid these interferences, authors proposed a dynamically partitioned memory, which assigns a private subset of memory banks to each hard real-time task so that no other task has access to it (the Merasa operating system sets the memory partition assigned to each core by modifying special hardware registers). Also, the Merasa processor runs based on a Round Robin bus policy.

The Merasa system-level software represents an abstraction layer between the application software and the embedded hardware. It provides the basic functionalities of a real-time operating system as a foundation for application software running on the Merasa processor. Merasa system-level software guarantees the isolation of memory accesses of various hard real-time tasks that are running on different cores to avoid mutual and possibly unpredictable interferences. This isolation should also enable a tight WCET analysis of application code. The resulting system software can execute hard real-time tasks in parallel on different cores of the Merasa multicore processor.

The Merasa processor and techniques were validated by means of determining WCET for a given application based on the use of two CAD tools, one academic and one from industry: Otawa [7] and RapTime [8], respectively. While Otawa extracts the control flow graph (CFG) from the binary code (thus, performing static timing analysis), RapiTime uses the extracted traces to estimate the WCET by measurements/simulations of the target hardware. Further, the Merasa project was continued on a new action: parMerasa [10,11].

These approaches [3,4,5,6] represent a considerable improvement of the state-of-the-art, but note:

a) Concerning the approach presented in [6], up to this moment, and from the best of our knowledge, it is applicable only to the Merasa processor. So, traditional processors used in embedded applications such as PowerPC, ARM, MIPS and LEON3 as well as well-stablished real-time operating systems for critical applications such as VxWorks, LynxOS, Integrity or RTEMS and their versions compliant with ARINC-653 (an avionics standard for safe, partitioned systems) [9] cannot take advantage of this approach yet.

b) A few anchor Brazilian companies, among them, Embraer, needs to have full access to the state-of-the-art technologies developed in the field of static (and dynamic) WCET analysis. However, this area of research is very sensitive, presenting dual use in aerospace industry, not only in commercial, but also in the defense domain. This needs justify the development of the current work.

In this work, the terms “task” and “workload” have the same meaning and are used interchangeably. The remainder of the paper is organized as follows: Section II describes the proposed approaches based on the IPET technique to compute the WCET for the LEON3 and MIPS processors. Section III presents the preliminary results towards the validation of the proposed approaches. Finally, Section IV draws the final conclusions of the work.

II. Proposed Approach to Compute WCET

A) LEON3 Processor

The determination of the WCET for the LEON3 processor is carried out by the following five steps (see Fig. 2):

ISBN: 978-1-63248-134-4 doi: 10.15224/ 978-1-63248-134-4-19

(1) CFG Generation: The first step of the approach is devoted to the reconstruction of the application code from the executable- to the assembly-level. Then, the code is translated into a control-flow graph (CFG) where the *edges* represent basic blocks and the *vertices*, conditional (or unconditional) branches from one basic block to another in the code. A basic block is a minimal set of ordered instructions in which its execution begins from the first instruction and terminates at the last instruction. There is no branching instruction in a basic block except possibly for the last one. A basic block terminates at either an instruction branching to another basic block or an instruction receiving transfer of control flow (CF) from two or more places in the program.

Once the CFG is built-up, it serves as input for the next analysis step (Timing Annotation) where the CFG is annotated with information such as: ranges for the input values of the program, loop bounds, shapes of nested loops, if iterations of inner loops depend on iteration variables of outer loops, frequencies of paths or branches taken, hardware anomalies (time penalty due to erroneous branch prediction at the execution stage of the pipeline) and if possible, unfeasible paths. In the case of "unfeasible paths", different paths through the CFG are taken depending directly or indirectly on input data. Some paths in the CFG will never be taken, for instance, those that have contradictory consecutive conditions. Eliminating such paths may increase the precision of timing analysis.

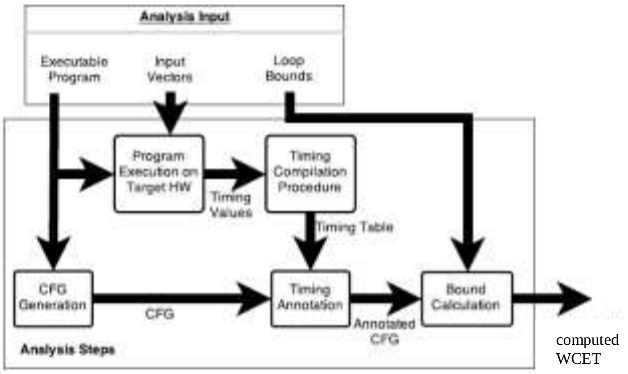


Fig. 2. WCET computation flow.

(2) Program Execution on Target HW: Executes the application in the target hardware with the provided input vectors and gather timing information of each instruction using measurement techniques. For the LEON3 processor, it is used the Debug Support Unit (DSU) Instruction Trace (from Xilinx) to collect the *timing tag* of every instruction executed through the processor pipeline. The *timing tag* is the time (in CPU clock cycles) required for an instruction to traverse the processor pipeline from the "Fetch" to the "Exception" stage, i.e., from the beginning of the 1st to the end 6th stage of the pipeline.

In more detail, the DSU executes individual basic blocks or code snippets by observing the real-system execution. Then, it combines these measured individual times and their worst-case effects observed locally. The input vectors are provided randomly or are retrieved from application (if they exist). To gather timing information of each basic block or code snippet, the DSU traces instructions and stores the *timing tag* of every executed instruction.

Fig. 3 depicts the execution of four instructions in the LEON3 processor with the *timing tag* TT_i of each instruction I_i being collected at the end of the 6th stage of the pipeline. The Latency L_i of an instruction I_i is defined as the difference of the *timing tag* TT_i of the current instruction I_i to the *timing tag* TT_{i-1} of the previous instruction I_{i-1}. For instance, the Latency of instruction I_1 is $L_1 = TT_1 - TT_0$, I_2 is $L_2 = TT_2 - TT_1$, and so on.

Note that in this approach, the latency of the first instruction to enter in the pipeline cannot be acquired (I_0 in Fig. 3). If that time is desired, five *nop* instructions must be prepended to the code snippet being analyzed and the analysis must start at the the first *nop*. Additionally, since the Latency of an instruction is acquired at the end of the 6th pipeline stage, the time of commit of the last instruction in the pipeline is lost. If that time is desired, a single *nop* instruction must be inserted at the end of the code snippet being analyzed.

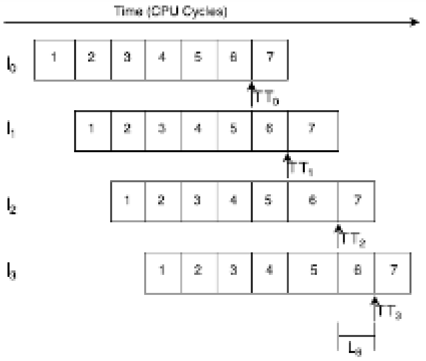
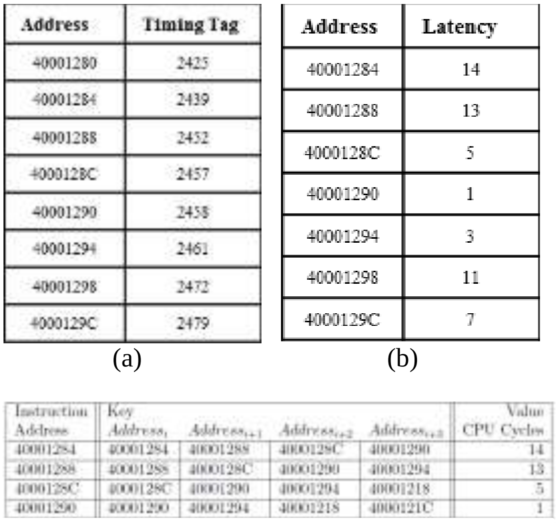


Fig. 3. Latency measurement for instructions passing through the LEON3 micro-pipeline architecture.

(3) Timing Compilation Procedure: Combines the *timing tags* measured from the target hardware into an intermediate structure named "Timing Table". The Timing Table is responsible for associating the address of an executed instruction I_i with the three future addresses of instructions (with respect to I_i) in order to determine the latency of instruction I_i into the pipeline.

The Timing Table is generated from a list of a tuple of instruction address and *timing tag* [(A_i; TT_i)] (see Fig. 4a). The latency L of each entry is calculated by the difference of the current instruction *timing tag* and the former instruction *timing tag* i.e., $L_i = TT_i - TT_{i-1}$ (for instance, $L_3 = TT_3 - TT_2$ in Fig. 3). The intermediate table is then generated with the addresses and computed latencies (Fig. 4b).

Fig. 4 shows an example trace of a task running on LEON3 processor measured with the DSU instruction trace of LEON3 mapped in a Xilinx FPGA. The resulting computed latency for each instruction is depicted in the intermediary table of Fig. 4b. For example, instruction with address 40001284 takes 14 clock cycles (cc) to be executed. This latency was computed from $L_i = TT_i - TT_{i-1}$ = 2,439 – 2,425 = 14 cc. The final Timing Table (Fig. 4c) is then generated from the computed latencies and instruction addresses of each run.



Address	Timing Tag
40001280	2425
40001284	2439
40001288	2452
4000128C	2457
40001290	2458
40001294	2461
40001298	2472
4000129C	2479

(a)

Address	Latency
40001284	14
40001288	13
4000128C	5
40001290	1
40001294	3
40001298	11
4000129C	7

(b)

Instruction	Key				Value
Address	$Address_i$	$Address_{i+1}$	$Address_{i+2}$	$Address_{i+3}$	CPU Cycles
40001284	40001284	40001288	4000128C	40001290	14
40001288	40001288	4000128C	40001290	40001294	13
4000128C	4000128C	40001290	40001294	40001218	5
40001290	40001290	40001294	40001218	4000121C	1

(c)

Fig. 4. Timing Table generation example: (a) Trace; (b) Intermediary Execution; (c) Timing Table.

(4) Timing Annotation: Annotates on the CFG vertices and edges the respective timing information stored in the Timing Table. In more detail, it annotates on every basic block (edge) of the CFG the number of clock cycles required for the CPU to traverse such basic block. It also annotates on every vertex of the CFG the cost (also given in clock cycles) for the CPU to execute a branch from one basic block to another one in the CFG.

It is worth noting that the LEON3 processor implements the Static Branch Prediction Approach with the "*Always Taken*" prediction. Then, the cost for a *branch taken* is 0 (zero) cc, while the one for a *mispredicted branch taken* in the LEON3 micro-pipeline is *n* cc. Note that *n* is dependent on the type of the instructions in the preceding stages of the pipeline (before the "Execution" stage, where the *branch* condition is verified). As more complex are the instructions in the preceding pipeline stages, more complex is the flush procedure to allow the processor to continue from a *mispredicted branch taken* decision.

(5) Bound Calculation: Computes the timing bounds for all paths of the annotated CFG. The timing bounds are computed by a modified version of the classic IPET approach [14] so that to take into account the specificities of the LEON3 micropipeline. Hereafter, we briefly describe the IPET technique.

The IPET Technique computes the timing bounds from the annotated CFG by using two possible techniques: IPET (Implicit-Path Enumeration Technique). In IPET, program flow and basic-block execution time bounds are combined into sets of arithmetic constraints. The idea was originally proposed in Li and Malik (1995) [16] and slightly modified in the current work, as described in the next paragraph.

For every edge (basic block) in the CFG is given a time coefficient (t_{edge}), expressing the upper bound of the contribution of that entity to the total execution time every time it is executed and a count variable (x_{edge}), corresponding to the number of times the entity is executed. See Fig. 5 for details. A local upper bound is determined by maximizing the sum of products ($\sum_{i \in edge} t_i * x_i$).

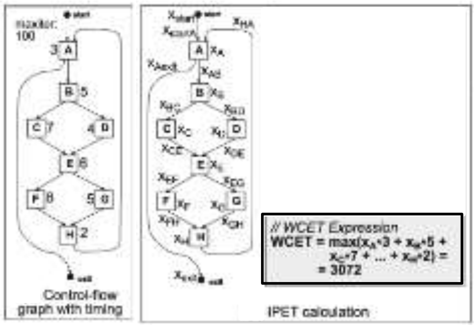


Fig. 5. Example of IPET computation.

Additionally, for every vertex in the CFG is given a time coefficient (t_{vertex}), expressing the upper bound of the contribution of that entity to the total execution time every time it is executed and a count variable (x_{vertex}), corresponding to the number of times the entity is executed. Similarly, a local upper bound is determined by maximizing the sum of products ($\sum_{j \in vertex} t_j * x_j$). Having noted this, the final result of an IPET calculation is the upper execution time bound and a worst-case count for each execution count variable for all entities:

$$max\ (\sum_{i \in edge} t_i * x_i) + (\sum_{j \in vertex} t_j * x_j)$$

subject to a set of constraints. These constraints reflect the structure of the task and possible flows such as unfeasible CFG paths. It should be mentioned that the second coefficient ($\sum_{j \in vertex} t_j * x_j$) was added as an original contribution of this work to the classic IPET formulation in order to satisfy the specific control flow of the LEON3 micro-pipeline. This coefficient represents the penalty cost from switching from one edge to another one in the CFG. In more detail, we assume one of the following values for t_j:

- 0 (zero) cc, for a *branch taken* in an Always Taken Approach;
- *n* cc, for the cost of a *mispredicted branch taken* in the LEON3 micro-pipeline. Note that *n* is dependent on the type of the instruction in the preceding stage of the pipeline (just before the "Execution" stage, where the *branch* condition is verified). Depending on the type of the executed *branch*, the instruction in the preceding stage of the pipeline is executed and in this case, it can take any number of clock cycles.

B) MIPS Processor

The proposed approach performs the analysis in two steps (Analysis and Computation) and requires a few user interactions during the process. Fig. 6 depicts the computation flow. Throughout the whole computation process, information is imported and exported among several files transparently to the user. Moreover, two commercial tools are called by means of the execution of specific scripts: GraphViz [12] and lp_solve [13]. After the Analysis step, the user needs to edit a text file (IPET.lp, Fig, 6) in order to proceed with the Computation step. In more detail, such edition process consists on defining the upper bound limits for all loops present in the analyzed code.

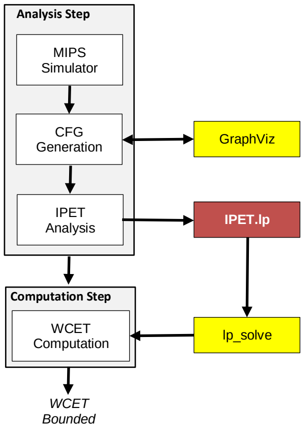


Fig. 6. Execution flow for the proposed approach. At the end of this process, the WCET for the MIPS processor is determined.

The input information at the very beginning of the process is the rebuilt Assembly code, which is simulated at the commercial MIPS Spin simulator. In the sequence, the Control-Flow Graph (CFG) is automatically generated by GraphViz tool. Graphviz is an open-source code used for graphical visualization. It is used to represent data structures such as abstract graph diagrams. The CFG nodes (edges) are formed by Basic Blocks and the vertex represent conditional/unconditional branches from one Basic Block to another one during the processor execution flow. A Basic Block is defined as a chunk of instructions in the code that are executed sequentially, with no branch instruction in between this slice of code. Additionally, the CFG contains information about the number of times a given Basic Block is visited along with a code loop (i.e., the upper bound limits of the loop). Based on the CFG and on the information provided by the MIPS simulator (number of instructions executed by the processor and the number of clock cycles per instruction), the computation of the WCET is performed by the commercial tool lp_solve. This tool is able to solve the system of equations defined by IPET technique.

III. Conclusions

In the last two decades, the worst-case execution time (WCET) bound computation has become a prime area of research for embedded real-time applications that are demanding not only fault-tolerant responsiveness, but also the guarantee that timing deadlines will not be violated. In this sense, it is a mandatory condition to have an accurate determination of the WCET parameter in order to guarantee the hard-real time response of these critical systems to the environment. We propose in this paper a method to automate the bounding process of WCET for workloads running in the LEON3 and MIPS architectures. This approach performs a static timing analysis of the application code and based on the IPET technique, bounds the WCET. In addition to the use of the IPET technique, the proposed approach also takes advantage of commercial, open-source tools such as GraphViz and lp_solve to produce a WCET upper bound.

Currently, we are working on the validation process of the proposed approach by performing a series of practical experiments where the WCET is being bounded for soft-cores of the LEON3 and MIPS processors. These cores have been mapped into Xilinx and Microsemi FPGAs and real-time application codes are being developed.

Acknowledgment

This work has been supported in part by **CNPq** (National Science Foundation, Brazil) under contract n. 306619/2015-6 (PQ).

References

[1] http://gaisler.com/index.php/products/processors/leon3 Last access: June 2017.

[2] Reinhard Wilhelm *et al.*, "The Worst-Case Execution-Time Problem – Overview of Methods and Survey of Tools", ACM Trans. On Embedded Computing Systems, vol. 7, no. 3, April 2008.

[3] Jakob Rosén, Petru Eles, Zebo Peng, Alexandru Andrei "PredictableWorst-Case Execution Time Analysis for Multiprocessor Systems-on-Chip", 2011 6th IEEE International Symposium on Electronic Design, Test and Application, pp 99-104.

[4] Sudipta Chattopadhyay, Chong Lee Kee, Abhik Roychoudhury, "A Unified WCET Analysis Framework for Multi-core Platforms", Proc. of RTAS 2012.

[5] Mingsong Lv, Wang Yi, Nan Guan, Ge Yu, Timon Kelter, Peter Marwedel, Heiko Falk, "Combining Abstract Interpretation with Model Checking for Timing Analysis of Multicore Software", ACM Trans. On Embedded Computing Systems, vol. 13, no. 4s, March 2014.

[6] Theo Ungerer *et al.*, "Merasa: Multicore Execution of Real-Time Applications Supporting Analyzability", IEEE Micro, Computer Society, September-October, 2010, pp. 66-75.

[7] http://www.otawa.fr Last access: June 2017.

[8] http://www.rapitasystems.com/products/RapiTime Last access: June 2017.

[9] http://www.windriver.com/products/product-overviews/PO_VxWorks653_Platform_0210.pdf Last access: June 2017.

[10] http://www.parmerasa.eu/index.php?menu=deliverables

[11] Theo Ungerer *et al.*, "parMERASA – Multi-Core Execution of Parallelised Hard Real-Time Applications Supporting Analysability", 2013 16th Euromicro Conference on Digital System Design, pp. 363-270.

[12] GRAPHVIZ. Graphviz. Disponível em: <http://www.graphviz.org/>. Last access: June, 2017.

[13] LP_SOLVE. Lp_solve reference guide. Disponível em: <http://lpsolve.sourceforge.net/5.5/>. Last access: June, 2017.

[14] WILHELM, Reinhard et al. The worst-case execution-time problem—overview of methods and survey of tools. Tecs, [s.l.], v. 7, n. 3, p.1-53, 1 abr. 2008. Association for Computing Machinery (ACM). http://dx.doi.org/10.1145/1347375.1347389.