Runtime Performance Evaluation of a Non-Preemptive Cooperative Multithreading Framework Through Tracing

Lea Jungmann, Zain A. H. Hammadeh, Jan Sommer, Daniel Lüdtke
Institute for Software Technology
German Aerospace Center (DLR)
Braunschweig, Germany
lea.jungmann@dlr.de, zain.hajhammadeh@dlr.de, jan.sommer@dlr.de, daniel.luedtke@dlr.de

***Abstract*—In the aerospace and automotive domains, there is a growing trend towards delegating more tasks to embedded software, employing sophisticated algorithms and machine learning-based solutions. As a result of this trend, the complexity of embedded software is escalating rapidly. Classical performance analysis methods, such as static worst-case execution time analysis, struggle to cope with this complexity without providing prohibitively over-approximated upper bounds.**

In this paper, we introduce a tracing-based performance analysis approach tailored to data flow space applications. We illustrate how traces are leveraged to extract arrival curves, minimum distance functions, and execution times. We showcase the utility of tracing in design decisions using an aerospace use case, e.g., optimising the number of cores to reduce end-to-end latency. Furthermore, we extracted and presented debugging information graphically. While our tracing-based performance analysis may introduce overhead on the extracted timing properties, such as worst-case execution time, this overhead is bounded by 6.5%. Finally, we demonstrated the efficacy of our proposed tracing-based analysis approach through its application in a space application scenario.

I. Introduction

Modern space applications, including Earth observation, in-orbit servicing, and autonomous spacecraft and rover missions on distant celestial bodies, entail intensive on-board data processing and sophisticated control algorithms. These applications can become very complex, with high requirements for reliability and performance. The high demand for small satellites, such as cube-sats, necessitates more modular and reusable software that meets mission requirements, including timing requirements.

Multi-core platforms can offer high performance with low power consumption compared to single-core platforms. However, the parallel execution and simultaneous access to shared resources on multi-core platforms introduce additional complexity to embedded software. Furthermore, reading from sensors involves a significant time delay relative to computing time. Although self-suspending processes are employed for sensor reading, they contribute to more intricate and less predictable timing models. Event-driven execution models,

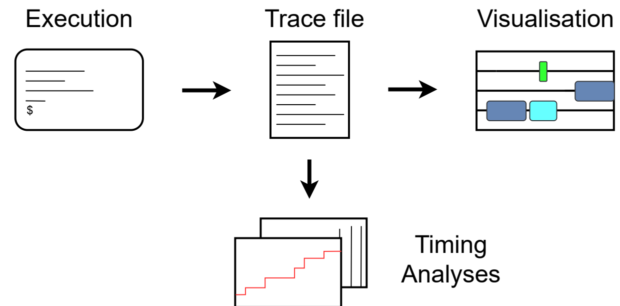

Fig. 1. Workflow of the tracing discussed in this paper: the execution of a program equipped with tracing results in a trace file which can either be graphically displayed or further analysed.

such as the publisher-subscriber model in Robotic Operating System version 2 (ROS2) [1], are also commonly employed for improved data predictability. In this case, a common industry practice is to assign tasks to a pool of threads where the threads cooperate to execute the tasks under a non-preemptive manner [1], [2]. Nevertheless, these models do not simplify timing considerations; instead, they introduce their own complexities.

Measurement-based performance analyses are widely utilized in the industry; however, they cannot guarantee complete coverage of all corner cases. Static methods, on the other hand, can offer formal guarantees on performance and are primarily employed for safety-critical applications. Despite providing over-approximated results, static methods struggle to smoothly scale with the complexity of modern hardware/software. Contemporary research often leans towards proposing hybrid approaches to address the heightened complexity of modern hardware/software and compute reliable guarantees. Tracing emerges as a versatile approach to extract crucial runtime information from complex embedded software to enhance formal methods. For instance, tracers have been used to define the activation pattern of tasks in the Real-Time Calculus (RTC) [3] approach and the Symbolic Timing Analysis for Systems

(SymTA/S) [4] approach. This technique finds application in various domains, such as robotics [5] and automotive [6], where it aids in extracting timing properties and establishing precedence relations between software components. Tracing is preferred over regular debugging, as the latter may lead to breakpoints violating timing requirements or skewing the observed performance.

In this work, we aim to extract timing properties of applications with complex timing behaviour, namely applications executed by cooperating thread pools. For that end, our proposal utilizes a tracing mechanism. Also, we present how to visualise these traces, and how to extract debugging information from them, using open-source tools. An overview of our proposed workflow is summerised in Fig.1. Our work employs the Common Trace Format (CTF) [7] to write traces. and the TraceCompass [8] to visualise the traces. Also, we use Babeltrace [9] to extract timing properties from the traces. We implemented our proposed tracing mechanism on an event-driven multithreading framework, namely Tasking Framework [2]. As our applications are intended to run on different operating systems, primarily Linux and RTEMS, we are focused on developing a cross-platform tracing mechanism.

In the following section, we explore the related work. In Section III, we briefly introduce Tasking Framework and its main features. In Section IV, we elaborate on the implementation of the tracing mechanism. Section V presents our approach to extract timing properties. The overhead of the proposed tracing mechanism is discussed in Section VI. We demonstrate the applicability of the proposed approach on a realistic case study in Section VII. Finally, Section VIII concludes the paper.

II. State of the Art

The extraction of runtime information is vital for the developing process. Knowing the execution behaviour is key to debugging and, later on, the optimisation of a system. An established way of extracting runtime information is tracing. Tracing records the behaviour of a system during its execution by placing hooks, called tracepoints, in the code [10]. At its core, tracing produces a trace file that can be read and analysed after it is produced [11]. Depending on the tracepoints and their eventual content, the runtime information gained with tracing can vary, depending on the observed system. If the system structure is not known or only known partially, the focus when extracting runtime information may lie in getting a more complete system model such as in [12], [13]. In other cases, such as in [5], runtime information, such as response times, is collected to aid in analysing timing behaviour.

Facing unknown behaviour in a real-time system, [12]'s approach uses execution traces and a task definition to model the system's runtime behaviour as a set of independent periodic tasks. All tasks occurring in a given execution trace are categorised as either periodic or non-periodic. Furthermore, they extract additional information on the periodic tasks such as their period and response time profile.

Usually, traces are extracted using Tracers, tools that use already existing or custom hooks to instrument the code for recording during runtime. A popular tracer for Linux applications is the Linux Trace Toolkit: next Generation (LTTng) [14], that is capable of tracing processes both in the kernel and user space. For kernel tracing, it uses tracepoints already embedded in the Linux kernel. For user space tracing, LTTng needs the application to be instrumented using either LTTng-style tracepoints or Java or Python logging statements that are then fed to a LTTng handler. LTTng uses a binary format called Common Trace Format (CTF) [7] to write its traces in a compact manner. LTTng is also used as a basis for other tracing tools, such as in [5], which presents a range of multi-purpose tracing tools for the ROS 2 that use LTTng as their tracing backend. LTTng is used because it has both user and kernel space tracing capabilities, making the trace as comprehensive as possible for Linux applications, as well its low overhead and real-time compatibility.

[13] employs the extended Berkeley Packet Filter (eBPF) for tracing in ROS 2. Other than LTTng, it does not require direct instrumentation, which would lead to having to recompile ROS2 standard libraries. Tracing is used in [13] to extract the flow of information within the system, since this information may not be directly accessible in industry scenarios. It does so by identifying and tracking ROS 2 nodes and callbacks during execution.

LTTng only runs on Linux systems, making it unsuitable for cross-platform applications. However, the format that LTTng uses, CTF, is an open standard that is intelligible to both industry tools such as Tracealyzer [15] and open-source solutions like TraceCompass [8]. Its binary nature makes CTF a very compact format already, its high flexibility regarding form and content of individual events, instances of tracepoints being passed, allows to control the amount of overhead and size of the resulting trace file.

To cope with the emerged challenges from using new programming language like Rust, Wang et al. proposed in [16] a context aware tracing for estimating the execution time of asynchronous tasks. The main concern of [16] is the Rust programs that are implemented as coroutines. Hence, the applicability of [16] is limited to Rust asynchronous programs.

III. Tasking Framework

The Tasking Framework is an open-source[1] non-preemptive, cooperative multithreading C++ framework and execution platform, mainly used in the development of space applications [2]. It is being developed by the German Aerospace Center. While it supports different platforms such as Linux,

[1] https://github.com/DLR-SC/tasking-framework

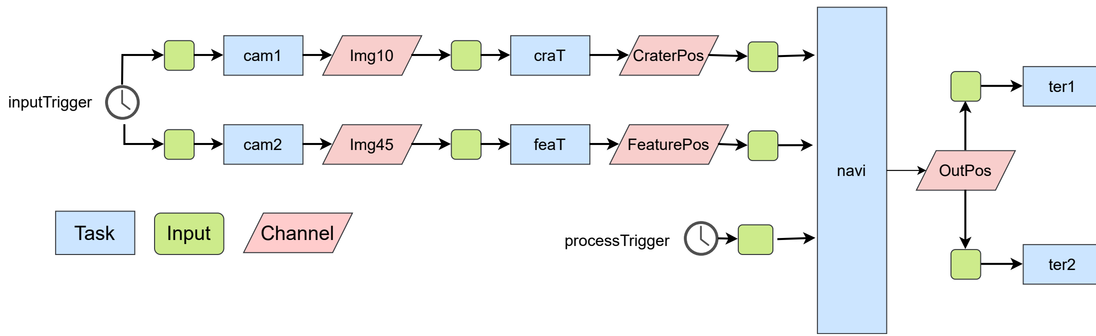

Fig. 2. The optical navigation subsystem in ATON [17] in Tasking Framework.

RTEMS and FreeRTOS, it can also be run on bare-metal. Tasking Framework has been used in several space projects, including Autonomous Terrain-based Optical Navigation (ATON) [17], Euglena Combined Organic food Production In Space (Eu:CROPIS) [18], and Scalable On-Board Computing for Space Avionics (ScOSA) [19].

In Tasking Framework, applications are modelled as a graph of tasks, channels and inputs, as can be seen in Fig.2, which models the optical navigation subsystem of ATON. This task and channel model is modelled after Petri nets, with tasks analogous to transitions and channels likened to places. As with Petri nets, channels and tasks are connected through inputs. Channels can be thought of as data storage while tasks are processing units that take their input from and push their output to channels. Once data is pushed on a channel, the inputs that connect tasks to the channel are notified of the new data on the channel. This may lead to the activation of the connected tasks. In addition, there are also events. Events are used to either periodically trigger a task or to trigger the task after a time-out.

The point in time at which a task is activated, that means marked as ready to be executed, depends on the activation model used for this task. Tasking Framework supports different activation models, meaning the conditions of activation can be chosen individually for each task. Thus, a task may wait for a push on all, one or some channels that it is connected to or may require multiple pushes on a channel before activation. The activation model for a task is chosen at compile time, however, the task barrier structure, a specialised kind of channel, may be used if the amount of pushes required for task activation has to be changed during runtime. The default call semantic for tasks is asynchronous, however, the task group structure can be used to implement synchronicity among a group of tasks, meaning that a task once executed can only be executed again after all other tasks in the group have also executed regardless of its own activation status.

When a task is activated, it will be queued for execution. Tasks are executed using a pool of threads, called *executors*, that collaborate on the execution of tasks. Tasks are executed by Tasking Framework in *non-preemptive* manner. Fig. 3 illustrates the execution model in Tasking Framework. There are three scheduling policies supported in Tasking Framework, namely First-In First-Out (FIFO); Last-In First-Out (LIFO); Fixed Priority. The scheduling is work-conservative, i.e., there is no idle executor as long as the ready queue is not empty. Executors collaborate in a load-balancing manner and every task can be executed by any available executor.

Currently, the application programming interface (API) of Tasking Framework supports only C++. Developing applications using different programming languages is not supported up to now.

IV. Implementation

This section outlines how a tracing mechanism was integrated into Tasking Framework and how it can be displayed with TraceCompass [8]. Tracing the Tasking Framework is reliant on the instrumentation of its code, i.e. the hooks placed in the source code to record a Tasking Framework application. We use the Common Trace Format (CTF) [7], which is a flexible and lightweight binary format, to write traces. The Tasking Framework is large enough that it would cause too much overhead to record every single action that is executed during the runtime. Thus, one must identify a configuration of points within Tasking Framework that give an accurate picture of the inner happenings of the framework. Preferably, with as few points as possible as to avoid causing too much overhead. These tracepoints are:

1) A push on a channel. The push on a channel happens whenever new data is made available to the channel. In turn, all connected inputs are notified informing the connected tasks that new data is available on the channel.
2) Activation of a task. The activation of a task signals that a task is ready to be executed and has been queued by the scheduler to wait for the next free execution slot.
3) Task starts & stops executing. This shows how long the task had to wait before being executed and how long it was executed.

Pushes are triggers for task activation and their presence or absence in a trace can contribute to error searches. The timing

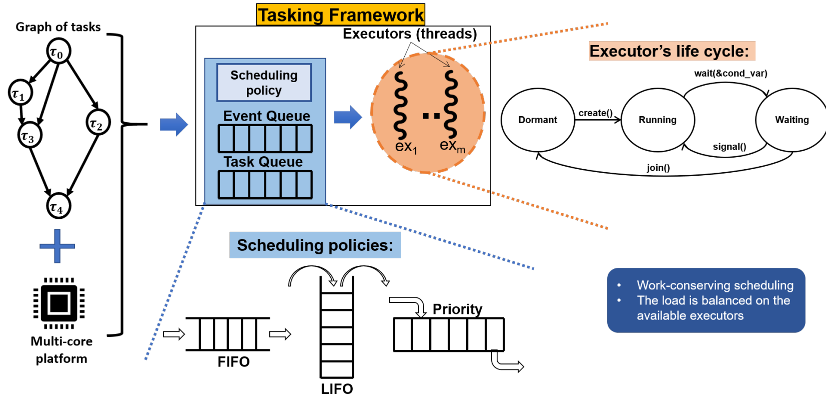

Fig. 3. The execution model in Tasking Framework.

information on the activation, start and stop of the execution of a task can provide information on execution and queueing wait times. The tracer class is implemented as a singleton to prevent conflicting write processes, especially when multiple threads are running and generating events at the same time. Customising the trace format also gives control over the amount of overhead produced by the tracer because custom events can use only the exact amount of data needed and do not have to fill fields with empty data to conform to standards. Custom events pose the question of how exactly their payloads are supposed to look like. Keeping the payload small is imperative to keeping the overhead small. Each tracepoint generates an event upon execution. Hence, the four custom events are:

- push on a channel λ_l, denoted by π_l
- activation of a task τ_i, denoted by α_i
- start of a task execution, denoted by $\sigma_{\uparrow i}$
- stop of a task execution, denoted by $\sigma_{\downarrow i}$

We introduce the trace θ as a finite set of these events.

Each of these events requires at least the identification number of the relevant task or channel in order to match the events to their corresponding tasks or channels. While this is enough to complete necessary calculations and calculate corresponding graphs, these graphs are not particularly readable for humans. The number associated to each task is not a speaking name and would require the developer or user to look up the numbers in Tasking Framework in a time-inefficient manner. To prevent this, the payload of each custom event includes not only the identification number of a task but also its four-character name that is used for display purposes and for user interaction.

We use TraceCompass [8] to visualise our traces. TraceCompass is an Eclipse Rich Client Platform (RCP) tool to read, visualise and analyse traces. TraceCompass provides a variety of charts. These charts allow for inspecting, measuring and analysing the opened trace. Like all Eclipse RCPs, TraceCompass can be modularly expanded with the help of plug-ins to add more functionality. Plug-ins for TraceCompass include plug-ins for additional analyses, scripting, global filters and support for additional trace types by different tracers and profilers. Since TraceCompass is built and specialised on Linux kernel and user space traces, it does not include many charts for custom traces. In fact, a completely custom CTF trace imported into TraceCompass will get two charts generated by TraceCompass, the Statistics chart, that shows the absolute and relative frequencies of the events in the trace, and a list of all events and their payloads contained in the trace, that is by default chronologically ordered. While manageable for very small traces, these two charts are not very helpful when used with traces that contain more than a handful of events. Additionally, while CTF does not have a limitation on the length of individual traces, there is a limit of about 1.6 million events that can be loaded into and displayed in TraceCompass. For traces containing more events, there are other tools such as Babeltrace [9], that are able to handle trace files of that size.

Making a custom graph is possible using a Python script and the EASE scripting module integrated into TraceCompass. The script takes the currently opened trace as input and iterates over the events. Each event whose name can be matched to one of the defined custom events is used to extract its *quark*, which in TraceCompass stands for a unique identifier for an object. In this case, the quark is generated from the event name, meaning the events are sorted by task or channel name. The event is then added to the state system using the quark and the timestamp to sort it to the right position. Once all eligible events are added to the state system, it is used to create a *TimeGraph*. TimeGraph lists all states, in this case tasks and channels, on the left side of the diagram while using a timeline as x-axis. This custom TimeGraph will be referred to as *Tasking Graph*. An example of a Tasking Graph can be seen in Fig.4, which displays the Tasking Graph of a trace of a skeleton implementation of the ATON optical navigation subsystem shown in Fig.2. This means, each task or channel

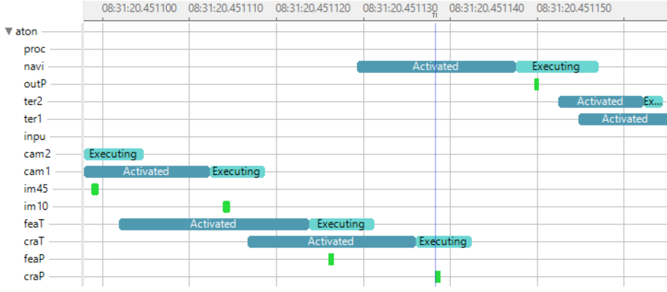

Fig. 4. Tasking Graph of the use case shown in Fig.2.

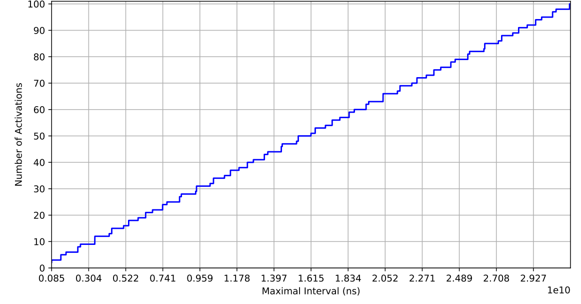

Fig. 5. Arrival curve $\eta^+(\Delta t)$ of the navigation task (navi) of ATON.

gets its own individual timeline that displays how the object changes states during the runtime. Activation and Execution periods for each task (light and dark blue respectively) are displayed on task timelines, while channel timelines display pushes (green).

V. Extracting timing properties

Next to the graphical representation of the trace, which can be useful for debugging, the generated trace can give more insights into the system behaviour. This information can then be used to decide whether a system needs traffic shaping or reconfiguring. The tracepoints on task activation and start of execution help to determine how long a task has to wait in each instance before it gets executed while the start and end times give insight to the execution time of a task. Let t^{event} represent the timestamp of the $event$.

The instance k of τ_i experiences a queuing time q_i^k:

$$q_i^k = t^{\sigma_{\uparrow i}} - t^{\alpha_i} \tag{1}$$

Hence, the maximum queuing time that τ_i suffers is:

$$Q_i = max\{q_i^k | \forall k \in \theta\} \tag{2}$$

The instance k of τ_i experiences an execution time c_i^k:

$$c_i^k = t^{\sigma_{\downarrow i}} - t^{\sigma_{\uparrow i}} \tag{3}$$

The longest observable execution time of τ_i is:

$$C_i = max\{c_i^k | \forall k \in \theta\} \tag{4}$$

When tracing the activations and executions of a task, one can also analyse the trace to study possibly emerging patterns in the task behaviour and use them to predict system behaviour. For systems that run, ideally, in perpetuity or for very long stretches of time, tracing can only offer a snapshot of the system behaviour. However, graphical analysis is not the only analysis that can be applied to a trace. The recorded events of a Tasking Framework trace allow for the extraction of the following information: execution time of every instance of a particular task, activation times and queuing times, as well as push behaviour.

In practice, we use Babeltrace and its python bindings to iterate over the trace. This allows us to extract execution and queuing times, as well as compute arrival curves and distance functions.

A. Arrival Curves

An arrival curve is a function that can be applied to a trace or any other timeline of events. The minimum and maximum arrival curves $\eta^-(\Delta t)$ and $\eta^+(\Delta t)$, are defined as functions on $\mathbb{R}^+ \to \mathbb{N}^+$, so that for any half-open time interval $[t, t + \Delta t)$ they return respectively either the minimum or maximum number of task activations α that can occur within the interval [20], [21]. An example of a maximum arrival curve can be seen in Fig.5. Arrival curves are *non-decreasing*, with $\eta^+(\Delta t)$ being *sub-additive*, meaning that the following is always true for $\eta^+(\Delta t)$:

$$\forall \Delta t, \Delta t' \in \mathbb{R}^+ : \eta^+(\Delta t + \Delta t') \leq \eta^+(\Delta t) + \eta^+(\Delta t') \tag{5}$$

B. Distance Functions

Distance functions are the pseudo-inverse of arrival curves. The minimum (*maximum*) distance function $\delta^-(n)$ (respectively $\delta^+(n)$) is defined on $\mathbb{N}^+ \to \mathbb{R}^+$ and returns the smallest (*largest*) time interval Δt that contains at least (*at most*) n events. An example for a minimum distance function can be seen in Fig.6. Minimum distance functions are non-decreasing and super-additive [21], meaning that every minimum distance function fulfils the following:

$$\forall n, n' \in \mathbb{N}^+ : \delta^-(n) + \delta^-(n') \leq \delta^-(n + n') \tag{6}$$

C. Extrapolating Trace Data

Exploiting the sub-additive and super-additive properties of $\eta^+(\Delta t)$ and $\delta^-(n)$, it is possible to extrapolate data for the behaviour of a traced system under the following assumption: the $\delta^-(2)$, which was observed within the trace, is also the global minimum. With this assumption in mind, distance functions can be extrapolated as follows:

$$\delta^-(n+1) = \delta^-(n) + \delta^-(2). \tag{7}$$

While arrival curves can be extrapolated with:

$$\eta^+(\Delta t + \Delta t') = \eta^+(\Delta t) + \eta^+(\Delta t') \tag{8}$$

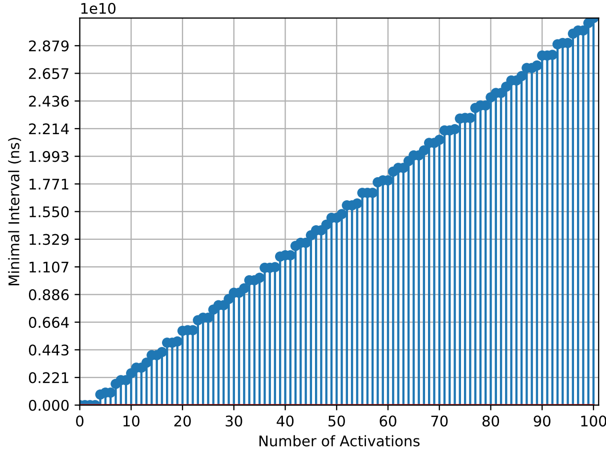

Fig. 6. Distance function $\delta^-(n)$ for the navigation task (navi) of ATON.

When using this extrapolation, it has to be noted that this is a safe extrapolation. Meaning, the extrapolated data has to be treated as lower or upper bound for the distance function and arrival curve respectively, meaning in case of a distance function that the extrapolated values represent the smallest possible value with no indication of an upper limit. Meanwhile, for the arrival curve the extrapolated data represents an upper bound with no indication for a lower bound.

VI. Overhead of Tracing

To measure the overhead caused by the tracing mechanism, example applications were profiled, resulting in a detailed overview of how much time the applications spent in which function. These example applications use dummy tasks, which are tasks with negligible computational effort, to replicate real-world applications in their structure. The offline overhead, i.e., the overhead of all functions that are involved in tracing, lies at around 15%-20%, depending on the profiler that is used. More relevant, however, is the overhead that is actively interfering with the timing behaviour of the tasks. The overhead of the tracing functions that affect the timing behaviour is caused by three functions, namely the ones that record pushes on a channel and the start and stop of a task executing. Their overhead sums up to 5.8%-6.5%, depending on the used profiler. Note, that the execution times of the tracing function are mostly fixed, since they always write the same amount of data. That means, if a task executes longer, the percentage of the overhead goes down accordingly. We computed the overhead using almost empty dummy tasks. Hence, the overhead shown here is likely to be an upper boundary of the possible overhead.

VII. Use Case

We are considering a use case inspired by the optical navigation subsystem of the ATON project, utilizing the Linux operating system and ARM Cortex-A53 (1.2 GHz) as an embedded quad core processor. The aim of the experiments in this section is to showcase the effectiveness of our proposed tracing mechanism as a performance analysis approach, focusing on extracting the timing properties of our use case. We conducted three experiments to exemplify how our tracing-based performance analysis is seamlessly integrated into the design process of safety-critical applications.

We have recorded a trace of 83523 events and extracted arrival curves and distance functions for event-driven tasks. Figures 5 and 6 illustrate $\eta^+(\Delta t)$ and $\delta^-(n)$ respectively for the navigation task (navi). Since the task is event-driven, its activation pattern is non-periodic. Our use case represents a graph of tasks. Therefore, we are interested not only in the timing properties of each task but also in the timing properties of different *chains* defined within the graph, specifically the *end-to-end latency*. We define a *chain* as the execution of a sequence of tasks from a *source* to a *sink*. In our use case, cam1 and cam2 are sources, and ter1 and ter2 are sinks. Hence, there are four chains. Let χ denote a chain, thus:

χ_1 : cam1 $\rightarrow$ craT $\rightarrow$ navi $\rightarrow$ ter1
χ_2 : cam1 $\rightarrow$ craT $\rightarrow$ navi $\rightarrow$ ter2
χ_3 : cam2 $\rightarrow$ feaT $\rightarrow$ navi $\rightarrow$ ter2
χ_4 : cam2 $\rightarrow$ feaT $\rightarrow$ navi $\rightarrow$ ter1

A. Design decision 1: Platform

The goal of this experiment is to demonstrate the capability of our tracing mechanism to be cross-platform. Hence, beside the above mentioned settings (Linux + Cortex-A53), we compiled our case study to run on RTEMS using the GR712RC board with LEON3 processor (40 MHz), which is the default radiation-hardened processor for space systems. We present the execution time and queuing time experienced by each task considering FIFO scheduling with one executor in Fig. 7 for Linux + Cortex-A53 platform. Also, in Fig. 8 we consider FIFO scheduling with one executor for RTEMS + LEON3 platform. The end-to-end latency for both platforms is presented in Fig. 9. RTEMS, as an RTOS, produces results with less variation and suffering from less interference thanks to the RTOS kernel. Therefore, the range between the minimum and maximum values is smaller compared to the results obtained from the Linux OS. However, the maximum observable execution times are not improved, nor the queuing times. In fact, the third quartile values using the RTEMS + LEON3 platform for both the execution time and the queuing time are about 100 times larger than the third quartile values using Linux + Cortex-A53. The main reason is the very slow radiation-hardened processor (LEON3 with 40 MHz) which is about 30 times slower than the high-performance platform (Cortex-A53 with 1.2GHz). The need for more on-board processing power is a major concern for researchers and space companies. Many missions aim to integrate high-performance commercial off-the-shelf (COTS) processors alongside the radiation-hardened

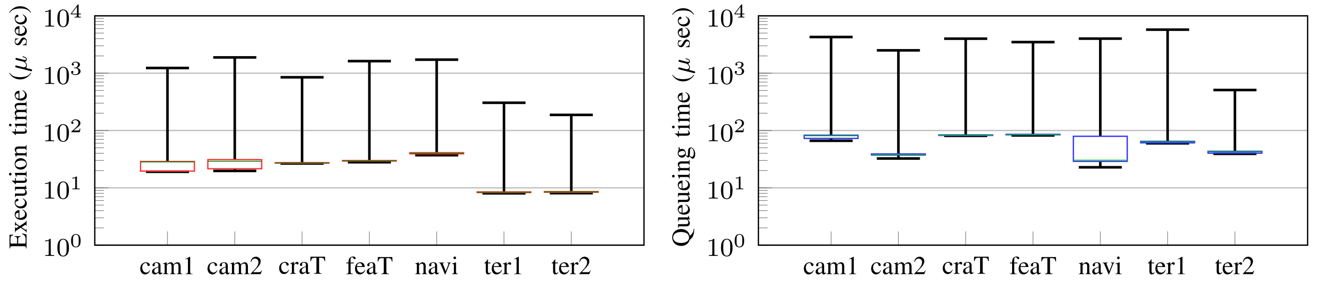

Fig. 7. The execution time and queuing time of the tasks in the use case under the FIFO scheduling using one executor on the Linux + Cortex-A53 platform.

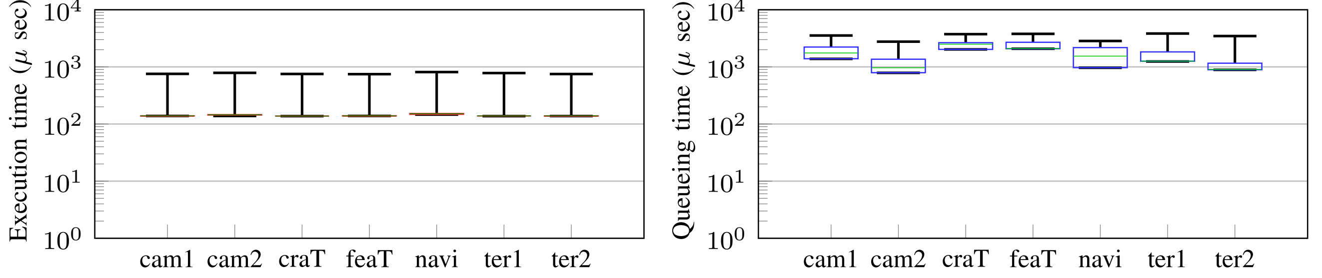

Fig. 8. The execution time and the queuing time of the tasks in the use case under the FIFO scheduling using one executor on the RTEMS + LEON3 platform.

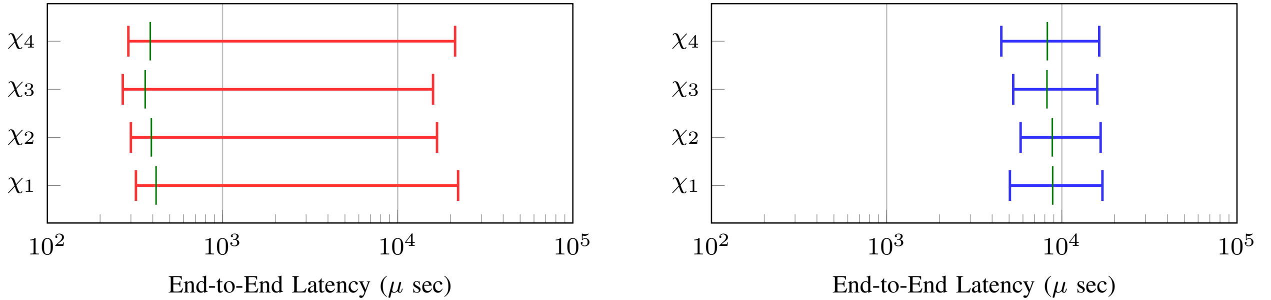

Fig. 9. The end-to-end latency of the chains under the FIFO scheduling considering one executor. In the left figure (in red), we consider the Linux + Cortex-A53 platform, and we consider the RTEMS + LEON3 platform for the right figure (in blue).

processors [22]–[24] to meet the required on-board processing power.

B. *Design decision 2: Scheduling policy*

In this experiment, we utilize our tracing mechanism to study the impact of scheduling policies and priority assignments on the timing behavior of tasks. Accordingly, we generated a new trace on Linux + Cortex-A53 architecture considering fixed priority scheduling, with priorities assigned as outlined in Table I[2]. The new results are presented in Fig. 10. As the Tasking Framework executes tasks in a non-preemptive manner, the queuing time of tasks may exceed their execution time. However, employing priority scheduling reduces the queuing time and improves task execution times.

[2]We refer here to the FIFO and priority scheduling implemented in the Tasking Framework, as depicted in Fig. 3

TABLE I
PRIORITY ASSIGNMENTS WHERE 1 IS THE HIGHEST PRIORITY

Task	cam1	cam2	craT	feaT	navi	ter1	ter2
Priority	1	1	3	2	4	5	5

Consequently, the end-to-end latency is enhanced. Fig. 11 illustrates the end-to-end latency for FIFO scheduling (depicted in red on the left) and priority scheduling (shown in blue on the right). Under FIFO scheduling, data processed from cam1 to ter1, in χ_1, experience the longest end-to-end latency. Conversely, under priority scheduling, χ_2 exhibits the longest latency.

With this experiment we show that it is possible to extract enough data using tracing to come to a sound decision regarding scheduling policy. In this paper, unless stated otherwise, we use FIFO scheduling for all other experiments.

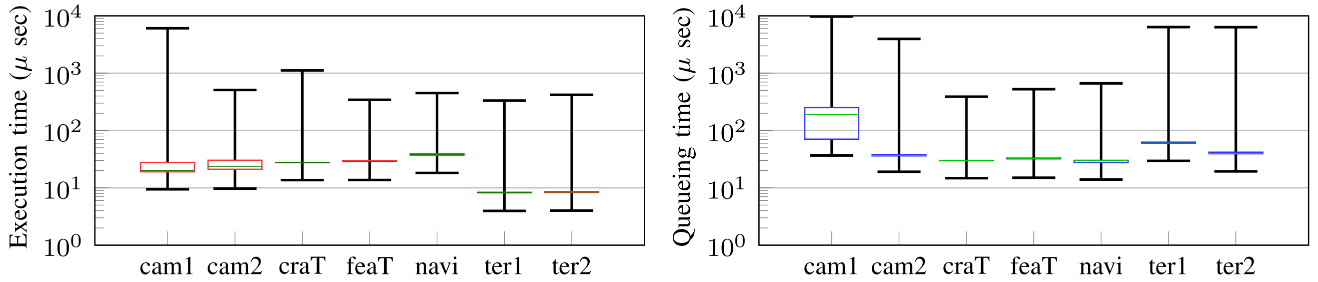

Fig. 10. The execution time and the queuing time of the tasks in the use case under the fixed priority scheduling using one executor on Linux + Cortex-A53 platform.

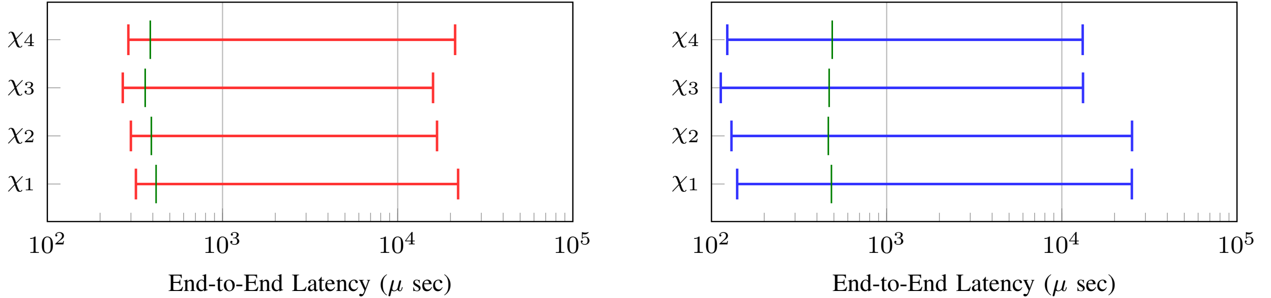

Fig. 11. The end-to-end latency of the chains considering one executor on the Linux + Cortex-A53 platform. In the left figure (in red), we consider the FIFO scheduling, and we consider the priority scheduling for the right figure (in blue).

C. Design decision 3: Number of executors

We aim to address a design question: how many cores should be allocated to the ATON optical navigation subsystem to strike a balance between delay and the number of cores? To achieve this, we executed our code under FIFO scheduling, considering two and three executors, respectively. The results are presented in Fig. 12 and Fig. 13. Comparing the results of one executor (Fig. 7) with two executors (Fig. 12), we observed that the execution times were better with two executors, but the maximum values increased. This can be attributed to the increased ratio of *cache misses* that tasks may experience when executed by two different executors on different cores. Consequently, the maximum queuing time of the tasks also increased significantly, although the minimum values improved. Fig. 14 demonstrates that the minimum end-to-end latency improved compared to the scenario with one executor under FIFO and priority scheduling. However, the maximum end-to-end latency increased significantly.

D. Comparing with static methods

Industrial embedded software are complex and formal methods cannot cope with it. Using languages like C, C++, RUST makes the analysis even more complicated. For instance, using virtual methods in C++ leads to indirect jumps, beside the indirect jumps caused by the switch-case statements and functions pointers in C and C++. Also, the objected-oriented programming in C++ makes bounding the loop more challenging for tools depend on the source code like oRange [25]. Solutions that use dynamic symbolic execution, like e.g. DELOOP [26], can help us to resolve indirect jumps and compute safe bounds on the bounded loops. In [26], the dynamic symbolic execution was used to compute flow facts for Tasking Framework. These flow facts were forwarded to OTAWA [27] to compute the WCET of the Tasking Framework functions, for instance, the push function. The main drawbacks of dynamic symbolic execution based solutions that they are platform dependent. As DELOOP was developed for armv7 architecture, it cannot be used out of the box to compute the execution time for, e.g., X86 architecture. Using portable tracing based solution like our proposed solution can overcome the challenges emerged from different programming languages and it is platform independent. Table II shows the WCET of the push function in all tasks of our use case for ARM Cortex-M3. The results in Table II are more pessimistic than the results computed using the traces because they consider the longest execution path in the push function, which may not observable in the trace.

TABLE II
RESULTS OF THE WCET ANALYSIS FOR THE PUSH FUNCTION IN THE USE CASE

Task	WCET (cycles)
cam1	2435
cam2	2435
craT	3635
feaT	3635
navi	4800

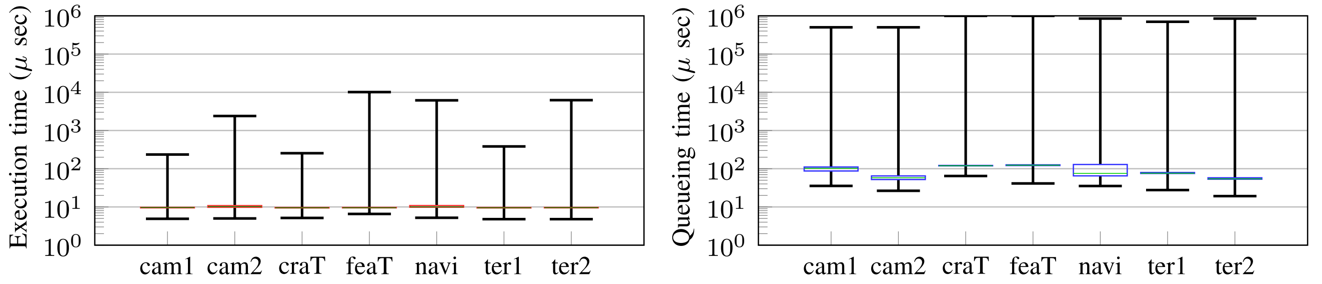

Fig. 12. The execution time and queuing time of the tasks in the use case under the FIFO scheduling using two executors on the Linux + Cortex-A53 platform.

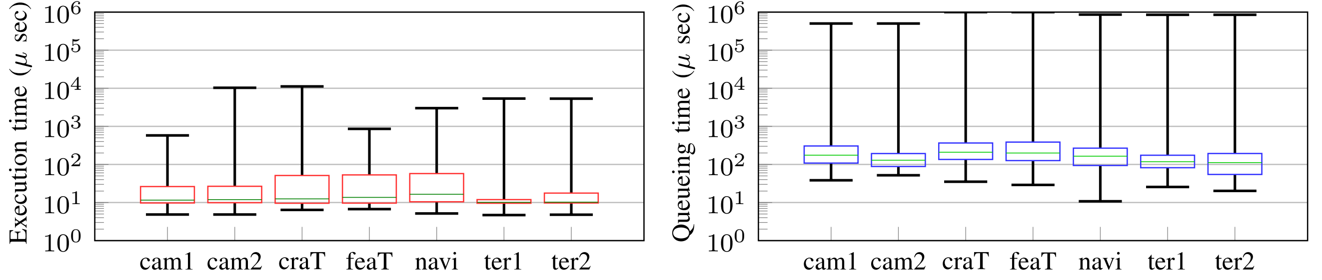

Fig. 13. The execution time and queuing time of the tasks in the use case under the FIFO scheduling using three executors on the Linux + Cortex-A53 platform.

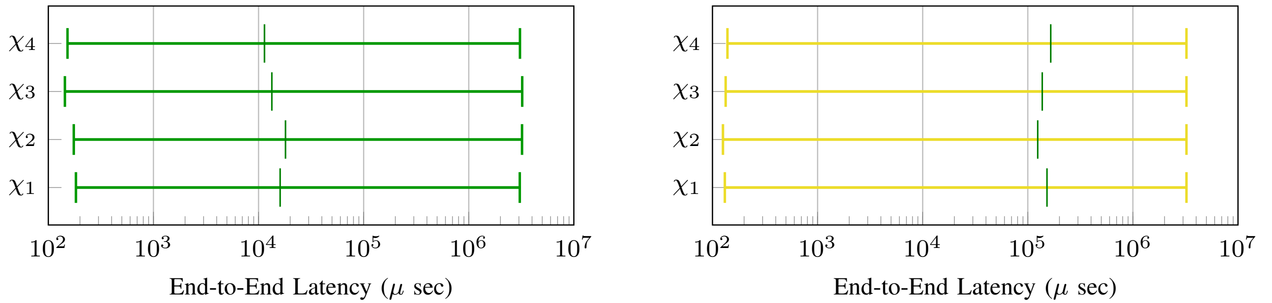

Fig. 14. The end-to-end latency of the chains considering the FIFO scheduling on the Linux + Cortex-A53 platform. In the left figure (in green), we use two executors, and we use three executors in the right figure (in yellow).

VIII. Conclusion and Outlook

With the increasing complexity of embedded software, especially on-board software, performance analysis using static methods faces the challenge of providing tight yet safe guarantees. Extracting timing properties using tracing is a promising technique to assist static methods to cope with the growing complexity of embedded software. In this work, we presented a tracing mechanism for performance analysis of data flow space applications that reuses open-source tools to offer a cross-platform solution. We showed how to use our solution to extract debugging and timing properties of a use case inspired by the optical navigation subsystem. Also, we studied the overhead of our solution.

Any tracing solution suffers from two main points: 1) the need for code instrumentation, 2) the overhead of the events. As eliminating the two points is not realistic, reducing the overhead or the impact of the overhead on the measured parameters is a topic for future improvements. Additionally, code instrumentation can be automated using auto-code generators to guarantee less error-prone instrumentation. For this, we aim to employ the Timing Modeling Language (TML) [28] to automatically instrument the auto-generated code for our applications. In such a step, the developer can generate the traceable code and resulting traces with minimum effort and minimum human errors. Fig. 15 illustrates the TML modeling interface that would make such an automated approach possible.

References

[1] D. Casini, T. Blaß, I. Lütkebohle, and B. B. Brandenburg, "Response-time analysis of ROS 2 processing chains under reservation-based

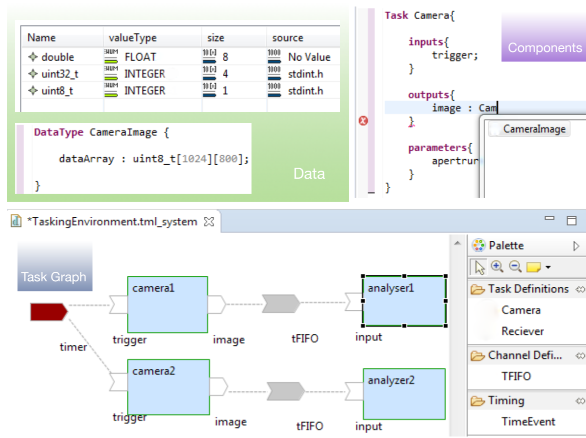

Fig. 15. The Timing Modeling Language (TML): an auto-code generator for Tasking Framework.

scheduling," in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, vol. 133. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019, pp. 6:1–6:23.

[2] Z. A. H. Hammadeh, T. Franz, O. Maibaum, A. Gerndt, and D. Lüdtke, "Event-driven multithreading execution platform for real-time on-board software systems," in *Proceedings of the 15th annual workshop on Operating Systems Platforms for Embedded Real-time Applications*, July 2019, pp. 29–34. [Online]. Available: https://elib.dlr.de/128249/

[3] S. Chakraborty, S. Kunzli, and L. Thiele, "A general framework for analysing system properties in platform-based embedded system designs," in *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*, ser. DATE '03. USA: IEEE Computer Society, 2003, p. 10190.

[4] R. Henia, "System level performance analysis – the SymTA/S approach," *IEE Proceedings - Computers and Digital Techniques*, vol. 152, pp. 148–166(18), March 2005. [Online]. Available: https://digital-library.theiet.org/content/journals/10.1049/ip-cdt_20045088

[5] C. Bédard, I. Lütkebohle, and M. Dagenais, "ROS2_tracing: Multipurpose low-overhead framework for real-time tracing of ROS2," *IEEE Robotics and Automation Letters*, vol. 7, no. 3, pp. 6511–6518, 2022.

[6] S. Quinton, T. T. Bone, J. Hennig, M. Neukirchner, M. Negrean, and R. Ernst, "Typical worst case response-time analysis and its use in automotive network design," in *Proceedings of the 51st Annual Design Automation Conference*, ser. DAC '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1–6. [Online]. Available: https://doi.org/10.1145/2593069.2602977

[7] M. Desnoyers. Common trace format (ctf) specification (v1.8.3). [Online]. Available: https://diamon.org/ctf

[8] Tracecompass documentation. [Online]. Available: https://www.eclipse.org/tracecompass/

[9] The babeltrace 2 documentation. [Online]. Available: https://babeltrace.org/

[10] B. Cornelissen, A. Zaidman, A. Van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," *IEEE Transactions on Software Engineering*, vol. 35, no. 5, pp. 684–702, 2009.

[11] N. Ezzati-Jivan, G. Bastien, and M. R. Dagenais, "High latency cause detection using multilevel dynamic analysis," in *2018 Annual IEEE International Systems Conference (SysCon)*. IEEE, 2018, pp. 1–8.

[12] O. Iegorov, R. Torres, and S. Fischmeister, "Periodic task mining in embedded system traces," in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017, pp. 331–340.

[13] H. Abaza, D. Roy, S. Fan, S. Saidi, and A. Motakis, "Trace-enabled timing model synthesis for ROS 2-based autonomous applications," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2024, pp. 1–6.

[14] The lttng documentation. [Online]. Available: https://lttng.org/docs/v2.13/

[15] Percepio Tracealyzer. [Online]. Available: https://percepio.com/tracealyzer/tracealyzer-for-linux/

[16] T.-Y. Wang, S.-H. Wang, C.-H. Tu, and W.-Y. Liang, "CAT: Context aware tracing for rust asynchronous programs," in *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, ser. SAC '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 483–492. [Online]. Available: https://doi.org/10.1145/3555776.3577669

[17] S. Theil, N. A. Ammann, F. Andert, T. Franz, H. Krüger, H. Lehner, M. Lingenauber, D. Lüdtke, B. Maass, C. Paproth, and J. Wohlfeil, "ATON (autonomous terrain-based optical navigation) for exploration missions: recent flight test results," *CEAS Space Journal*, March 2018. [Online]. Available: https://elib.dlr.de/119557/

[18] O. Maibaum and A. Heidecker, "Software evolution from TET-1 to Eu:CROPIS," in *10th International Symposium on Small Satellites for Earth Observation*, R. Sandau, H.-P. Röser, and A. Valenzuela, Eds. Wissenschaft & Technik Verlag, April 2015, pp. 195–198. [Online]. Available: https://elib.dlr.de/100859/

[19] A. Lund, Z. A. Haj Hammadeh, P. Kenny, V. Vishav, A. Kovalov, H. Watolla, A. Gerndt, and D. Lüdtke, "ScOSA system software: the reliable and scalable middleware for a heterogeneous and distributed on-board computer architecture," *CEAS Space Journal*, vol. 14, no. 1, pp. 161–171, 2022.

[20] S. Künzli and L. Thiele, "Generating event traces based on arrival curves," in *13th GI/ITG Conference-Measuring, Modelling and Evaluation of Computer and Communication Systems*. VDE, 2006.

[21] Z. A. Haj Hammadeh, "Deadline miss models for temporarily overloaded systems," Ph.D. dissertation, Technische Universität Braunschweig, 2019.

[22] G. Lentaris, K. Maragos, I. Stratakos, L. Papadopoulos, O. Papanikolaou, D. Soudris, M. Lourakis, X. Zabulis, D. Gonzalez-Arjona, and G. Furano, "High-performance embedded computing in space: Evaluation of platforms for vision-based navigation," *Journal of Aerospace Information Systems*, vol. 15, no. 4, pp. 178–192, 2018. [Online]. Available: https://doi.org/10.2514/1.I010555

[23] D. Keymeulen, S. Shin, J. Riddley, M. Klimesh, A. Kiely, E. Liggett, P. Sullivan, M. Bernas, H. Ghossemi, G. Flesch, M. Cheng, S. Dolinar, D. Dolman, K. Roth, C. Holyoake, K. Crocker, and A. Smith, "High performance space computing with system-on-chip instrument avionics for space-based next generation imaging spectrometers (ngis)," in *2018 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, 2018, pp. 33–36.

[24] D. Lüdtke, T. Firchau, C. G. Cortes, A. Lund, A. M. Nepal, M. M. Elbarrawy, Z. H. Hammadeh, J.-G. Meß, P. Kenny, F. Brömer, M. Mirzaagha, G. Saleip, H. Kirstein, C. Kirchhefer, and A. Gerndt, "Scosa on the way to orbit: Reconfigurable high-performance computing for spacecraft," in *2023 IEEE Space Computing Conference (SCC)*, 2023, pp. 34–44.

[25] A. Bonenfant, M. de Michiel, and P. Sainrat, "oRange: A tool for static loop bound analysis," in *Workshop on Resource Analysis, University of Hertfordshire, Hatfield, UK*, vol. 9, no. 09, 2008, p. 08.

[26] H. Abaza, Z. A. Haj Hammadeh, and D. Lüdtke, "DELOOP: Automatic flow facts computation using dynamic symbolic execution," in *20th International Workshop on Worst-Case Execution Time Analysis (WCET 2022)*, ser. Open Access Series in Informatics (OASIcs), C. Ballabriga, Ed., vol. 103. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, pp. 3:1–3:12. [Online]. Available: https://drops-dev.dagstuhl.de/entities/document/10.4230/OASIcs.WCET.2022.3

[27] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat, "OTAWA: an open toolbox for adaptive WCET analysis," in *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*. Springer, 2010, pp. 35–46.

[28] T. Franz, A. M. Nepal, Z. A. Haj Hammadeh, O. Maibaum, A. Gerndt, and D. Lüdtke, "Tasking Modeling Language: A toolset for model-based engineering of data-driven software systems," in *OBDP2021 - 2nd European Workshop on On-Board Data Processing*, no. 2, June 2021. [Online]. Available: https://elib.dlr.de/145077/