DLINE JOURNALS

Digital Signal Processing and Artificial Intelligence for Automatic Learning

Online ISSN: 2583-5009

DSPAIAL 2024; 3 (1)

https://doi.org/10.6025/dspaial/2024/3/1/23-30

Parallel Software Framework for Time-Critical Core Systems

Vincent Nelis, Patrick Meumeu Yomsi, and Luís Miguel Pinho
CISTER Research Centre, Polytechnic Institute of Porto (ISEP-IPP)
Porto, Portugal
{nelis@isep.ipp.pt}
{pamyo@isep.ipp.pt} {lmp@isep.ipp.pt}

Received: 27 September 2023
Revised: 3 December 2023
Accepted: 14 December 2023

ABSTRACT

The Parallel Software Framework for Time-Critical Many-Core Systems aims to develop a new designframework that allows current and future applications with high performance and real-time requirements to fully exploit the huge performance opportunities of the most advanced many-core processors. We have developed a new timing analysis methodology to estimate themaximum execution time of parallel applications running on a many-core architecture. We discuss briefly the results obtained by running it on thethree project use cases on the many-core development board.

Keywords: *Parallel Software, Software Framework, Time Critical Systems*

1. Introduction

The objective of the European project P-SOCRATES [7] (Parallel Software Framework for Time-Critical Many-core Systems) was to develop a new design framework to allow current and future applications with high-performance and real-time requirements to fully exploit the huge performance opportunities brought by the most advanced many-core processors, whilst ensuring a predictable performance and maintaining (or even reducing) the development costs of the applications. The main outcome of the project is the UpScale SDK [1].

Upscale is a framework for the development of real-time high-performance applications in many-core platforms. The SDK targets systems that demand more and more computational performance to process large amounts of data from multiple data sources, whilst low these performance requirements to be achieved, by integrating dozens or hundreds of cores, interconnected with complex networks on chip, paving the way for parallel computing. Unfortunately, parallelization brings many challenges, by drastically affecting the system's timing behavior: providing guarantees becomes harder, because the behavior of the system running on a multi-core processor depends on interactions that are usually not known by the system designer. This causes system analysts to struggle to provide timing guarantees for such platforms.

UpScale tackles this challenge by including technologies from different computing segments to successfully exploit the performance opportunities brought by parallel programming models used in the high-performance domain and timing analysis from the embedded real-time domain, for the newest many-core embedded processors available.

Most of the current state-of-the-art software techniques for analysis and scheduling assume that the system activities (the tasks) are functionally independent and most of their parameters, including their worst-case execution time (WCET), are known at design time. However, at run-time, the tasks that are co-scheduled on different cores share hardware resources, including caches, communication buses and main memory. Those resources introduce implicit functional dependencies among the tasks, as concurrent accesses to the same resource are not allowed, affecting their timing behaviour. This effect is exacerbated when scaling to many-core architectures. Therefore, current analysis and scheduling techniques cannot be applied as-is and need to be augmented to include all the sources of contention due to the increased number of shared resources.

As part of P-SOCRATES, we have developed a new timing analysis methodology to estimate the maximum execution time of parallel applications running on a many-core architecture. Preliminary results towards this direction have already been presented (see for example [4, 3, 5, 9, 8]) but we will not elaborate on those works in this paper. Our timing analysis methodology has been automated and the corresponding tool is now part of the UpScale SDK. In this paper, we describe the methodology, its automation tool, and we discuss briefly the results obtained by running it on the three project use-cases on the Kalray MPPA-256 many-core development board [2].

2. Overview of the Application Structure and Execution Model

In the P-SOCRATES execution model, the real-time applications start their execution on the host cores of the accelerator (i.e. the IO cores in the Kalray MPPA). To execute faster, they can offload some parts of their computation onto the accelerator at any time during their execution. The offloaded parts are organized in sets of potentially parallel segments of code that may be modeled as a directed acyclic graph (DAG). DAG nodes are implemented with OpenMP tasks with edges representing dependencies among these tasks (implemented with the depend OpenMP clause). We call each offloaded part, a "phase" of the application. Each phase can thus be seen as a graph of openMP tasks that can execute concurrently on different clusters. The focus of our analysis work is on the DAGs, where little previous works exist. Traditional real-time techniques can then be used to integrate the full computation, i.e. the sequential execution on the host cores and the offloaded phases.

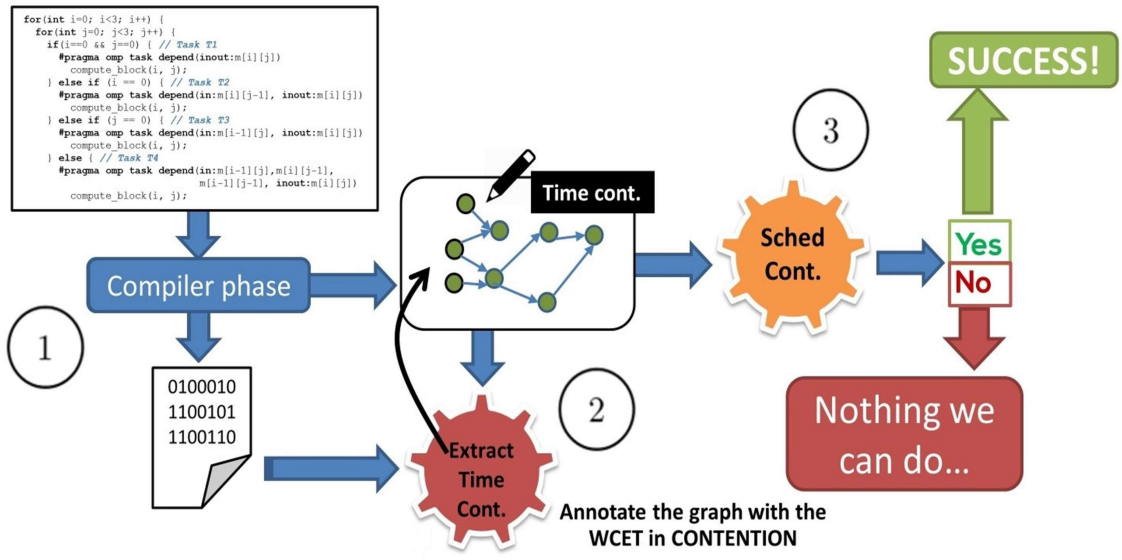

Figure 1. P-SOCRATES analysis flow using dynamic mapping and scheduling

Digital Signal Processing and Artificial Intelligence for Automatic Learning
Volume 3 Number 1 March 2024

To analyze the timing behavior of the applications, we define two different analysis flows: one for systems that use dynamic mapping and scheduling techniques and another one for static techniques. Due to space limitations, we only provide a brief summary of the differences between these two approaches. When dynamic techniques are used, the task-to-thread mapping of the lightweight runtime is performed during execution, based on OpenMP task mapping algorithms, and the scheduling in the operating system uses global migration of threads. Therefore the system adapts better to variability in the load, achieving in principle higher performance (due to better load balancing). However, as variability is higher, less information exists to perform both timing and schedulability analyses, which may lead to pessimistic theoretical worst-case scenarios (which in practice may not occur). When static approaches are used, a specific mapping algorithm assigns the OpenMP tasks to particular threads in the OpenMP runtime, with each thread then statically assigned to a particular core, with the operating system using partitioned per-core schedulers. In this case, the system is less adaptable, in many cases incapable of taking advantage of eventual spare capacity (a core may be idle while tasks wait in other cores), which may lead to worse average performance. However, there is more information on the configuration of the system, which may allow a tighter analysis, leading to improved worst-case scenarios.

3. Analysis Flow for Systems using Dynamic Mapping and Scheduling Techniques

Figure 1 depicts the end-to-end analysis flow. As mentioned above, adopting the dynamic scheduling scheme means that all the OpenMP tasks are not pinned to a specific thread and can migrate from one core to another at runtime. In this case, there is only one queue for all pending/waiting OpenMP tasks. As soon as a core finishes the execution of a task, it starts (or resumes) the execution of the first task in that single shared queue. To analyze this configuration, the implemented steps are as follows.

Step 1: *The application is compiled and the compilation of its source code generates at least two files: a multibinary file – the executable file to be run on the MPPA – and one DAG per application phase. Every phase (i.e., every "#pragma omp target" directive found in the source code that offload parts of the computation to the MPPA accelerator) leads to the creation of a set of omp tasks generated and executed in the accelerator. This set of tasks and their inter-dependencies are represented by a Task Dependency Graph (TDG) created at this point by the compiler. More precisely, the TDG represent the graph of dependencies between all the tasks.*

Step 2: *In this step, we estimate the maximum execution time of every openMP task when it suffers a maximum interference on the shared resources. We call this execution time the MEET of the tasks (Maximum Extrinsic Execution Time). The MEET is thus measured by enforcing the "nastiest" execution environment in which the analyzed tasks suffer as much interference as possible from other tasks and applications running concurrently. In these execution conditions, every analyzed task is executed multiple times over the same set of "worst" input data. This step requires to slightly modify the source code of the application and insert specialized code that will artificially generate those nasty execution conditions at runtime. In short, in this step all the openMP tasks of the application execute sequentially on the same core, by a single thread, while all the other threads execute on the other cores and generate as much interference as possible on the shared resources (memory, NoC, etc.). At the end of this step, every node of every TDG (which represents an OpenMP task) is annotated with its MEET.*

Step 3: *The annotated TDGs and the timing parameters of the applications (period and deadline) are given as input to the schedulability analysis tool to check whether all the timing requirements are met (i.e., no deadline is missed). Since in the case of dynamic mapping and scheduling we do not know beforehand on which core the tasks will execute, nor concurrently with which other tasks, the schedulability analysis must use the MEET as the worst-case execution time of every task, i.e., it assumes a maximum interference on every task. If the schedulability analysis fails, we have no other option than changing the timing parameters of the application (period and deadline) or optimizing its source code. We will see in the next section that we can reduce this pessimism by using a static scheduler.*

4. Analysis Flow for Systems using Static Mapping and Scheduling Techniques

By using a static scheduling scheme, more information is available at design time. For example,

the task-to-thread and thus the task-to-core mappings are known and, for every application, the subset of openMP tasks running concurrently is also known. This additional information allows us to considerably reduce the inherent pessimism of a dynamic scheduler, using a more complex analysis flow (see Figure 2).

Step 1: *The first step is similar to Steps* 1 *and* 2 *for dynamic scheduling. The MEET is computed for every OpenMP task of the TDGs and is annotated to the graphs.*

Step 2: *The mapping tool is used to define a static task-to-core mapping that is schedulable. If the tool finds a valid mapping then the process ends and returns it. Note that the actual execution time of the tasks at runtime can only be lower than their MEET since the runtime interference will be lower. This means that any mapping found at this step that meets all the deadlines while assuming the MEET of every task is also guaranteed to meet all the deadlines for shorter tasks execution times.*

Step 3: *If the mapping tool does not find any mapping that meets all the timing requirements, then we check if there exists a feasible mapping in the opposite scenario, i.e. while assuming the best execution conditions. That is, we measure in this step the maximum execution time of every openMP task when there is no interference whatsoever on the shared resources. To do so, similarly to the extraction of the MEET, we run all the openMP tasks sequentially on one core, by a single*

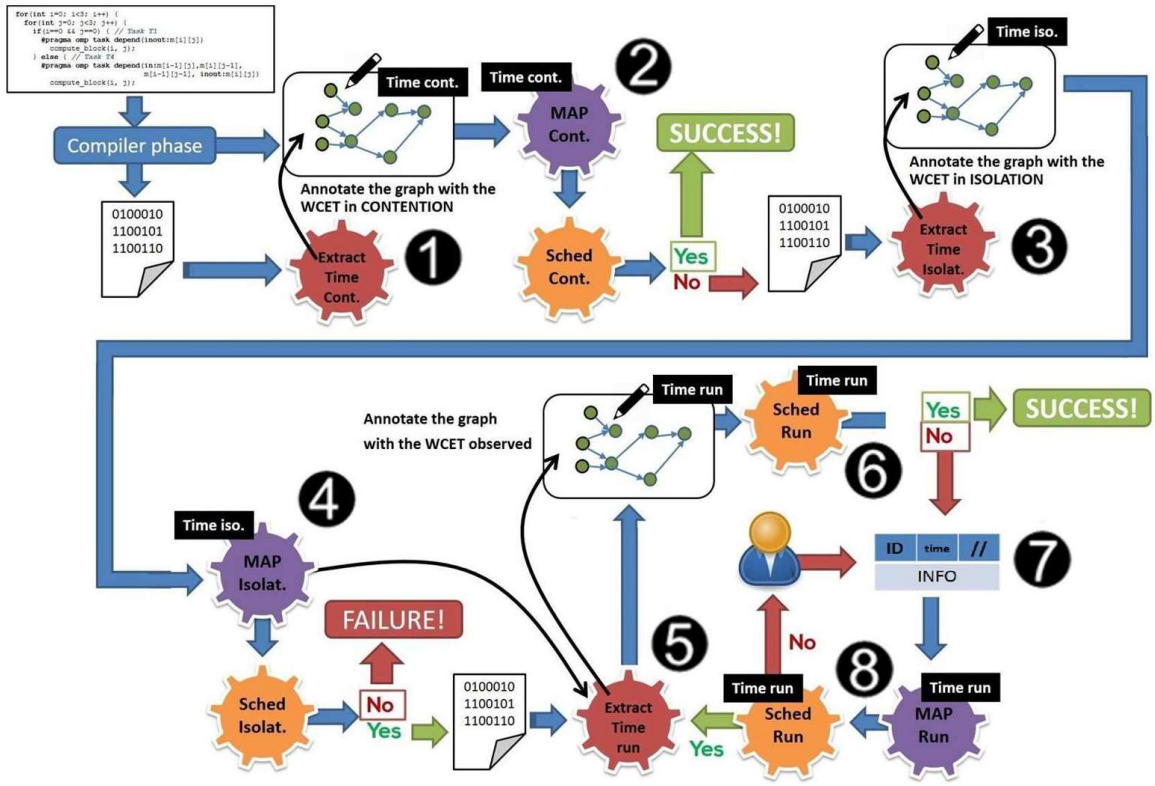

Figure 2. P-SOCRATES analysis flow using static mapping and scheduling

thread, but this time all the threads running on the other cores stay idle (busy-waiting) and do not use any resource. We call this maximum execution time the "MIET" of the tasks (Maximum Intrinsic Execution Time). At the end of this step, the MIET of every OpenMP task is annotated to the TDGs. The reason for measuring the MIET of the tasks is because in the case of static scheduling, we know that some tasks will not execute concurrently with some others (if they are mapped to the same core for example) and thus assuming the MEET was the most pessimistic assumption we could make. The MIET, on the other hand, is the most optimistic one.

Step 4: *This step is similar to step 2. The mapping tool tries to derive a mapping that meets all the timing requirements but contrary to step 2, it assumes that all the OpenMP tasks execute for their MIET (instead of using the MEET estimates as in step 2). If it does not find a feasible mapping then the process stops. In this case, there is no way to find a feasible mapping since the tool did not even find one assuming the best execution conditions, i.e. no interference whatsoever between the OpenMP tasks. The application is thus not schedulable. If the tool did find a valid mapping, then we need to check whether that mapping still meets all the timing requirements even with the interference that will occur at runtime, which is the next step.*

Step 5: *This step runs the application by using the static mapping found at step 4 and records the maximum actual execution time of every OpenMP task (also referred to as MAET – Maximum Actual Execution Time). Those MAETs are also added to the TDGs information.*

Step 6: *The schedulability analysis tool is now used to check if the mapping that has been tested at the previous step is still schedulable when assuming for each OpenMP task its maximum actual/observed execution time (MAET) recorded at Step 5. If this is the case, then the process stops and returns the mapping. Otherwise, the process moves on to Step 7.*

Step 7: *At this stage, we know that the mapping defined at step 4 meets all the timing requirements if the tasks do not execute for longer than their MIET. However, when executed at step 5, it turned out that some of the tasks took longer to complete (as potentially expected) and with those new estimations of their execution times (i.e., the MAETs), the mapping failed the test at step 6. In this case, we display a table containing for each task: its MIET, MAET, MEET, and the set of tasks it has potentially been executed concurrently with (i.e., the subset of tasks allocated to a different core). If a task has a MAET closer to its MEET than to its MIET, we can conclude that the execution of that task is strongly impacted by the tasks that execute concurrently. Hence, it may be wise not to execute them on different cores. On the contrary, if its MAET is closer to its MIET, it means that the task is barely affected by the execution of the concurrent tasks and it may be judicious not to change that task-to-core allocation. During that step, we let the user register a "restriction", telling the mapper that in the next mapping to be produced a particular task cannot be assigned to the same core as the set of tasks it is currently mapped with. Those restrictions will be taken into account by the mapping tool at the next step 8 when trying to find another mapping.*

Step 8: *The mapping tool is used again to derive a new mapping. This time the tool considers the restrictions defined by the user at step 7 and assumes the tasks execution times obtained at step 5, i.e. the MAETs. If the tool still does not find any valid mapping, then the user may try another set of restrictions with less, or different constraints. If this time a valid mapping is found, then we must re-test it, i.e., we must re-extract the maximum execution times (i.e., the MAETs) observed at runtime for that particular mapping and re-check the schedulability of the system while assuming those new MAETs. That is, the process goes back to Step 5 with the mapping obtained at this step.*

Except for these last two steps of defining task restrictions to guide the mapping, which requires manual intervention, the whole analysis process has been automatized within the analysis tool presented in the next section. The feedback loop could also be automatized, but there is yet no known heuristic for the mapping which is able to converge to a stable state.

5. Our P-SOCRATES Timing Analysis Tool

Figure 3 is a screenshot of the tool developed to run the entire timing analysis methodology for a given application. The tool is written in Python 2.7 and is now part of the UpScale SDK, available at [1]. It offers a generic interface to easily implement, connect, and run together different application scripts, possibly written in different programming languages. The tool is based on three main concepts: commands, actions, and variables.

The **commands** *are the basic blocks of the tool. A command is defined in a specific programming language and has a pre-defined type. A command is typically a small script that is used to perform a specific operation and its type describes how it must execute it. For example, one can define the commands "Upload the source code to the testing device" or "Compile the code remotely*

on that device". The former is of type "SFTP – Put" and its code lists the files to be uploaded (following a pre-defined syntax), whereas the latter can be a Shell script of type "SSH - Remote

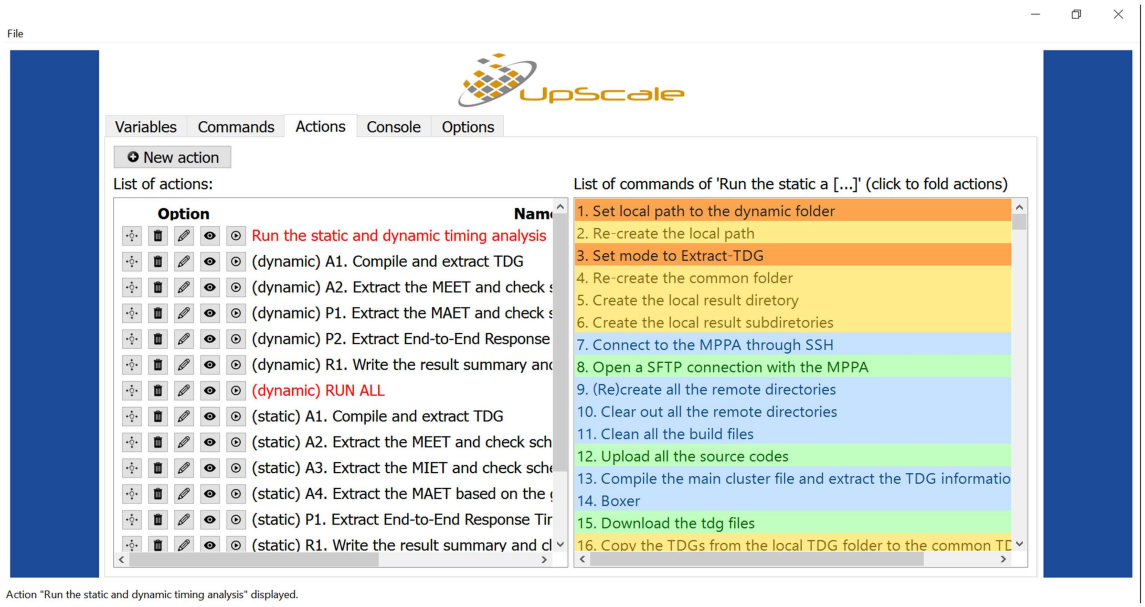

Figure 3. Screenshot of the P-SOCRATES analysis tool-chain that is part of the UpScale SDK.

command[s]", meaning that it will be executed by a remote terminal through SSH. The current version of the tool supports scripts written in Python, R, as well as in any shell script supported by the machine running the script. Files can be sent and received through SSH or SFTP and Shell scripts can be executed remotely through SSH.

*The **variables** are defined by the user. They are simply characterised by a name, a type, and a value. Before each command is executed, the tool performs a simple "search and replace" on the code of the command to replace every reference to a variable with its value. Therefore, variables can be used and accessed by every command, irrespective of its programming language, simply by referring to it as "@{variable name}" in the command's code.*

*The **actions** are the means to connect the commands together. Each action is defined as a set of commands and executing an action simply runs all its commands in the order defined by the user. Note that the tool also provides special control-flow commands that allow to implement loops and simple conditional statements. Those control-flow commands are kept relatively simple as the ambition of the tool is (for now) not to design a new programming language.*

6. Results and Conclusions

Our timing analysis methodology has been tested on three use-case applications at the end of the project: a pre-processing sampling application for infra-red detectors, an online semantic analysis tool, and a complex event processing engine. Although it is difficult to draw general conclusions from the analysis of these use-cases, the results did provide information which allows to reason on the static and dynamic scheduling and mapping approaches proposed in the project, as well as on the use of the overall analysis flow. It is infeasible in this short paper to summarize the three application use-cases down to a level of details that would allow the reader to verify our conclusions, or to make his own. This is why we simply summarize here the main results that we obtained and the conclusions we made.

One focus of our analysis was on the difference between the MIET, MEET, and MAET of the openMP tasks. That is, we wanted to evaluate the gap between the maximum execution time measured for every openMP task in situations where (a) there is no interference at all on the

Digital Signal Processing and Artificial Intelligence for Automatic Learning
Volume 3 Number 1 March 2024

shared resources (MIET), (b) there is an extreme contention for the shared resources(MEET), and (c) the tasks are executed normally (globally or statically) and their inter-task interference is thus accounted for (MAET).

In [6], we showed that on the Kalray MPPA-256 (Andey) many-core platform the interference generated by concurrent tasks can slow down the execution of a given task by a factor 8, i.e. a task's MEET may be up to 8 times higher than its MIET. In the second half of the P-SOCRATES project, we change the board from a Kalray MPPA Andey to a Kalray MPPA Bostan because the company officially announced that they will no longer support the Andey. Unfortunately, those slow-down factors of 8 could not be reproduced on the Bostan. The reason is that on the Andey, the task interference generated when measuring the MEET of a given task was artificially created by (1) allocating data in the same memory banks as the data of the analyzed task, and then (2) repeatedly accessing those data at runtime. This way we were saturating the controllers of the memory banks used by the task under analysis, thereby slowing down its overall execution. On the Bostan board, we were no longer able to apply this approach because the linker scripts that allowed us (in the Andey) to allocate data to specific banks were not yet implemented at the time on the Bostan.

An interesting observation that we made while analyzing the execution time of the openMP tasks running on the Bostan board is that, by repeatedly calling the built-in "printf()" function from the other cores (that do not run the analyzed tasks), we were able to generate a near-starvation scenario. That is, for some tasks, the execution was taking around 49 milli-seconds (with interference) against 117 micro-seconds (without interference) and 124 micro-seconds (when executed normally). This means a slow-down factor or more than 400. Note however, that this result may be due to the internal implementation of the printf function that may temporarily freeze all the cores for some debugging reasons. We have not yet investigated further the reasons for such a slow-down factor.

The results obtained in this analysis allowed to note that static approaches are more time predictable than the dynamic ones. Although for a large number of tasks static mapping and scheduling approaches may experience lower performance at the runtime due to their by-nature conservativeness (thus increasing the actual response time of the application), their prediction on the worst-case response time are tighter and more accurate. The work also leads to conclusions on the need to research on more accurate interference analysis, which leads to less pessimism than worst-case.

References

[1] P-SOCRATES Consortium. UpScale-sdk. Retrieved from http://www.upscale-sdk.com/. Last accessed November 30, 2016.

[2] Kalray Corporation. (2015, April 8). Kalray. Retrieved from http://www.kalrayinc.com.

[3] Fonseca, J. C., Nélis, V., Raravi, G., Pinho, L. M. (2015). A multi-DAG Model for Real-time Parallel Applications with Conditional Execution. In *30th Annual ACM Symposium on Applied Computing* (pp. 1925–1932). Salamanca, Spain.

[4] Maia, C., Bertogna, M., Nogueira, L., Pinho, L. M. (2014). Response-time analysis of synchronous parallel tasks in multiprocessor systems. In *22nd International Conference on Real-Time Networks and Systems* (pp. 3:3–3:12). Versailles, France: ACM.

[5] Melani, A., Bertogna, M., Bonifaci, V., Marchetti-Spaccamela, A., Buttazzo, G. C. (2015). Response-Time Analysis of Conditional DAG Tasks in Multiprocessor Systems. In *27th Euromicro Conference on Real-Time Systems* (pp. 211–221).

[6] Nélis, V., Yomsi, P. M., Pinho, L. M. (2016). The Variability of Application Execution Times on a Multi-Core Platform. In *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)* (Vol. 55, pp. 6:1–6:11). Schloss Dagstuhl – Leibniz-Zentrum fuerInformatik. doi:10.4230/OASIcs.WCET.2016.6.

[7] P-SOCRATES – Parallel Software Framework for Time-Critical Many-core Systems. (n.d.). Retrieved from http://www.p-socrates.eu.

[8] Pinho, L., Nélis, V., Yomsi, P. M., Quiñones, E., Bertogna, M., Burgio, P., ... Mardiak, M. (2015). P-SOCRATES: a Parallel Software Framework for Time-Critical Many-Core Systems. *Microprocessors and Microsystems (MICPRO), 39*(8), 1190–1203. Elsevier.

[9] Serrano, M. A., Melani, A., Vargas, R., Marongiu, A., Bertogna, M., Quiñones, E. (2015). Timing characterization of OpenMP4 tasking model. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)* (pp. 157–166).