Heterogeneous Runtime Monitoring for Real-Time Systems with art2kitekt

1st Miguel García-Gordillo
Instituto Tecnológico de Informática
Universitat Politècnica de València
Valencia, Spain
miguelgarcia@iti.es

2nd Joan J. Valls
Instituto Tecnológico de Informática
Universitat Politècnica de València
Valencia, Spain
jvalls@iti.es

3rd Sergio Sáez
Instituto Tecnológico de Informática
Universitat Politècnica de València
Valencia, Spain
ssaez@iti.es

***Abstract*—Monitoring the execution of real-time systems has many advantages, it is not only useful to understand the behaviour of an application but also to find unfulfilled timing constraints in an implementation. However, real-time operating systems usually do not include the tracing tools to observe the behaviour during the execution. This paper presents the art2kitekt runtime monitoring tool, used to measure and to visualise the temporal characteristics of a real-time application. To demonstrate the functionality of the tool, the behaviour of an RTEMS-based application running over a Xilinx Zynq UltraScale+ is observed.**

***Index Terms*—monitoring, tracing, real-time, RTEMS**

I. Introduction

A real-time system is characterised not only by the correctness in the desired functional behaviour, but also by guarantying precise timing in its response times. In case of hard real-time, due to its safety-critical nature, the unavailability or malfunctioning of the system, such as a delay in a response, can produce detrimental effects to its environment and to their users.

The development process must ensure both the expected behaviour and the temporal feasibility of the system. Thread priorities, temporal constraints, and the precedence relationship of the activities are necessary to define a proper operation of the system.

In early phases of the development, schedulability analysis algorithms, combined with static code analysis techniques, are used to define and validate the temporal configuration of the system, in order to comply with the real-time requirements. The analytical processes are typically very complex and need to ensure the feasibility of the system under all scenarios. Hence, they usually provide very pessimistic results that are almost never met in a real execution. The observation of the system can help to improve the results of the schedulability analysis, mixing static analysis with the observed timing behaviour, and also to ease a better understanding of the application.

On the other hand, it is necessary to evaluate the system implementation and compare it with the expected behaviour. Fixing the errors due to an incorrect implementation or finding events that were not taken into account in the designed model are made easier with the utilisation of a monitoring approach.

With that purpose in mind, we are continuously working in the improvement of the art2kitekt (a2k) tool-suite, a collection of tools integrated in a common web-based framework. The aim of these tools is to help the engineers in both the design and the temporal verification of real-time systems using a model-based approach.

In this paper we propose a runtime monitoring tool that extends art2kitekt. This new tool provides the observed runtime statistics, which can be useful to understand the real behaviour of the system and to improve the design of the system under development. The tool also may be used to find unfulfilled timing constraints in difficult cases that cannot be discovered in the analysis.

In order to demonstrate the new functionality, we have developed a solution to monitor an embedded application running in the real-time operating system RTEMS, that we have extended with a trace recorder module. Also, we have selected the Xilinx Zynq UltraScale+ as the hardware platform. It is an heterogeneous processing system comprised of a quad-core Cortex-A53 and of a dual-core Cortex-R5 real-time processing unit, allowing us to execute both the application under test and the trace analyser in different processors but in the same system-on-chip (SoC), reducing the communication latency between them.

The paper outline is organised as follows: Section 2 enumerates the related research work. Section 3 presents the system model employed by the art2kitekt tool suite. Section 4 describes the monitoring tool proposed to be integrated into the tool suite. Section 5 defines the implementation of the runtime monitoring tool in a Zynq Ultrascale+ platform and details the specific solutions reached for this architecture. Section 6 shows the results of a case study using art2kitekt. Finally, Section 7 explains the concluding remarks.

II. Related work

The idea of monitoring the behaviour of a system has been studied for years. Beginning with a set of isolated and simple tests [1] which have been used for measuring the efficiency of a real-time operating system (RTOS), e.g. obtaining both the time consumed by task switching and the speed of interrupt handling. As well, a general framework [2] has been defined for extracting runtime traces from a generic real-time system

and to analyse them to obtain its temporal properties. The different state machine models used to analyse the traces and to compute the aforementioned properties are also defined.

In the embedded domain, monitoring has been mainly applied to Linux-based system, such as the trace recorder for Linux called Hierarchical Scheduling Framework (HSF) [3], a Linux kernel module implemented as a plugin of their real-time scheduler framework (RESCH), and it has the capability of recording events only of the tasks scheduled by RESCH.

Nowadays, the Linux Trace Toolkit next generation (LTTng) [4] has gained popularity. It is a modular tool for tracing that allows integrated analysis of both kernel space and user space in Linux-based systems. It has been evaluated in several studies, such as the analysis framework [5] to extract metrics from real-time application on Linux Systems, using the LTTng tracer. Also, it has been consider in a multi-level trace-oriented analysis approach based on LTTng [6], with the aim of finding the causes of latency problems in software systems.

Moreover, different solutions have been developed in the area of the real-time systems. Such as GRASP [7], a tool-set with the capabilities of tracing, visualising and measuring the behaviour of real-time systems, and implemented in a platform with the μC/OS-II operating system. METrICS is also presented [8] as a measurement environment for multi-core time-critical systems, running on top of the PikeOS RTOS. In another study, embedded real-time systems trace recording have been developed and evaluated [9], contributing with several technical solutions and trade-off considerations.

In particular, at the time of writing, a native RTEMS tracing framework is under development and apparently it will be included in the 5th version of the operating system [10]. The advantage that this tool will offer is to take existing code in compiled format and instrument it in order to log different events and records in runtime, without rebuilding the code from the source and without annotating the source with trace code.

The above studies are designed using software approaches, usually instrumenting both the application and the kernel, with the disadvantage that this kind of methodology increases the execution time of the observed system, in some way altering its behaviour. Trying to avoid this problem, solutions like an external non-intrusive measurement tool based on a Field-Programmable Gate Array (FPGA) [11] can be used for the evaluation of both context switching and external interrupt latency in RTEMS. This type of solutions usually increases the cost of the development platform and reduces the quantity of statistics obtained in each execution.

Other types of studies are focused on the idea of analysing the captured traces and obtain results thereof. Such as the tool CoreTAna [12], that derives an AUTOSAR compliant model of a real-time system from a dynamic analysis of its trace recordings, deducing an abstraction of the systems structure and of the timing behaviour. Or such as a tool [13] focused on detecting problems related to scheduling and priorities, using the traces provided by LTTng on a Linux Preempt-RT Kernel.

III. System model

As mentioned above, the monitoring tool presented in this writing is an extension of a2k, a web-based framework focused on the modelling and on the analysis of real-time systems. The first step in the definition of an a2k project is to create the different abstraction levels of the system, represented by the available models.

First of all, the **platform model** defines the hardware architecture model. A flexible definition of the different platforms types can be made, such as processors or memories. They can be later instantiated and replicated in the design, connecting their instances as necessary to define the entire platform. An important concept in this type of model is the **executor**, every platform instance with the capability of executing software, either a processor or a hardware accelerator.

In a second step, the **application model** allows to describe the behaviour of the system, defining the **flows**, functional groups of **activities** with usually a data-flow relationship. A flow is defined by:

- **Period:** The time interval between two consecutive flow request times, i.e. the instant which the flow becomes ready to be executed
- **Jitter:** The difference between the minimum and the maximum start time of a flow, relative to its request time

A flow also contains a list of activities and the precedence relationship between them, i.e. the sequence in which they must be executed and the possible simultaneities.

An activity is defined by:

- **Offset:** The minimum amount of time, relative to the flow request time, which an activity must wait before starting
- **Deadline:** The maximum time, relative to the flow request time, at which an activity should have completed its execution

Once the platform and the application model have been defined, both are connected using the **deployment model** and the **execution model**, determining the following constraints:

- **Processor Affinity:** The executor instance within which activity can be executed
- **Scheduler Assignment:** The type of scheduler associated to an executor and the scheduling constraints for the activities

Also, the engineer must define the estimated Worst Case Execution Time (WCET) and the estimated Best Case Execution Time (BCET) for every activity in each of its allowed executors.

The observation of the real behaviour of the system, using the proposed runtime monitoring, is useful to verify if the implementation does not comply with the defined constraints, e.g. the response time of an activity exceeds its deadline. Moreover, monitoring can help engineers to adjust the activity execution time or to analyse the behaviour using the observed statistics, defined by the worst case, the best case and the average of:

- **Execution Time:** The amount of time required to execute an instance of an activity in a given processor type

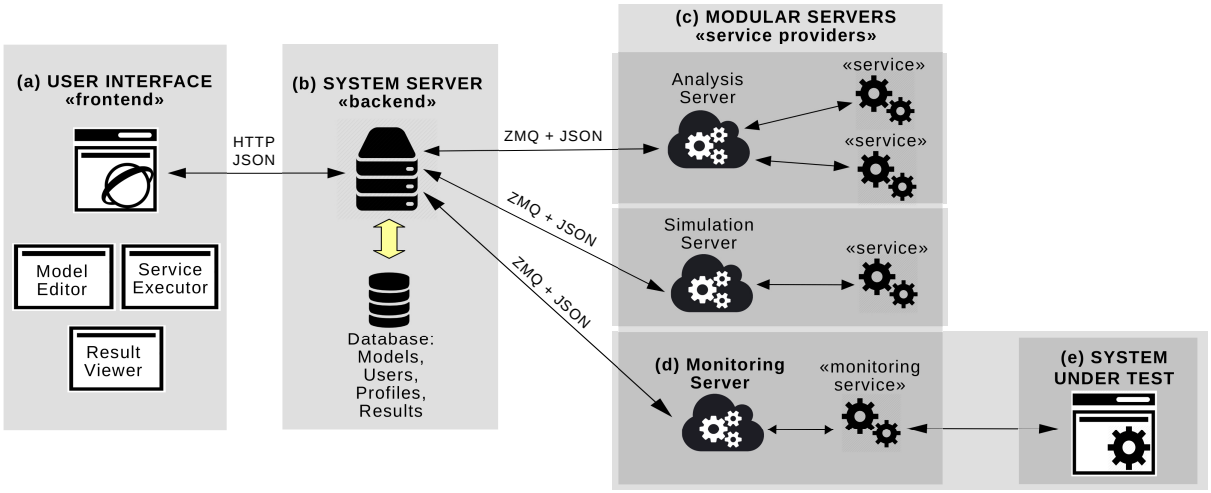

Fig. 1. Architecture of the art2kitekt tool suite

- **Response Time:** The interval between the request time of the flow which the activity belongs to and the instant that said activity is completed
- **Inter-arrival Time:** The time between two consecutive flow request times, i.e. the period in a periodic flow

IV. Tool description

As previously stated, an extension of a2k has been developed with the purpose of observing the real behaviour of the system in runtime. It has been integrated as an independent server and has the capability of communicating with the system to be monitored, an extended version of the system under development.

In order to understand the whole framework (Figure 1), we have divided the explanation in three sections, describing the details of a general design, common to any type of architecture to be observed:

- **art2kitekt tool suite:** Collection of tools that help designing, analysing, simulating, and monitoring real-time systems
- **Monitoring server:** In charge of analysing traces and of generating as a result the observed temporal behaviour
- **System under test (SUT):** The observed system is extended to record temporal traces that will be processed on runtime

The main effort in this development has been made in the monitoring server and in the extension of the SUT. Sections IV-B and IV-C detail the new contributions.

A. art2kitekt tool suite

The art2kitekt tool suite is a collection of web services that are provided for the design of real-time systems. The common part in the tool is composed by both the user interface and the system server.

The user interface (Fig.1(a)) is a web-client that allows to define and to configure the different abstraction layers defined in section III, such as the platform model, the application model or the deployment model. It also permits the engineer to manage the execution of the services and to visualise the results of these executions.

The system server (Fig.1(b)) controls access to the database, that stores the information about models, users and development profiles. It also manages connections to modular servers (Fig.1(c)).

This tool suite has been designed as a distributed system in a flexible framework. It supports the inclusion of new services to the tool suite in different servers, such as the monitoring service explained in this manuscript. These services may use the models previously defined in a2k, combined with new information introduced by the engineer or with the results provided by other services.

B. Monitoring server

The monitoring server (Fig.1(d)), which provides the monitoring service, is in charge of managing the execution of the system observation, of acquiring the traces with the temporal information and of processing the received data to build the statistics of the observation. The following points are a description of the different elements in this server:

1) Monitoring service: It is a service that allows to control the execution of a monitoring test, during the time window in which the SUT is observed. Through the user interface, it allows to establish and release the communication with the backend, to send the configuration from a2k to the SUT, and to start and stop the execution of the monitoring process.

2) Monitoring process: It manages the execution of the monitoring test and is started from the monitoring service. This process is executed until the test duration is reached or a stop command arrives, and is destroyed when it finishes its execution. It is the thread in charge of running the trace collector and the event analyser, and offers the mechanisms to publish the results in a2k.

3) Trace collector: This module is liable for the reception of the traces, usually grouped in long buffers. It must be designed depending on the SUT architecture and on the communication interface used. Besides, it must ensure the order in the reception of the traces and checks the integrity of them.

4) Event analyser: This component, called after the trace collector, analyses the input traces and generates the monitoring statistics, to be published in the a2k result viewers or to be processed by another service. A state machine architecture computes the observed statistics and verify its integrity.

C. System Under Test

The SUT (Fig.1(e)), that will be observed by the monitoring tool, must be extended with the following elements:

1) Tracepoints: To register the behaviour of the system, some instructions, called tracepoints, must be inserted in the application. They should contain enough data to extract the characteristics of the event.

The minimum information required is:

- **Timestamp:** Instant when the event occurs. It could be defined by an absolute time or by a relative time. Additionally, it could be represented by time units, such as nanoseconds, or by clock cycles, that can be converted to time units using the clock frequency. Also, depending on the number of bits used in this field, possible overflows in the timestamp must be taken into account
- **CPU:** Unique identifier of processor unit and core where the event occurs
- **Type:** Different classes of events which describe the action that occurred at that moment in the execution

The points where these tracepoints are usually inserted, in order to help to understand the behaviour, are the following:

- At the beginning and at the end of the activities, defining the code block belonging to this activity
- During the context switch to define which thread is executed in each processor and which one leaves the execution
- At the flow request time to compute the response time of the activities belonging to the flow
- At the beginning and at the end of the interrupt handlers
- At the beginning and at the end of the calls to a hardware resource

2) Trace recorder: This component is called every time that a tracepoint is saved. It is in charge of capturing the timestamp and of saving the traces in a buffer. It must be optimised to reduce the overhead introduced to the application in order to modify the SUT behaviour as little as possible.

3) Monitoring task: A specific thread is usually necessary for managing the communication between the SUT and the monitoring service, and helps to provided the tracepoints to be processed. The priority of this thread should be the lowest or even be implemented in an idle thread if possible.

V. System implementation

In order to demonstrate the new functionality introduced in a2k, the decisions adopted will be explained, such as the board selected or the real-time operating system, also the solution implemented will be detailed, paying special attention to the difficulties found, with a specific subsection for each of them.

The board selected to develop the demonstrator is based on a Xilinx Zynq Ultrascale+, a Multi Processor SoC (MPSoC) composed by:

- Real-Time Processing Unit (RPU), a Dual-core ARM Cortex-R5
- General Application Processing Unit (APU), a Quad-core 64-bits ARM Cortex-A53
- Programmable Logic (PL), a FPGA-Based unit

The point of selecting this kind of heterogeneous MPSoC is to take advantage of the resources shared in the same SoC. The group formed by the RPU and the PL is useful to implement a HW/SW co-design application with real-time capabilities, i.e. the SUT to be monitored by a2k.

The other advantage is to execute a general purpose operating system in the APU, e.g. Linux, and to run the Monitoring Server on it. Executing this part of the Test System (TS) and the SUT in the same SoC allows to communicate them using the shared memory, reducing the latency of the messages and improving the overhead introduced in the SUT, compared with implementing the communication using other type of ports, such as a serial port or an Ethernet port.

Another important decision is which real-time operating system manages the execution of the SUT in the RPU. The authors of this manuscript, and its partners in many projects, usually work with the RTEMS operating system, so it has been the selected RTOS in this system implementation. It is an open source real-time operating system, usually used by the space industry projects, which has been ported to several embedded processor architectures such as ARM, Intel and LEON.

In the following subsections, the focus will be on the details of the solutions adopted in this implementation, considered by the authors as the key points of this development.

A. RTEMS tracing

An intrusive software technique has been chosen to collect the necessary data and to determine the temporal behaviour of the system, i.e. tracepoints must be added to the source code and generate the executable with them, in order to record the events and the time when they are executed.

Param	Timestamp Counter	CPU Id	Event Type	Param 1	Param 2
Byte	0:3	4:7	8:11	12:15	16:19
Size	4	4	4	4	4

Fig. 2. Trace Structure

1) Trace structure: A fixed-size structure (figure 2) has been chosen to ease the management of the tracepoints and of the buffers where they are stored. Additionally to the timestamp, the cpu and the type, the trace includes two more fields with an extra information to complete every type, as indicated in table I.

TABLE I
EVENT TYPE DEFINITION

Event Type	Param 1	Param 2
Thread switch	Swapped out thread id	Swapped in thread id
ISR begin	ISR id	-
ISR end	ISR id	-
Flow release	Flow id	Release id
Activity begin	Activity id	Release id
Activity end	Activity id	Release id
Resource begin	Resource id	-
Resource end	Resource id	-

Usually, the traces are used to monitor the beginning and the end of the execution of an application section, such as an activity or an interrupt handler. The identifiers of each of the monitored sections are static defined in a header file, as part of the traced code. An exception is the release identifier, which is an incremental counter associated to each flow and serves to distinguish the different instances of the same flow.

2) Timestamp counter: In the case of RTEMS, the absolute time from the beginning of the SUT execution is computed in each tick of the operating system, whose frequency depends on its initial configuration, e.g. every 10 milliseconds. The procedure consists of increasing an absolute counter with the difference between the current value and the last stored value of the system clock. Using the function `_Timecounter_Getbinuptime()` of the SuperCore of RTEMS, it is easy to get the time from the beginning, but with an accuracy equal to the frequency of the system tick.

RTEMS provides the `_Timecounter_Binuptime()` function to obtain the time at a particular moment with a higher accuracy, equal to the system clock frequency. It calculates the accumulated time from the last tick plus the difference with the current value of the system clock, which gives a greater accuracy, however, it also increases the overhead.

It is possible to get a timestamp with a greater accuracy without increasing the cost. This solution requires the utilisation of the operating system clock directly and uses the RTEMS `rtems_counter_read()` function. This clock is a free-running counter and usually runs with a frequency close to the CPU, in this case a 32-bit counter running at 499.995 MHz.

With this configuration, the counter will turn back to zero every 8.59 seconds, adding a requirement to the tracing system: a tracepoint must be executed before this time is elapsed, otherwise it will not be possible to compute the absolute time from this relative time. This constraint is satisfied by adding to the application at least one task with a shorter period.

3) Trace recorder: The execution of a tracepoint instruction calls the trace recorder that is in charge of capturing the current timestamp and store the event in a tracing buffer. The main requirement is to implement this critical code with a reduced overhead, trying to modify as little as possible the behaviour of the monitored application.

A specific test was defined to measure the overhead introduced in the system due to the execution of the trace recorder. The test consists in introducing a series of tracepoints followed one after another from the same thread. The time difference between two of these consecutive traces, if the thread has not been preempted, will be used to compute the overhead introduced.

According to the tests executed on the Cortex-R5 of the Zynq-US+ at a frequency of 499.995 MHz, the overhead introduced in the system was observed. The distribution of the 3749 samples is represented in figure 3, where 91.33% of the samples obtained introduce an overhead below 440 ns. Also, only a 0.053% of the samples is above 460 ns, with a worst observed case of 550 ns.

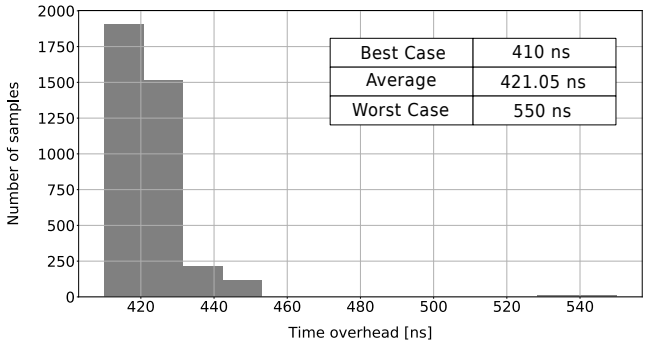

Fig. 3. Tracepoint overhead histogram

4) Context-Switch tracing: The developer is responsible for inserting the tracepoints at the beginning and at the end of the execution of each activity, interrupt handler or use of shared resources so that it can be monitored. The subsequent analysis of these traces will be the one that tries to reconstruct the behaviour of the system. But there are other points that must be monitored to know the real behaviour and usually are not at developer level, such as the context switch where the scheduler changes the executed thread. The proposed solution is to use the RTEMS User Extensions [10], an API provided by RTEMS that allows to extend the code of certain parts of the operating system, such as context changes. A callback is defined to be called every time a context switch occurs and in which a call to a tracepoint will be included, in order to register the threads involved.

5) Monitoring task: The monitoring task is in charge of managing the double-buffer in the trace-recorder and of sending the last one used to the monitoring service for its later analysis, using the protocols designed for that purpose. Its execution is not a part of the initial SUT, so it must modify as little as possible the behaviour of the system.

The size of the buffer should be defined depending on the requirements of the application to be monitored. In case of the example exposed in section VI, the maximum number of events per buffer is 1024, i.e. the trace recoder can save a maximum of 1024 traces before the monitoring task runs again and empties the buffer. For this particular instance, it means the application needs an extra memory allocation of 40kB (2 x 1024 x 20Bytes) for the trace storage.

B. TS and SUT communication

An MPSoC provides many advantages in this system, such as the use of shared memory in the inter-processor communications. Both processors involved in sending messages can use a reserved area in the shared memory. Also, in this solution, it is combined with the use of the Inter Processor Interrupt (IPI), as a signal to notify the receiver when the data is already available in the memory [14].

The shared memory has been configured to reserve the regions shown in figure 4. Each of them have been called by the type, i.e. Mailbox or Shared Area, and by the owner, i.e. RPU or APU. The owner of an area is the only one with write permissions, meanwhile, other processors have read-only permissions in order to avoid writing collisions.

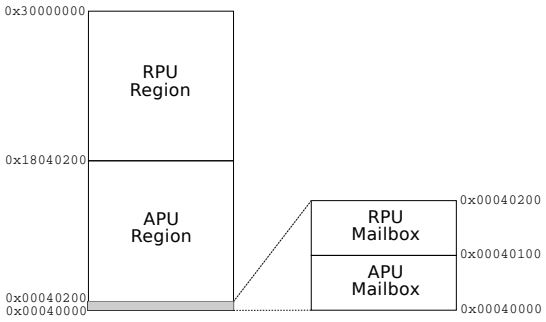

Fig. 4. Shared Memory Map

The proposed communication is a point-to-point protocol that allows to send short messages between two processors using the mailboxes previously defined. The transmitter writes within a mailbox a fixed structure message to start the communication and enables a signal in the IPI, which will cause an interruption in the receiver (ISR-RX). After that, the transmitter thread remains on a hold state, waiting for the acknowledgement. The receiver, in the interrupt handler, copies the mailbox content in a pending message queue and notifies, through another IPI signal (ISR-ACK), if the message has been added to the queue to be processed. When the transmitter receives the IPI signal, it finalises the waiting state and it is able to use the mailbox again for the next message if necessary.

1) Message structure: As mentioned above, the communication system uses the mailboxes to send the messages. These messages have a fixed structure with the fields defined in Figure 5. Source and destination addresses can be defined to route the message in a top layer of the communication stack. As well as a message identifier can be also defined, easing the receiver process the input message.

Param	Dest Addr	Src Addr	Data Format	Msg Id	Data Pointer	Data Size	CRC
Byte	0	1	2	3	4:7	8:11	12
Size	1	1	1	1	4	4	1

Fig. 5. Mailbox Message Structure

The Data Format field details what kind of data is attached to the message, and the Data Pointer and the Data Size fields define the location of this data. There are two format types:

- `message_fdata_void` Message without additional data
- `message_fdata_shmem` Additional data located in shared memory region. The pointer defines the data offset from the beginning of the shared memory region and the size, indicating the number of bytes allocated

The last field of the structure is the polynomial CRC with the purpose of validating the integrity of the message.

2) Memory allocation: A dynamic storage allocator manages the use of the shared memory between the two processors involved in the communication. For this purpose, the monitoring tool provides the TLSF allocator (Two Level Segregated Fit) [15], designed to meet real-time requirements, thanks to its bounded execution time.

The transmitter should allocate an area in its shared memory region using TLSF and write the data in this area. After that, it will send the mailbox to the receiver. The Data Pointer in the message must identify the data area to be processed and the receiver is in charge of notifying the source that it has finished reading the data. Logically, since the transmitter is the only one with writing permissions in this memory region, it must be in charge of releasing the allocated area, but never before the receiver has finished working with it. This approach allows large-size tranmissions, such as images or large arrays, with zero copy messages.

VI. Experiment and tool results

This section shows the results provided by the monitoring tool, testing the execution of an algorithm of Guidance, Navigation and Control (GNC). The observed application runs over a Zynq-US+ platform on the top of the RTEMS operating system and it has been designed using the a2k tool suite and the services that a2k provides.

The functional structure of the application is composed by three flows:

- **Data-Handling:** in charge of dispatching the messages and of managing the dynamic memory
- **Gnc:** the main algorithm in the application
- **Monitoring:** explained in Section IV-C and responsible of managing the monitoring traces

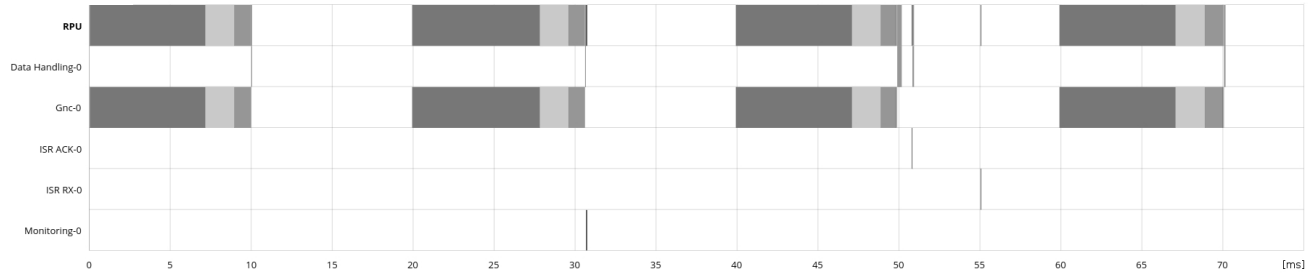

Fig. 6. Observed behaviour

The precedence relationship of their activities, i.e. the order in which they have to be executed, is shown in Figure 7.

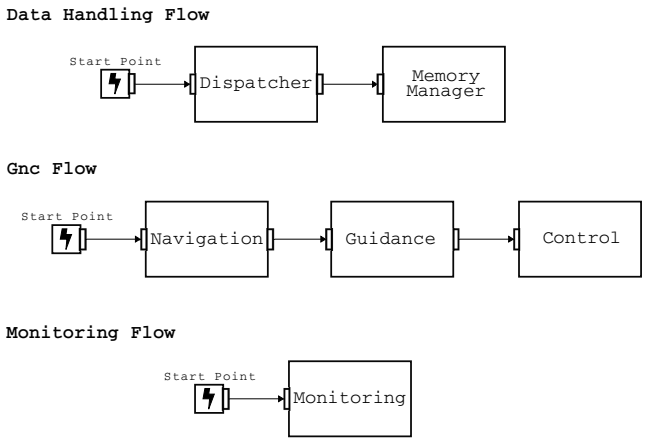

Fig. 7. Application Model Precedences

The execution period of the flows is defined in the application model, as a parameter of the description of the flows. The analysis services provided by a2k compute the assignment of the activities in the different threads and determine the execution priorities, in order to comply with the system constraints. The characteristics of the application are summarised in Table II.

TABLE II
APPLICATION MODEL - PERIODIC FLOWS

Flow	Period [ms]	Activities	Thread	Priority
Data Handling	20	Dispatcher	#0	2
		Memory Manager		
Gnc	20	Navigation	#1	1 (highest)
		Guidance		
		Control		
Monitoring	100	Monitoring	#2	3

Otherwise, the execution of the interrupt handlers are modelled using sporadic flows, because the periodicity of this type of flows cannot be defined. Instead, its Minimum Inter-Arrival Time (MIT) should be observed, the minimum time between the beginning of two executions of the same flow.

The implementation of the application follows the description made in section V. Also, the tracepoints have been added to the source code in order to store the temporal events, which enable the rebuilding of the temporal behaviour in the monitoring server.

The results provided by the monitoring tool are shown in a2k in two different formats: graphically and numerically.

1) Behaviour chart: A gantt-chart represents the execution of each activity categorised in processors (bold name) and in threads (bellow its processor). This graph helps the engineer to achieve a better understanding of the temporal behaviour. Figure 6 depicts the gantt-chart corresponding to the experiment used in this evaluation.

TABLE III
OBSERVED EXECUTION TIME

Flow	Activity	Execution Time [us]		
		Min	Avg	Max
Data Handler	Dispatcher	1.77	6.64	17.82
	Memory Manager	0.87	4.55	6.17
Gnc	Navigation	7061.85	7063.15	7064.66
	Guidance	1766.04	1766.05	1766.09
	Control	883.24	883.25	883.41
Monitoring	Monitoring	97.87	100.30	114.17

2) Observed execution time of activities: A statistics table describes the temporal behaviour bounds of the activities. Table III describes the minimum, maximum and time average of the execution time.

TABLE IV
INTERRUPT HANDLERS - OBSERVED EXECUTION TIME

Interrupt Handler	Execution Time [us]		
	Min	Avg	Max
ISR TX-ACK	6.41	6.83	7.96
ISR RX	1.96	2.24	2.85

3) Observed execution time of interrupt handlers: The observed behaviour of the sporadic activities, in this case the interrupt handlers, are defined in Table IV and Table V. Table IV depicts the execution time of the interrupt routines, whereas Table V shows the observed minimum inter-arrival time.

TABLE V
INTERRUPTS - OBSERVED MINIMUM INTER-ARRIVAL TIME

Interrupt Handler	Minimum Inter-Arrival Time [us]
ISR TX-ACK	99.96
ISR RX	99.95

All aforementioned charts and statistics help the engineer in the task of checking the compliance of the system with the temporal system requirements.

4) Monitoring overhead: The overhead introduced due to monitor the system in a worst-case scenario could be observed adding the communication overhead, the execution time of the monitoring task and the increment of time due to the tracepoints introduced.

In the above example, consider one monitoring period, around 84 traces are necessary every 100ms to monitor the system. The execution time of the dispatcher and the interrupt handlers involved in sending one tracing buffer have to be added to the equation and, of course, the monitoring task, in charge of managing the buffers.

TABLE VI
MONITORING OVERHEAD

	Worst Case [us]
Dispatcher task	17.82
Interrupt handler TX-ACK	7.96
Interrupt handler RX	2.85
Monitoring task	114.27
Tracepoints	0.550 (x84)
Overhead in 100ms	**189.10 us**
Overhead percentage	**0.19 %**

VII. CONCLUSIONS

In this paper we propose a runtime monitoring tool that extends the a2k tool suite. This new tool can be used to measure and to visualise the temporal characteristics of a real-time application. The engineer can analyse this observed behaviour and compare it with the expected one, helping in the validation process of the system. Also, it is useful to understand the behaviour of the system and to improve its theoretical model.

The selected approach to monitor the real-time application is a software tracing method that consists of adding tracepoints to the code and of recording them during the execution, adding a minimised overhead. A monitoring task is embedded in the application and provides the traces to the monitoring service. This service is in charge of analysing the traces, in order to be able to rebuild the observed behaviour, and of providing the results to the a2k viewers.

The a2k monitoring tool has been tested in a heterogeneous MPSoC using and RTEMS based system. The statistics of the temporal behaviour of a GNC application has been depicted in the same way that they would be provided by a2k.

Future work for this study focuses on improving the runtime monitoring tool in different ways. First of all, improving the RTEMS implementation, including tracepoints in the release time of the flows, the instant which the flow becomes ready to be executed, to provide the observed response time of the activities. And also, implementing the monitoring recorder in other real-time operating systems, different from RTEMS.

ACKNOWLEDGEMENTS

The work presented in this paper has received funding from the ECSEL Joint Undertaking under grant agreement No 737475 (AQUAS project). This Joint Undertaking receives support from the European Unions Horizon 2020 research and innovation programme and Spain, France, United Kingdom, Austria, Italy, Czech Republic, Germany.

REFERENCES

[1] K. M. Sacha, "Measuring the real-time operating system performance," in *Real-Time Systems, 1995. Proceedings., Seventh Euromicro Workshop on*, pp. 34–40, IEEE, 1995.

[2] A. Terrasa and G. Bernat, "Extracting temporal properties from real-time systems by automatic tracing analysis," in *Real-Time and Embedded Computing Systems and Applications*, pp. 466–485, Springer, 2004.

[3] M. Asberg, T. Nolte, and S. Kato, "A loadable task execution recorder for hierarchical scheduling in linux," in *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2011 IEEE 17th International Conference on*, vol. 1, pp. 380–387, IEEE, 2011.

[4] M. Desnoyers and M. R. Dagenais, "The lttng tracer: A low impact performance and behavior monitor for gnu/linux," in *OLS (Ottawa Linux Symposium)*, vol. 2006, pp. 209–224, Citeseer, 2006.

[5] F. Rajotte and M. R. Dagenais, "Real-time linux analysis using low-impact tracer," *Advances in Computer Engineering*, vol. 2014, 2014.

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

[7] M. Holenderski, M. Van Den Heuvel, R. J. Bril, and J. J. Lukkien, "Grasp: Tracing, visualizing and measuring the behavior of real-time systems," in *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, pp. 37–42, 2010.

[8] S. Girbal, J. Le Rhun, and H. Saoud, "Metrics: a measurement environment for multi-core time critical systems," *Embedded Real Time Software and Systems, ERTS*, vol. 18, 2018.

[9] J. Kraft, A. Wall, and H. Kienle, "Trace recording for embedded systems: Lessons learned from five industrial projects," in *International Conference on Runtime Verification*, pp. 315–329, Springer, 2010.

[10] RTEMS Project, *RTEMS User Manual - Release 5.ec95748*, 2019.

[11] F. Nicodemos, O. Saotome, and G. Lima, "Rtems core analysis for space applications," in *2013 III Brazilian Symposium on Computing Systems Engineering (SBESC)*, pp. 125–130, IEEE, 2013.

[12] A. Sailer, M. Deubzer, G. Lüttgen, and J. Mottok, "Coretana: A trace analyzer for reverse engineering real-time software," in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 1, pp. 657–660, IEEE, 2016.

[13] M. Côté and M. R. Dagenais, "Problem detection in real-time systems by trace analysis," *Advances in Computer Engineering*, vol. 2016, 2016.

[14] A. Pérez, A. Otero, and E. de la Torre, "Performance analysis of see mitigation techniques on zynq ultrascale+ hardened processing fabrics," in *2018 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pp. 51–58, IEEE, 2018.

[15] M. Masmano, I. Ripoll, P. Balbastre, and A. Crespo, "A constant-time dynamic storage allocator for real-time systems," *Real-Time Systems*, vol. 40, no. 2, pp. 149–179, 2008.