International Journal of Advancements in Electronics and Electrical Engineering– IJAEEE
Volume 4 : Issue 2 [ISSN : 2319-7498]
Publication Date : 30 October, 2015

Preliminaries on a Hardware-Based Approach to Support Mixed-Critical Workload Execution in Multicore Processors

Fabian Vargas, Bruno Green

***Abstract*— The use of multicore processors in general-purpose real-time embedded systems has experienced a huge increase in the recent years. Unfortunately, critical applications are not benefiting from this type of processors as one could expect. The major obstacle is that we may not predict and provide any guarantee on real-time properties of software running on such platforms. The shared memory bus is among the most critical resources, which severely degrades the timing predictability of multicore software due to the access contention between cores. To counteract this problem, we present in this paper a new approach that supports mixed-criticality workload execution in a multicore processor-based embedded system. It allows any number of cores to run less-critical tasks concurrently with the critical core, which is running the critical task. The approach is based on the use of a dedicated Deadline Enforcement Checker (DEC) implemented in hardware, which allows the execution of any number of cores (running less-critical workloads) concurrently with the critical core (executing the critical workload). This approach allows the exploitation of the maximum performance offered by a multiprocessing system while guaranteeing critical task schedulability. A case-study based on a dual-core version of the LEON3 processor was implemented and mapped in a Xilinx Spartan 3E FPGA.**

***Keywords*— Multicore Processor, Critical Application, High-Performance Embedded System, Critical Task Schedulability, Deadline Enforcement Checker (DEC).**

I. Introduction

It is of common agreement among designers and users that multicore processors will be increasingly used in future embedded real-time systems for critical applications where, in addition to reliability, high-performance is at a premium. The major obstacle is that we may not predict and provide any guarantee on real-time properties of software on such platforms. As consequence, the timing deadline of critical real-time tasks may be violated. In this context, the shared memory bus is among the most critical resources, which severely degrade the timing predictability of multicore workloads due to access contention among cores. In critical embedded systems (e.g., aeronautical systems) this uncertainty of the non-uniform and concurrent memory accesses prohibits the full utilization of the system performance. In more detail, the system is designed in such a way that the processor runs in the stand-alone mode when a critical task has to be executed, i.e., the bus controller allows only one core to run the critical workload, and inhibits the other cores to execute less-critical workloads till the completion of the critical task.

In order to counteract this problem and properly balance reliability against performance as long as possible, we present in this paper a new approach that supports mixed-criticality workload execution by fully exploiting processor parallelism. It allows any number of cores to run less-critical tasks concurrently with the critical core, which is running the critical task. The proposed approach is not based on any multicore static timing analysis or any timing model of multicore processor parts such as pipeline, cache memory and interactions of these parts with shared memory bus. Instead, the approach is based on the use of a dedicated Deadline Enforcement Checker (DEC), which works as follows: when a critical task starts running, the DEC allows the execution of any number of cores (running less-critical workloads) concurrently with the critical core (executing the critical workload) till the moment when the DEC predicts that the critical workload deadline will be violated if the processor continues running all cores concurrently. At this moment, the DEC inhibits the non-critical cores to execute less-critical tasks till the completion of the critical task by the critical core. Then, it is said that the system is switched from the "shared mode" (where two or more cores are fighting for shared memory bus access) to the "stand-alone mode", where a unique (the critical) core is running.

Given the above, the proposed approach presents the following features and advantages compared to the existing techniques:

a) It minimizes the computational complexity imposed by multicore static timing analysis: it needs only to analyze interactions between pipeline and cache models of a single core when executing a given critical task. All *inter-core conflicts* generated during the execution of the critical and the various less-critical tasks caused by their non-uniform and concurrent memory bus accesses are *not* taken into account by the DEC.

b) From the above (a) statement, it is also concluded that the proposed approach does not need the development and it is not based on timing analysis models of multicore processors. Therefore, the approach is not based on the faithfulness of the multicore processor model to guarantee a precise workload timing prediction.

c) In contrast to the existing approaches, the proposed approach does not require any knowledge about the

Fabian Vargas and Bruno Green are with the Catholic University – PUCRS. Electrical Engineering Dept.
Av. Ipiranga, Porto Alegre, Brazil.

implementation of the less-critical workloads or even the number of workloads that will run concurrently with the critical task. The DEC is configured according to specific code structure and timing characteristics of the critical task, as it will be described in Section III. Then, the DEC monitors online the critical task execution and automatically switches the bus usage from the "shared" mode to the "stand-alone" mode to guarantee the maximum possible processor performance with workload schedulability. Note that if the number of less-critical tasks changes, there is no need to re-compute the timing analysis process for the critical task to guarantee workload schedulability. This condition is ensured because the DEC can switch from the "shared mode" to the "stand-alone mode" automatically, no matters is the number of less-critical tasks are running in parallel with the critical one.

d) The approach can be applied to any type of processor, considered that the designer is able to collect two signals from the processor ("*Program Counter*" and "*Annul*"). The latter signal indicates if the current instruction in the pipeline was actually executed or not.

e) Given that the approach can be applied to any type of processor, it allows a large spectrum of real-time operating systems to be used. Thus, traditional and well-stablished real-time operating systems for critical applications such as VxWorks, LynxOS, Integrity or RTEMS and their advanced versions compliant with ARINC-653 (an avionics standard for safe, partitioned systems) [1] could also be considered in the whole system design.

In this work, the terms "task" and "workload" have the same meaning and are used interchangeably. Moreover, we assume that there is only one critical core which is in charge of running the critical task. Concurrently to this core, there can be any number of non-critical cores, each of them executing one or more less-critical tasks. The remainder of the paper is organized as follows: Section II presents the fundamentals of the problem and the existing solutions. Section III describes the proposed approach and the Deadline Enforcement Checker (DEC). Then, Section IV draws the final conclusions of the work.

II. Preliminaries

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].

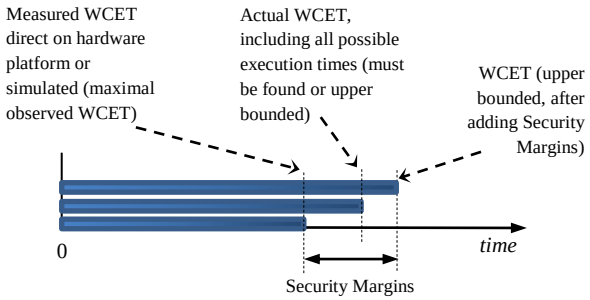

Fig. 1. Basic notions concerning timing analysis of systems.

A lot of research has been carried out within the area of WCET analysis. 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. As response to this specific need, several approaches dealing with WCET prediction in multicore platforms have been proposed [3,4,5,6]. These approaches represent a considerable improvement of the state-of-the-art, but note:

a) They are not trivial in such a way that they must not only analyze all interactions between pipeline and cache models of a single core when executing a given critical task, but they also analyze all inter-core conflicts generated during the execution of the critical and the various less-critical workloads and their non-uniform and concurrent memory bus accesses. This analysis is even more complex when the number of cores running in parallel increases.

b) If the number of tasks changes, the whole process must be recomputed in order to reschedule the tasks into the TDMA (resp. FCFS or Round-Robin) bus slots.

c) It may happen that after predicting the execution of a critical task in a given multicore platform, the designer concludes that this task is not schedulable when executed in concurrence with other (less-critical) tasks. So, the whole analysis is useless and a new analysis process must restart on the basis of a smaller number of less-critical tasks to running in parallel with the critical one. The final goal is to guarantee schedulability of the critical task. If this is not attained yet, then the whole process is restarted again with an even smaller number of less-critical tasks. This "re-do" work is long and complex. So, time consuming.

d) 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 and LEON3 [7] 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) [1] cannot take advantage of this approach yet.

III. The Proposed Approach

Fig. 2 depicts the situation where one critical task (TC) and one less-critical task (T1) are running on two cores. When both tasks are executed (in the "shared mode"), the WCET of the TC violates its deadline D. Thus, the problem is unschedulable (Fig. 2a). However, if TC is executed in the "stand-alone mode", it is schedulable (Fig. 2b). In contrast with existing approaches, where only the critical task is executed at a time in the multicore platform till its completion, the proposed methodology is capable of scheduling the TC by considering the following scenario: initially both tasks (TC and T1) are executed on the system. Then, reference points (RPs) are used to observe on-line the execution time of the TC and decide switching the processor from the "shared mode" to the isolated execution of TC (Fig. 2c). The approach can be applied to any type of processor, considered that the designer is able to collect two signals from the processor ("*Program Counter*" and "*Annul*"). The latter signal indicates if the current instruction in the pipeline was actually executed or dropped down due to speculation-caused timing anomalies of the processor.

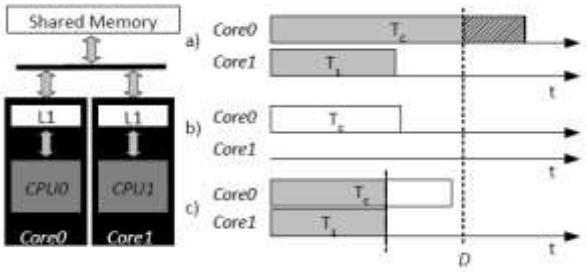

Fig. 2. Scheduling based on WCET when are considered for execution (a) both tasks, (b) only the critical task and (c) proposed approach.

In this work, we propose an approach to improve core utilization by running several tasks in parallel while guaranteeing the critical task schedulability. The target platform can be a TDMA, First-Come First-Served (FCFS) or Round Robin bus-access policy multicore system. We assume a single core to run a unique critical task in parallel with any number of cores running less-critical tasks. If running in the stand-alone mode the critical task is perfectly schedulable. However, if it is running in parallel with other less-critical tasks, it cannot be guaranteed its schedulability, unless the proposed approach is considered. Our methodology is split in two steps:

i) Off-line, we analyze the control-flow-graph (CFG) of the critical task and safely compute the *remaining WCET* at several Reference Points (RPs) of the TC running in the stand-alone mode. The *remaining WCET* ($WCET_R$) is defined as the WCET between the considered RP till the end of the code.

ii) On-line, for the system running in the shared mode, we use a dedicated Deadline Enforcement Checker (DEC) to monitor the real execution time of TC and to check whether there is a risk that the critical task misses its deadline due to system overload. If so, the less-critical tasks are temporarily paused so that the critical task continues in the stand-alone mode from that monitored point till its completion. After the critical task is complete, the less-critical tasks can resume execution from the point they were temporarily paused. The proposed approach is schematically depicted in Fig. 3.

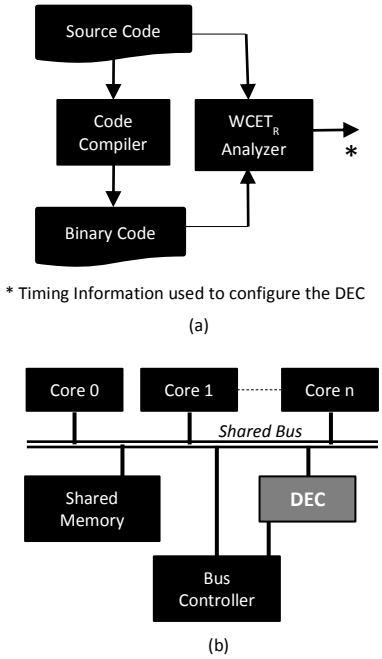

Fig. 3. Overview of the proposed approach: (a) $WCET_R$ computation at the Reference Points (RP) of the TC running in the stand-alone mode; (b) General block diagram of a multicore processor connected with the DEC.

Fig. 4 shows the computed timeline of a critical task. The reference point *zero* (RP_0) is hereby defined as the *start of the task*. The *Critical Time (CT)* of a given *Reference Point (RP)* is given by:

$$CT(RP_n) = DeadlineTime - WCET_R(RP_n) - t_{over}$$

Where $WCET_R(RP_n)$ is the remaining worst-case execution time from the RP_n to the task end instant, which was statically computed for the processor running in the "stand-alone" mode. Finally, t_{over} is a constant cost associated to the bus arbiter to switch the processor from the "shared" to the "stand-alone" mode.

When the critical task starts, the elapsed time is initialized and incremented, clock-by-clock by the DEC and compared against the CT collected at the last RP that it passed by. It should be noted that during the critical task execution, the CPU passes through several code paths with different times (since the path taken along the code is a function of its inputs). Therefore, it is mandatory that the CT be updated at every RP distributed along the code. When the DEC compares the elapsed time against the CT collected at the most recent RP that it passed by, three possible situations may occur:

a) The elapsed time is smaller than the CT; then, the processor continues executing in the "shared" mode.

b) The elapsed time is equal to or greater than the CT; then, the DEC advises the bus arbiter to switch the processor from the "shared" to the "stand-alone" mode.

c) The elapsed time is greater than the "Deadline"; then, the DEC issues an "Error Indication" signal.

It is worth noting that during task execution in the "stand-alone" mode, upon arriving in the current RP, if the DEC detects that the execution of the critical task is faster than predicted according to the last RP that it passed by, the DEC signals to the bus arbiter to switch back to the "shared" mode in order to privilege task concurrency in the multicore platform.

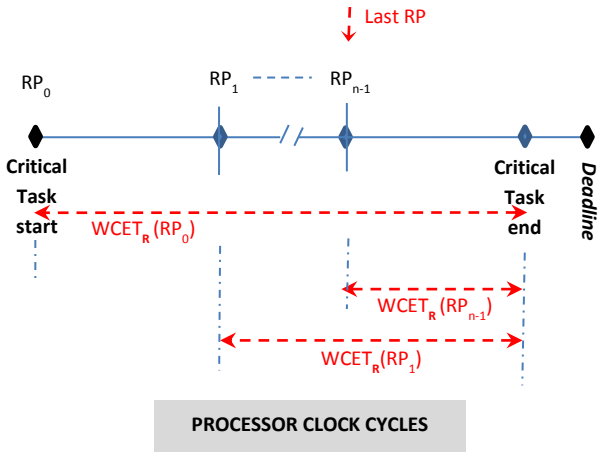

Fig. 4. Timeline of a critical task execution.

Fig. 5 depicts a general view of the DEC internal blocks and their respective signals. The first block is the *Checkpoint Monitor*, whose goal is to **identify the current RP of the task**. This block can accomplish such goal in different ways (depending on the type of the information that is treated by the block) which are briefly described below:

a) The *Checkpoint Monitor* Block observes the instruction address bus for memory reads. By comparing the instruction addresses flowing through the bus, this block detects when monitored core reaches a specific RP. A drawback of such approach is that there is no guarantee that the *instruction* read is actually executed by the processor; for example the instructions fetched just after a *branch* or a *speculative* execution by the CPU. As consequence, it constrains the locations where a RP can be inserted (since a RP cannot be inserted just after a *branch* or an instruction that is speculatively executed). Another problem is that if the processor has an instruction cache, then the address of the executed instruction will not appear in the bus in case of a cache hit.

b) Additionally to observe the instruction address bus, the *Checkpoint Monitor* Block can also inspect the critical core internal signals (such as the program counter - PC). Unlike the previous method depicted in (a), by monitoring the internal signals it is easy to know whether an instruction is executed or not, relaxing the locations in which RPs can be inserted (this approach enlarges the universe of locations a RP can be inserted and so, it is a better option compared to the previous one). Nevertheless, this method is more intrusive.

c) Finally, the *Checkpoint Monitor* Block can receive explicit information about the RP currently reached by the core, directly from processor general purpose I/O pins (e.g., GPIO port of the processor or any other communication channel). In this case the *Checkpoint Monitor* Block of the DEC is omitted and the RP Identification (RP_{ID}) is fed directly to the second block (*Checkpoint Time Controller*). Nevertheless, this method has an important drawback: higher detection latency compared to the methods (a) and (b) described above. Moreover, the critical task must be modified to write the RP_{ID} on the GPIO port of the processor or on any other communication channel. It should be noted that system designers are hesitant about this change in the user code. Although, it has the advantage that it does not need to access any processor internal signal or access to the processor bus. Therefore, it can potentially work with processors in which system designers do not have access to the bus or internal processor signals (for instance, the processor is a "black-box" third-part intellectual property core or even a COTS processor).

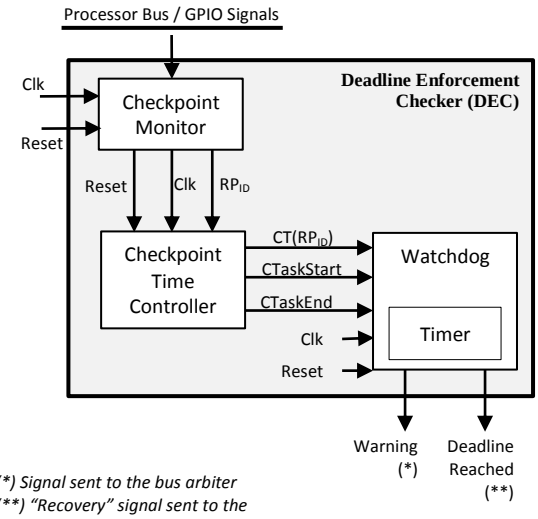

Fig. 5. General view of the DEC internal blocks and their respective signals.

Among the three methods above, (b) is the one currently implemented in the DEC.

The output of the *Checkpoint Monitor* Block is a Reference Point Identification (RP_{ID}) that is fed to the *Checkpoint Time Controller*, whose responsibility is to **correlate a RP_{ID} with its Critical Time $CT(RP_{ID})$**. This mapping process is performed by means of a content-addressable memory (CAM): the CAM input is a RP_{ID} and the output is the corresponding $CT(RP_{ID})$. The *Checkpoint Time Controller* also notifies when the critical task starts running (CTaskStart) and ends (CTaskEnd).

The final block (*Watchdog*) is **responsible for notifying the event "Warning"**. When this event is yielded, the bus controller forces the processor to switch execution from the "shared-bus" mode into the "stand-alone" one (in which the bus is exclusively allocated to the critical core). This block also signals the "Deadline Reached", which yields an "Error Indication" signal. Fig. 6 depicts the control-flow graph (CFG) of the DEC operation.

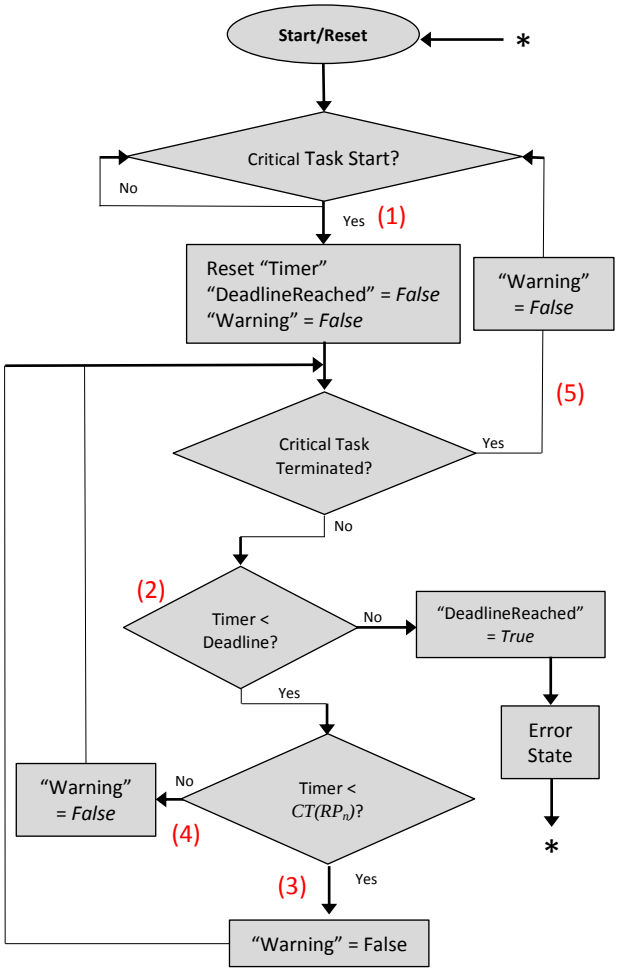

Fig. 6. Control-flow graph (CFG) of the DEC operation.

The indicated points in Fig. 6 are detailed as follows:

(1) By default when a critical task starts running, the processor bus is set on the "shared-bus" mode.

(2) The condition:

"$CT(RP_n) = DeadlineTime - WCET_R(RP_n) - t_{over}$"

is tested and one of the three previously mentioned situations may occur: (a) the elapsed time is smaller than the CT; (b) the elapsed time is equal to or greater than the CT; and (c) the elapsed time is greater than the "Deadline" and then, the DEC issues an "Error Indication" signal.

(3) At this moment, the processor is running in the "shared-bus" mode.

(4) At this moment, the bus arbiter switches processor operation from "shared" to "stand-alone" mode.

(5) Critical task (being executed in the "shared" or "stand-alone" mode) terminates, conditionally that it does not violates deadline.

IV. Final Considerations and Conclusions

We presented a new approach that supports mixed-criticality workload execution in a multicore processor-based embedded system. Given that the proposed approach can be applied to any type of processor, it allows a large spectrum of real-time operating systems to be used. Thus, traditional and well-stablished real-time operating systems for critical applications such as VxWorks, LynxOS, Integrity or RTEMS and their advanced versions compliant with ARINC-653 (an avionics standard for safe, partitioned systems) could also be considered in the whole system design.

The approach allows any number of cores to run less-critical tasks concurrently with the critical core, which is running the critical task. The approach is based on the use of a dedicated a hardware-based Deadline Enforcement Checker (DEC), which allows the execution of any number of cores (running less-critical workloads) concurrently with the critical core (executing the critical workload). This approach allows the exploitation of the maximum performance offered by a multiprocessing system while guaranteeing critical task schedulability.

A case-study based on a dual-core version of the LEON3 processor [7] was implemented and mapped in a Xilinx Spartan 3E FPGA. The measured area overhead is considerably small, in the order of 4.14% for the dual-core version of the LEON3 processor. Furthermore, several critical application codes based on WCET benchmarks [8] were compiled to this processor. The experimental results demonstrated that the proposed approach is very effective on combining system high-performance with critical task schedulability within timing deadline.

Acknowledgment

This work has been supported in part by CNPq (National Science Foundation, Brazil) under contract n. 303701/2011-0 (PQ).

References

[1] http://www.windriver.com/products/product-overviews/PO_VxWorks653_Platform_0210.pdf Last access: June 2015.

[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://gaisler.com/index.php/products/processors/leon3. Last access: June 2015.

[8] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET benchmarks – past, present and future. In Björn Lisper, editor, Proc. 10th International Workshop on Worst-Case Execution Time Analysis (WCET'2010), pages 137–147, Brussels, Belgium, July 2010. OCG.