Journal of Intelligent & Robotic Systems (2021) 101: 37
https://doi.org/10.1007/s10846-020-01301-1

Real-Time Performance Evaluation for Robotics

An Approach using the Robotstone Benchmark

Matheus Leitzke Pinto[1] · **Marco Aurélio Wehrmeister**[2] · **André Schneider de Oliveira**[2]

Received: 31 March 2020 / Accepted: 14 December 2020 / Published online: 2 February 2021

Abstract

This paper discusses a novel approach to evaluate the real-time performance of computing platforms embedded in robotic systems. The motivation behind this work is the need for the robotic systems to meet their timing constraints, thus requiring time predictable real-time systems. We proposed a benchmark, named the Robotstone, which is an adaptation of a traditional real-time performance benchmark widely known in the real-time systems community. Thus, the presented work makes it easier for a robotic engineer to apply the benchmark in the modern robotics context by filtering out issues that do not matter on the robotics applications and adapting the benchmark's relevant portions. The Robotstone has a set of experiments related to time-constrained application scenarios usually found in robotic systems. Each experiment defines an application-specific parameter that increases at every iteration until the system overloads. The real-time performance is then evaluated through the breakdown point, i.e., the system configuration in the application scenario when any functional or timing constraint is not met. The proposed toolset has been evaluated on two distinct platforms representing a class of embedded computing systems usually employed in robotic systems. Obtained results demonstrate the applicability of the Robotstone benchmark for a quick assessment of computing systems' real-time performance often required on initial development stages of a robotic system.

Keywords Real-Time performance · Real-Time systems · Robotstone · Robotics · Benchmark

1 Introduction

Real-time systems (RTS) are computing systems used in applications that impose timing constraints that are often strict [36, 56]. In the software domain, RTSes are comprised of a set of tasks that perform application-specific activities.

Task execution must be *time-predictable*, i.e., the task execution time must be known at the design stage; this includes not only the execution of the algorithm, but the execution time of the RTS application programming interfaces (APIs) and libraries used in the implementation. The execution time is an important parameter for ensuring that the system tasks finish processing before their *deadline*, which are derived from the application time constraints. Another necessary software component is the *real-time scheduler*, which is responsible for controlling access to the processing units, and also for sharing processing time among multiple periodic and aperiodic tasks that run concurrently. Operations performed by the scheduler must also be time-predictable, e.g., evaluating the *scheduling policy* and releasing the tasks at the exact moment that each task is ready to be executed. The *real-time operating system* (RTOS) provides the services required by the RTS, which include not only real-time task-related services, i.e., task primitives and the scheduler, but real-time inter-process communication (IPC) and synchronization services (e.g., communication queues, mutexes, and others) as well.

Commercial off-the-shelf (COTS) hardware and other state-of-the-art computing platforms introduce sources of *indeterminism* that impact the task execution times [7], e.g., cache memories, pipelines, bus arbitration schema, and many other features of superscalar architectures. Because of such hardware complexity, in practice, some stress testing is required on the hardware platform to assess the timing predictability of the software. These tests should be used to obtain the upper limits of task code execution times

✉ Matheus Leitzke Pinto
matheus.pinto@ifsc.edu.br

[1] Departamento Acadêmico de Eletrônica, Instituto Federal de Santa Catarina, Florianópolis, 88020-300, Brazil

[2] Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial, Universidade Tecnológica Federal do Paraná, Curitiba, 80230-901, Brazil

and RTOS services. Moreover, in order to meet the time-to-market constraints, a quick evaluation of the *real-time performance* of the hardware/software platform can be done when the system target platform is chosen. The real-time performance can be evaluated using several metrics, such as the number of concurrent tasks the target platform can support, the system workload (i.e., the amount of code or information processed within the time interval), the task frequency, the maximum message length to be transported, etc. The performance is affected by factors such as scheduling overheads, priority inversions, inter-task dependencies and interference, timing anomalies, and domino effects [7, 36]. Thus, if two platform configurations have similar real-time performance, the engineer can choose the one that presents the best trade-off between cost and performance.

One application domain that requires the use of RTS is robotics, because a robotic system usually interacts with the physical environment. An example of a robot that has hard real-time constraints is the autonomous car [29, 58]: if the sensor input data is not processed within an usually strict timeframe, the system fails to meet its deadlines, which may lead to catastrophic consequences such as an accident with loss of human life. Therefore, it is necessary to choose appropriate hardware and software tools to develop a computational platform configuration that processes information under real-time constraints. Moreover, robotics engineers (as well as other computing systems engineers) usually face challenges when creating a robot that presents timing constraints, mainly due to a lack of knowledge about the meaning of "real-time concepts" [56]. According to the current literature, it seems that for a great portion of the robotics community, "real-time" means the processing of information while the robot is operating, which gives to the operator a sense of instantaneous response (e.g., [5, 6, 25, 33, 51, 60, 64]). A somewhat dated discussion [56] has already highlighted this common misunderstanding. The correct term for this sort of robot operation is "*online processing*". Another misconception for many robotics engineers relates to end-to-end response latency: the faster the response, the greater "real-time" compliance the system is considered to have (e.g., [6, 8, 27, 42, 44]). In this sense, performance in a robotic system is generally related to the average response time. However, real-time analysis is about predicting the exact time, or at least the upper-bound limit, at which operations are performed so task deadlines are met [17, 36, 40].

In this study, we consider that robotic systems are structured in a scheme such as the diagram presented in Fig. 1. The computing system serves to control robot operations by means of its *perceptual* and *actuator subsystems*. This computing system may be composed of one or many devices, where each is responsible for a specific type of processing. These devices may be connected by simple communication interfaces – such as an I^{2}C or SPI interface – or by means of network infrastructure to form a distributed system. A device may have a single processing core, or multiple cores (multicore). All of these characteristics are determined by the complexity of the application.

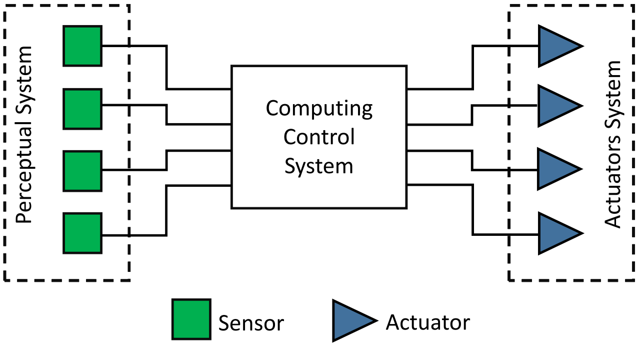


Fig. 1 A generic robotic system structure diagram

The software that runs on the computing control system, i.e., the robotic application software, also varies in complexity according to the hardware or the level of abstraction of the control task. As one can see from several works [11, 15, 43, 44, 49], the tools usually adopted for the development of this application provide abstractions in which the system is seen, even if indirectly, as a set of *nodes* that communicate by some specific mechanism. A node can represent a device, a core, or even an operating system (OS) process, depending on the abstraction being used. The nodes generally use the *publisher/subscriber* paradigm [16, 59] to communicate with each other. Nodes called *publishers* send messages through abstractions called *topics*. To receive messages from a topic, the node must be *subscribed* to it. The Robot Operating System (ROS) framework [49] adopts the concept of nodes and is widely used by the community. It provides, among several software tools, a middleware for communication between the different modules that compose a robotic system. In the ROS, nodes are created as OS processes and communicate primarily through the TCP protocol managed by the tool. If the robot's control system is just a device, then some nodes will be processes that will compute the internal data, while others will be responsible for communicating with the software device drivers to control the robot's peripherals. In case the system is distributed, each computational device will see the other through one or more processes/nodes. These nodes will receive/send messages through the topics and will interface between devices.

The ROS, like various middleware for robotics, was not initially created for real-time computing [22, 41]. This is due to the fact that it does not provide the necessary services, such as task creation, scheduling, and QoS management. In addition, because it makes extensive use of indeterministic libraries of the C++ language (e.g., dynamic memory allocation), it is not possible to integrate its services within real-time software tools and libraries, such as an RTOS. A discussion related to this topic can be explored in [23, 46]. In this case, the ROS developer usually mitigates indeterminism by increasing the operating frequencies of the nodes, so that the software can meet the application requirements. However, as already covered in this chapter, this does not guarantee that all imposed deadlines will be met.

Real-time software development, including that of robot software, comprises several stages of a process that is often iterative [9, 52, 61], e.g., requirements engineering, analysis, design, implementation, verification, validation, and several other activities. In this study, we focus on the evaluation of the real-time aspects of a robot's software system under development. Such a workflow is presented in Fig. 2. Firstly, the potential target computing platforms need to be evaluated regarding their real-time capabilities, i.e., the real-time services and predictable execution times for the task set that implements the robot software controller. Characteristics such as real-time performance, cost, and the complexity of the system must be considered when the execution platform is chosen. The second step is the development of the real-time tasks that implement specific robot services and algorithms. The tasks are generally developed using some task model imposed by the selected scheduler. The next step is to specify the task execution time. As mentioned, the whole system usually presents many sources of indeterminism that impact the task execution time. Thus, execution time is often obtained by running the task set on the target platform and measuring the upper-bound value that represents the *worst-case execution time* (WCET) [1, 63]. After determining the measured WCET of the tasks, it is possible to perform a scheduling test to verify the task set schedulability in the design phase, i.e., the verification that the timing constraints of all tasks can be met [17, 36, 40]. However, it is essential to highlight that, although this is a common practice, measuring the upper-bound does not provide any guarantee that the correct WCET of a task has been determined and, hence, used in the scheduling test [24, 28]. Finally, all tasks must be run concurrently, to check if any deadline will be missed in the robot operation. For a safety-critical robot, some kind of fault-tolerance system must be integrated to maintain the correct operation of the application, even in the presence of timing violations and the indeterminism of the execution platform [7].

We consider that there is a lack of well-established methods or tools to determine the computing controller platform, based on its real-time capabilities, for a robot (corresponding to the first step in Fig. 2). We also consider the possibility to create a solution to the problems mentioned, based on the knowledge in the RTS community. In that manner, this work proposes a novel approach to evaluating computing platforms for a robotic systems controller in terms of real-time performance. The *application-oriented benchmark Robotstone* was developed for this reason. It provides a set of experiments that represent possible robotic applications, with the goal of capturing specific parameters and evaluating system real-time capabilities without considering the details of the underlying software and hardware. The experiments comprise *synthetic applications* that do not perform the useful work of working robots (there are no perceptual or actuator systems). However, these applications overload the computing platform like a real application by using *synthetic code* and *synthetic messages*. The Robotstone is an adaptation of the original *Hartstone* benchmark [32, 62], which proposes a synthetic code that, we believe, represents the processing patterns of robotic applications. Robotstone also adapts some procedures in the Hartstone model by considering the publisher/subscriber communication paradigm in experiments related to message exchange, becoming more compatible with the evaluation of robotic applications.

The proposed Robotstone delimits the *breakdown point* of the system as its configuration (number of tasks, frequencies, etc.) when one or more deadlines were not met. This result gives the figure of merit for how well the system achieves real-time capabilities in an application scenario – that is, the real-time performance evaluation based on the Robotstone benchmark. Seven experiments are proposed to evaluate the real-time performance of a system in different application scenarios. Unlike Hartstone, it is easy to change the task parameters and perform tests with a configuration closer to the expected load of a real robotic system being designed. The benchmark also provides *fined-grained measurements* of some task parameters so that the developer can see the more in-depth system behavior. To verify the applicability of the benchmark, two embedded platforms were analyzed as case studies. We conducted a case study on systems with RTOS without using a know real-time robotic middleware. The reason for this is that we believe

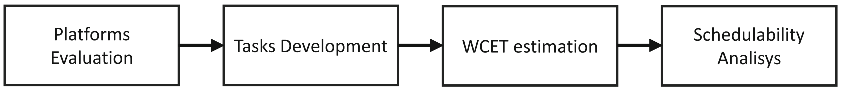


Fig. 2 A workflow for RTS development

that the RTOS will return better results for system evaluation. That's because there are fewer software layers to insert unpredictability into the application. Furthermore, RTOSes already have a well-defined interface and are well known by the real-time developer community. In any case, we advise that in future studies, real-time robotic middleware [3, 12, 13, 44] to be tested. The obtained results showed that the Robotstone was suitable for evaluating computing platforms for robotics, as the resources expected for RTS were verified on both platforms, corroborating the suitability of the benchmark for real-time robotic applications. This paper is organized as follows. Section 2 presents related works; Section 3 describes the Robotstone benchmark; Section 4 discusses the embedded system platforms used in the experiments as well as the justification of their choice for real-time performance evaluation of a robotic application; Section 5 analyzes the results of executing the Robotstone benchmark on the selected embedded platforms; finally, Section 6 draws the conclusions and points to future research directions.

2 Related Works

As mentioned, this study intends to provide a means to evaluate the real-time performance of the computing platforms, i.e., hardware and software, that are used to implement robotic systems. One can see that, although some works in the literature discuss the evaluation of robotic systems performance, very few works are concerned with the real-time performance of the employed computing platform.

Related to robotic middleware survey, we analyze two works about the subject [22, 41]. The authors on both works discuss and provide a similar qualitative assessment of the middleware currently available for robotic applications, including real-time capabilities. They assess real-time capability in terms of a binary "yes/no", which indicates the presence of real-time modules. According to [22] and [41], the evaluated qualitative attributes are relevant to robotic software development. One can cite the following examples of robotic middleware that propose real-time capabilities: Orocos [12, 13], RT-middleware [3], and XBotCore [44]. However, relying on a "yes/no" assessment, as presented in [22] and [41], is not sufficient for a well-supported decision-making process in robotic systems development.

In [42], the publisher/subscriber system implemented in the ROS [49] framework was evaluated in terms of the end-to-end latency of the message sending mechanism. Various combinations of ROS1 and ROS2 have been tested in order to measure the latency and its determinism for data exchange between ROS nodes. Another work [27] proposed a robotic software framework named aRDx. According to those authors, aRDx provides a communication layer that is able to exchange data under real-time constraints. The real-time performance of aRDx was evaluated and compared with other software frameworks, namely, ROS [49], Orocos [13], YARP [43] and aRD [15]. The evaluation considered the round-trip time and communication bandwidth for data exchange between nodes within distinct distribution domains. The main goal was to assess the scalability of the frameworks in terms of the number of communicating nodes and how this impacts the predictability of the data exchange. Although both works employed benchmarks, schedulability analysis was not performed in order to assure that the communicating nodes did not miss any deadline for sending or receiving the messages. Also, there was no evaluation of whether or not the task set was actually schedulable within the context of each computing unit.

Many works published in the last few years intended to evaluate the predictability of RTSes, running on modern hardware platforms, that aim to improve average performance, i.e., predictability is usually sacrificed in favor of better average performance [63]. In [8], a set of performance metrics were analyzed for a control system of a toroidal device used for studies on thermonuclear fusion. In this kind of application, it is quite important to know the time predictability of the stimuli/responses of the controllers. Various configurations of a software stack were evaluated, including the performance of some services provided by distinct RTOSes, namely, VxWorks, Xenomai, RTAI for Linux, and regular Linux. The I/O signals were measured with an oscilloscope, and thus, the latency and jitter of those signals were determined. In addition, in [4], the use of RTOSes in low-footprint microcontrollers (MCUs), i.e., an MCU constrained to a maximum of 4 KB of RAM and 128 KB of ROM, was studied. The goal was to evaluate how predictable the RTOS services running in such computing systems were. According to the authors, two approaches were possible: application-based benchmarks or benchmarks that evaluate the most frequently used RTOS services. However, the authors claimed that the former approach was not suitable for evaluating RTOS behavior, but was suitable for the application behavior. Distinct benchmarks were discussed and their metrics were evaluated in terms of how well they could measure RTOS features, including the time predictability of the service execution. Although these works present contributions relevant to the evaluation of the real-time capabilities of a computing system, the proposed methods are based of empirical measurements, i.e., they lack the more formal and analytical approach based on the body of knowledge of the real-time community.

On the other hand, model-based approaches have also been employed to evaluate the time predictability of an RTS. A tool named Oris is presented in [14]. Oris is based on both preemptive timed Petri nets and stochastic

Petri nets, in order to enable to qualitative verification and quantitative evaluation of reactive systems. The evaluation was done on a set of system parameters informed by the user, e.g., execution time, activation period, and others. Although the Oris approach was based on formal analysis, the critical information of the system tasks was informed manually by the user. In other words, the user needs to execute the tasks on real hardware to discover the execution time, which, in turn, may not represent the real WCET of the tasks. WCET could be obtained by using a detailed model of the CPU, which is unlikely to be achieved for modern superscalar processors [63]. Furthermore, the work presented in [47] advocates the importance of determining real-time parameters in earlier design stages. Those authors define a set of benchmarks to evaluate the performance of distinct system-level analysis methods employed to assess distributed RTSes. The impact of various abstractions is discussed in terms of the pitfalls and the reasons for pessimistic performance predictions. The results showed that the choice of an appropriate analysis abstraction matters, and that the analysis accuracy of different approaches depends highly on the particular system characteristic. In other words, providing precise models of the target computing platform and also the system parameters are important issues for an accurate estimation in system-level performance analysis.

Robotics engineers must support their designs with quantitative data in order to choose the most suitable middleware and computing platform for implementing the various parts of a robotic system. Therefore, an evaluation of real-time performance must be based on the analyses already used in the design of the RTSes [17, 36, 40]. Such an analysis should use benchmarks instead of using a quantitative method based on empirical intuition and the average duration of task execution time, as is commonly found in the literature [4, 8, 27, 42, 44]. Moreover, studies that employ benchmarks [27, 42] must not neglect the fact that the underlying computing system impacts the predictability of software task execution (i.e., the schedulability of the task set), and hence, the latency and jitter of the data exchange [63].

This work aims to fill the analysis gap not addressed by the literature by proposing a reliable benchmark. The evaluation is performed not only by means of task set schedulability on robotic systems, but also by the real-time performance of the communication system based on the publisher/subscriber paradigm. This work discusses the Robotstone benchmark that was briefly introduced in [48] with the name *Hart-ROS*. We choose to change the name because Hart-ROS – “Hart” derived from Hartstone, and “ROS” from the Robot Operating System framework [49] – seems to be specific to ROS middleware, and not to the robotic systems in general. Besides that, some significant changes were performed since Hart-ROS. An assessment based on new experiments and improvements in mathematical notation was introduced.

Differently from [48], this study presents an in-depth description of the Robotstone. The Robotstone follows the guidelines of the Hartstone benchmark model [32, 62]. For investigations related to the processing domain (PD), we use the Hartstone PN (*periodic tasks, nonharmonic frequencies*) series, because they use a more straightforward model than other proposed series, in addition to providing the necessary information about the real-time performance of a system. For the communication domain evaluation [32], the experiments on the Robotstone were adapted from Hartstone to evaluate and support the architectures of robotic system, from simple RTOS software to complex real-time middleware. Finally, this work presents the implementation of the benchmark, rather than simply presenting the model as it was presented in Hartstone.

3 The Robotstone Benchmark

The Robotstone is an application-oriented benchmark [32, 62] for robotics computing controllers, where real-time performance is evaluated using a synthetic application running on the target system. A synthetic application does not perform any practical work in an operating robot. However, it applies the same workload to a system as a similar real application, so that it is possible to capture observations from the synthetic system. It uses a set of experiments, each one corresponding to a representative robotic application, with the goal of exploring a specific system capability. Synthetic applications are composed of real-time tasks, called *synthetic tasks*. A generic task τ_i, where i is a unique identifier is modeled as follows. It runs recurrently during system operation in instances called *jobs* $\tau_{i,k}$, $\{k\} = \{1, 2, 3, ...\}$. As observed from the Gantt chart of Fig. 3, a task is ready to run a job in an *activation time* a_k. After activation, the execution of task code will begin at *start time* st_k. The time needed to run the entire code for the task is given by the *execution time* parameter, C_i (corresponding to its WCET). The moment a job completes the execution of its code is called its *completion time*,

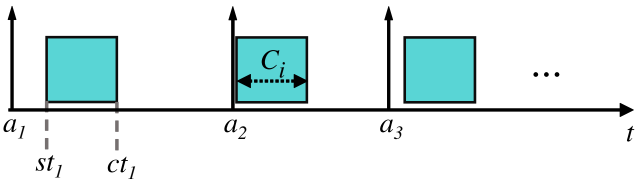


Fig. 3 A Gantt chart that represents the jobs execution from a task τ_i. Being $\{k\} = \{1, 2, 3, ...\}$, a_k indicates the activation times, st_k are the start times, ct_k are the completion times and C_i is the execution time

ct_k. The timing constraints of a task are met when its job completion time occurs before its *deadline*.

The evaluated systems are interpreted as *nodes* in a benchmark experiment. A node can be an OS process, thus a machine can be composed of many nodes. A node runs a set of real-time tasks whose timing constraints must be met so that the timing constraints of the system are met. Each node in the Robotstone schedules tasks using a *rate-monotonic* (RM) scheduling policy [39]. In an RM policy, task activation times occur during regular *activation periods* T_i. The task deadlines are always at the end of an activation period, so a job completion time must occur before the next activation. In this model, the shorter the activation period, the higher the task priority. An RM policy was chosen because it is simple to implement using *fixed-priority preemptive* (FPP) schedulers [35], which are usually present in commercial RTOS (e.g., FreeRTOS [2], Xenomai [26], VxWorks [31], μC/OS-II [31], QNX [37]). In that manner, the target system must provide an FPP scheduler so the RM policy can be applied in the benchmark. Furthermore, in most control applications, which include robotic systems in general, the sampling period is constant due to the models of digital control systems used [45]. These samples will be handled by tasks that will run periodically, justifying the use of RM.

Each benchmark experiment is performed in *test steps*. An application begins with *baseline tasks* that will increase its parameters during the testing iterations. The goal is to overload the system until it reaches the *breakdown point*. With this approach, a functional or temporal system limitation will occur. An example of a functional system limitation is the lack of memory for more task creation. A temporal limitation occurs when one or more deadlines will be missed by the synthetic tasks. Thus, real-time performance can be evaluated through the scenario obtained at the breakdown point. For example, if a system takes a bigger workload than another system, which has similar processing speed, in an experiment where a breakdown occurs, it will have a best real-time performance at first glance, because it can handle a greater load in real time. Also, if a system reaches the breakdown point in the first step of a basic scenario, it is not suitable for use as an RTS in this application scenario.

There are two classes of applications that the Robotstone applies to. The first one is related to PD experiments, which are basically the general case on any RTS or simple robotic system. The system capabilities that directly affect task schedulability are stressed by increasing the value of parameters such as activation frequency, workload, and number of tasks at each step. In this case, the synthetic tasks execute the *synthetic code* referred to as *Kilo Whetstone Instructions* (KWI) [19, 62], based on a set of instructions considered representative in scientific numerical calculation, which are weighted using floating point arithmetic. Some common operations on robots include inverse kinematics calculations, route planning, artificial intelligence algorithms, and point cloud filtering [21, 54, 55]. These operations use instructions similar to those that comprise the KWI, which makes the KWI a synthetic code representative of robotic applications.

For PD experiments, a *sufficient test* [39] can be used to verify that the tasks at the breakdown point satisfy what is expected in the RM scheduling. It is an analytical method to verify whether a set of tasks running concurrently in a single processing core can meet its timing constraints when the system is running. Equation 1 was used to apply this test. The variable U indicates *processor utilization*, which is a workload metric. If this value is equal to 1, the processor is occupied 100% of the time. However, the RM test requires that processor utilization for n tasks is always less than $n \cdot \left(2^{1/n} - 1\right)$, so the tasks will meet their deadlines. Otherwise, the timing constraints of the tasks cannot be verified. To calculate U, the execution time C_i and activation period T_i of each task τ_i, $i \in \{0, 1, 2, ..., n\}$, is considered. The factor C_i/T_i can be referred to as the *task utilization*, U_i. This parameter is a workload metric related to a specific task. In the remainder of this paper, both U and U_i will be expressed as percentage values.

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n \cdot \left(2^{1/n} - 1\right) \tag{1}$$

In an indeterministic system, U_i can vary for several reasons, but it varies mainly because of the variations in the execution time. However, this value is defined *a priori* in the benchmark by means of Eq. 2. A parameter present in the equation is ω_i, which expresses the task τ_i workload as well as the U_i parameter. However, the ω_i is measured in *KWI per activation period* (KWIPP) – the period is related to the task activation period. We will call this parameter as the task *synthetic workload*. The processor speed is determined by parameter ν, given in *KWI per second* (KWIPS) obtained by the Robotstone before an experiment.

$$U_i = \frac{\omega_i}{T_i \cdot \nu} \tag{2}$$

Equation 1 can be used to check if the task set should have met their deadlines based on tasks utilization. It is worth noting that although there is an *exact test* to verify schedulability with RM scheduling [30], it is more complex than the test presented in Eq. 1. Therefore, for simplicity and didactic reasons, we decided to use the latter in this work.

The second class of benchmark applications relates to *processing and communication domain* (PCD) experiments. Experiments that use this application have the goal of exploring the capability of the target system for managing both message transactions and task code execution while the system maintains deadlines. This kind of evaluation is more complete because it explore more complex application cases, i.e., when tasks trade messages and process internal information based on these. In addition to the code being synthetic, messages are also. These messages are composed of dummy bytes that stress the communication system by their length. In this case, in addition to the existence of baseline tasks, there are also *baseline messages*. Thus, the system capabilities are overloaded by increasing the value of parameters that affect the communication system and scheduler, e.g., message size, number of messages being transported, task workload, etc. These experiments use the node concept directly and consider the publisher/subscriber communication paradigm [50, 59]. As highlighted in Section 1, these concepts are used in robotic software tools. If the system natively has the publisher/subscriber mechanism, the benchmark test will use the API provided by the system. If the system does not have it, system services must be used to implement the paradigm.

In the remainder of this section, we present the developed experiments and the concepts behind them, and the software architecture of the Robotstone benchmark.

3.1 Definitions of the Proposed Experiments

The Robotstone is structured as a set of seven experiments: three are for the PD and four are for the PCD. The *system workload* or *scheduling load* gradually increases with the execution of the experiment at each step. As suggested in [20], a test step has a duration of 10 s. System workload increases when the amount of code executed by the task set increases within a time interval, or when the length of the messages increases. The scheduling load in the PD means the amount of tasks ready to run per time interval (either by the size of the task set and/or by its activation frequencies). In the PCD, the scheduling load is also affected by the number of messages being transported per period. Therefore, each experiment provides a figure of merit for how well the system can maintain its real-time constraints with the increase in system workload and scheduling load in a synthetic application that represents robotics applications.

Before executing PD- and PCD-related experiments, the benchmark must obtain the speed ν of the processor at the node. The speed is required by Eq. 2. For this, a synthetic task will run continuously for a predetermined period, without interruption. Then, ν will be obtained by the ratio of synthetic code executed to the execution period. The same value can be used in all experiments. After obtaining the processor speed, experiments related to either the PD or PCD can be performed.

In PD experiments, a node named *Master* runs the selected experiment, which has five baseline tasks $\{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5\}$. This relationship is presented in Fig. 4a. The task parameters can be adjusted by the user according to their project needs. However, here we present default parameters to evaluate the Robotstone capabilities. It uses the task frequencies proposed in [62] ($\{63, 30, 14, 10, 6\}$ Hertz), as well as the proposed initial processor utilization $U \approx 15\%$. This value is divided equally between tasks, that is, $U_i \approx 15\%/5$. With this value, the tasks will easily pass the RM sufficient test. The task workloads are obtained using Eq. 2.

According to [62], in each testing iteration, the experiment specific parameters are increased, so a processor utilization between 2% and 3% increases. Therefore, the PD experiments performed on a Master node are defined as:

- *Experiment 1* – At each step, the workloads of the synthetic tasks are increased by 10% of their initial value. This experiment stresses system workload by increasing task workload.
- *Experiment 2* – At each step, the activation frequency of each task increases by 10% of its baseline value. This experiment stresses the scheduling load by increasing task frequency.

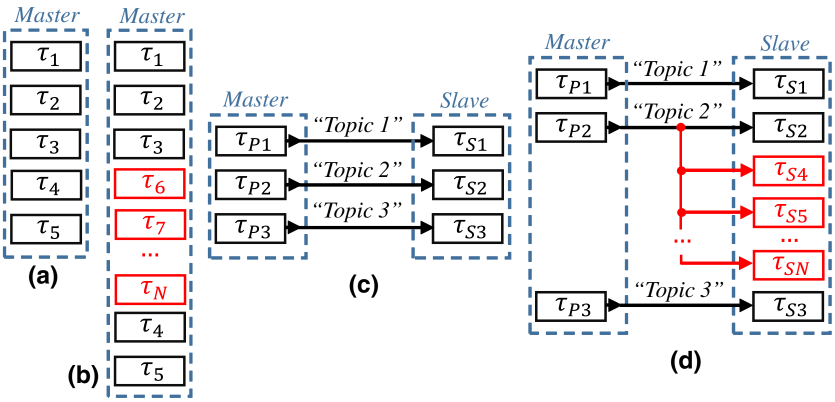


Fig. 4 Each diagram represents some experiment scenarios, being **a** and **b** corresponding to the PD and **c** and **d** to the PCD. Each τ_i, where i an unique identifier, represents a task. The dashed rectangles represent a node. Arrows represent a topic, with the arrowhead being the subscriber side

- *Experiment 3* – At each step, a task enters the system with the same parameters as τ_3 (Fig. 4b). This experiment stresses the system workload and scheduling load by increasing the number of running tasks.

The benchmark provides four experiments for the evaluation of PCD applications. For this, there must be one node that is considered the Master, and another node considered the *Slave*. On the Master side, the experiments begin with the baseline publisher tasks $\{\tau_{P1}, \tau_{P2}, \tau_{P3}\}$ that will send messages on individual topics. On the Slave side, the baseline subscriber tasks $\{\tau_{S1}, \tau_{S2}, \tau_{S3}\}$ wait for messages on their corresponding topics.

The relationship between tasks, topics, and nodes is shown in the diagram in Fig. 4c. The publisher tasks execute their workloads, and then send their messages. The subscriber tasks wait for the messages, and after receiving them, perform their workloads. As in PD experiments, task parameters can be defined by the user. However, here we present the default parameters defined in Robotstone. The size of a baseline message is defined as the word length of the target system architecture because it is considered the minimum load the system can handle, and it also represents the simpler messages on robotic applications. The frequencies of the messages were set as indicated in [32] ($\{7, 5, 3\}$ Hertz). As in the PD experiments, the baseline task workloads were set so processor utilization was approximately 15%, and was equally divided between tasks ($U_i \approx 15\%/3$ at each node). Therefore, the PCD-related experiments performed on the Master and Slave nodes were defined as:

- *Experiment 4* – As in the PD experiments, at each step, the synthetic task workloads are increased by 10% of their initial value. This experiment stresses the processing system workload by increasing the task workloads while maintaining the message size. Thus, it serves to evaluate how the system maintains deadlines by increasing the processing of information.
- *Experiment 5* – The messages length is increased by a power of two at each step. This experiment stresses the communication system workload by increasing the messages size. Because there are a large number of robotic applications, a variety of message types with varying sizes are also present, thus justifying an experiment that verifies the impact of different sizes.
- *Experiment 6* – As in the PD experiments, at each step, the task frequencies are increased (and hence the number of messages sent) by 10% of the baseline values. This experiment stresses, mainly, the scheduling load in the PCD by increasing the frequency of messages sent by the tasks. With a justification similar to Experiment 5, various robotics applications require different frequencies. For example, sensors can send samples in ranges of Hz to kHz.
- *Experiment 7* – At each step, the number of tasks with the same parameters as τ_{S2} is increased, subscribing to the same topic (Fig. 4d). This experiment stresses both system workload and scheduling load in the PCD by increasing the number of subscribers and the number of messages sent to them in the same topic. This is a common behavior in robotics when multiple nodes/tasks read a value published by the same sensor.

As an extra feature, the benchmark also provides the *fine-grained measurement* of tasks *response times* and *response jitters* for a more in-depth system view. A response time $R_{i,k}$ of a task job $\tau_{i,k}$ is the difference between its activation time and its completion time, as presented in the Gantt chart in Fig. 5. In that figure, the white gaps represent the *interference* caused by other tasks and system overheads. Consider that during a test step of the experiment, m instances $\tau_{i,k}$ are executed with the respective response times $R_{i,k}$, $\{k\} = \{1, 2, 3, ..., m\}$. One of the parameters obtained is the *worst-case response time* (WCRT) R_i corresponding to the longest response time achieved by a task τ_i (see Eq. 3). If any deadline is missed in a test, $R_i > T_i$ will occur. Another parameter obtained is the *average-case response time* (ACRT) R_i^{avg}, given by Eq. 4. Finally, the response jitter J_i^R represents the response time variation. It was calculated using Eq. 5.

$$R_i = max\left(R_{i,1}, ..., R_{i,m}\right) \tag{3}$$

$$R_i^{avg} = \frac{1}{m}\sum_{k=1}^{m} R_{i,k} \tag{4}$$

$$J_i^R = \frac{1}{m}\sum_{k=2}^{m} |R_{i,k} - R_{i,k-1}| \tag{5}$$

3.2 Software Architecture

The Robotstone benchmark was developed using API layers as presented in Fig. 6, where each layer uses the lower-layer services. The highest software layer is the *Robotstone Application* layer, which implements the synthetic applications that will run as experiments. The applications require basic real-time services, such as task management and communication through the publisher/subscriber paradigm, available through the *Real-Time API*. That layer is basically a wrapper API for the target system services, indicated by *System API*. The creation of the Real-Time API is justified by making the benchmark more portable for different architectures. Both upper layers were made in C++ language, to be portable to UNIX-like systems and MCUs.[1] The programming

[1] The projects using the Robotstone, can be downloaded in https://github.com/MatheusPinto/robotstone

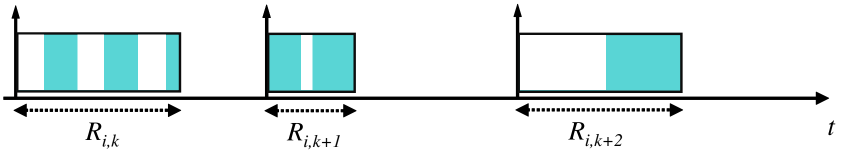


Fig. 5 A Gantt chart that represents the jobs response times $R_{i,k}$, $\{k\} = \{1, 2, 3, ...\}$, from a task τ_i. The response time begins in activation and finishes in the completion time. The white gaps on rectangles are caused by interference from other tasks and system overheads

effort required from the user to port Robotstone to a new architecture is to insert the System API services in the Real-Time API. These services can be implemented in pure C language since correct compiler directives are included. It is important to note that the specific implementation of Real-Time API layer services depends on System API. Thus, although Real-Time API provides, for example, publisher/subscriber communication services, it will be real-time or not dependent on the implementation of the system. This implementation could be provided by an RTOS, as well as real-time middleware services.

At each Master and Slave node, a specific task is used to manage the experiments: ${}^{M}\tau_{man}$ in the Master and ${}^{S}\tau_{man}$ in the Slave. The synthetic tasks run the application and are controlled by the management tasks that have the highest priorities. In the PCD experiments, ${}^{M}\tau_{man}$ and ${}^{S}\tau_{man}$ must communicate with each other within specific topics, as observed in Fig. 7. Synthetic tasks communicate between topics as well, but this is not presented in the figure.

Figure 8 presents a simplified Unified Modeling Language (UML) class diagram of the benchmark. The package indicated as “Robotstone Application” means the classes related to the Robotstone Application layer from Fig. 6. The classes in the same C++ *namespace* `RealTime` presented in Fig. 8 are related to the Real-Time API layer. Modules with the “wrapper” stereotype indicate that it is necessary the inclusion of some System API layer services to operate. The Master node will have an instance from `RobotMaster` class, and the Slave will have an instance from `RobotSlave` class. These two classes inherit from the more generic class called `Robotstone`. The `Task` class has basic methods for creation, priority attribution, destruction, and other task operations. `Signal` objects are used to notify about events between tasks in a same node.

Two subsystems were created in the class diagram to represent internal implementations of the Real-Time API layer. The “RealTime Subsystem” is related to the generic RTS services and configurations (e.g., scheduler starting). Note in Fig. 8 that the Robotstone Application uses these services. The “Communication Subsystem” is part of the `Communic` namespace. Both are related to classes and services for real-time communication between tasks using the publisher/subscriber mechanism. One of the services provided by “Communication Subsystem” is the action of inquiring about publishing or subscribing to a topic. As can be observed, each topic is related to an object, and can have zero or more `Publisher` and `Subscriber` instances. The use of the `Topic` class is not required if a system has the publisher/subscriber paradigm natively. In this case, only the `Publisher` and `Subscriber` instances exist in the Robotstone Application. These two kinds of objects will have a relationship with the “Communication Subsystem” (e.g., by sharing variables or using standard services) that is represented by the line between the `Topic` object and the subsystem.

The `RobotTask` class, together with the `Robotstone` and its subclasses, constitute the Robotstone application layer in the model presented on Fig. 6. This class provides a more specific abstraction of the synthetic tasks instead of simply using the `Task` class. As can be seen in Fig. 9, a `RobotTask` object consists of a task (a `Task` object), a `Signal`, a `Publisher`, or a `Subscriber`. Note that `RobotTask` does not inherit from `Task`, but is composed of it. The `Signal` object aims to signal synthetic tasks to their first activations, so that they are

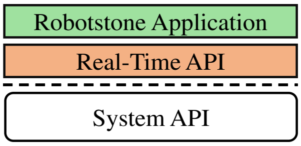


Fig. 6 The software API layers present in the Robotstone benchmark

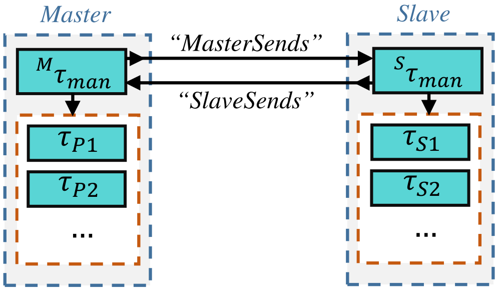


Fig. 7 A diagram that represents the interaction between the management tasks in the Master node (${}^{M}\tau_{man}$) and in the Slave node (${}^{S}\tau_{man}$). The arrows indicate communication topics. In the “MasterSends” topic, ${}^{M}\tau_{man}$ is the publisher and ${}^{S}\tau_{man}$ is the subscriber. In the “SlaveSends” topic, ${}^{S}\tau_{man}$ is the publisher and ${}^{M}\tau_{man}$ is the subscriber. Being $i \in \{1, 2, 3, ...\}$, τ_{Pi} indicate the publisher tasks and τ_{Si} indicate the subscriber tasks

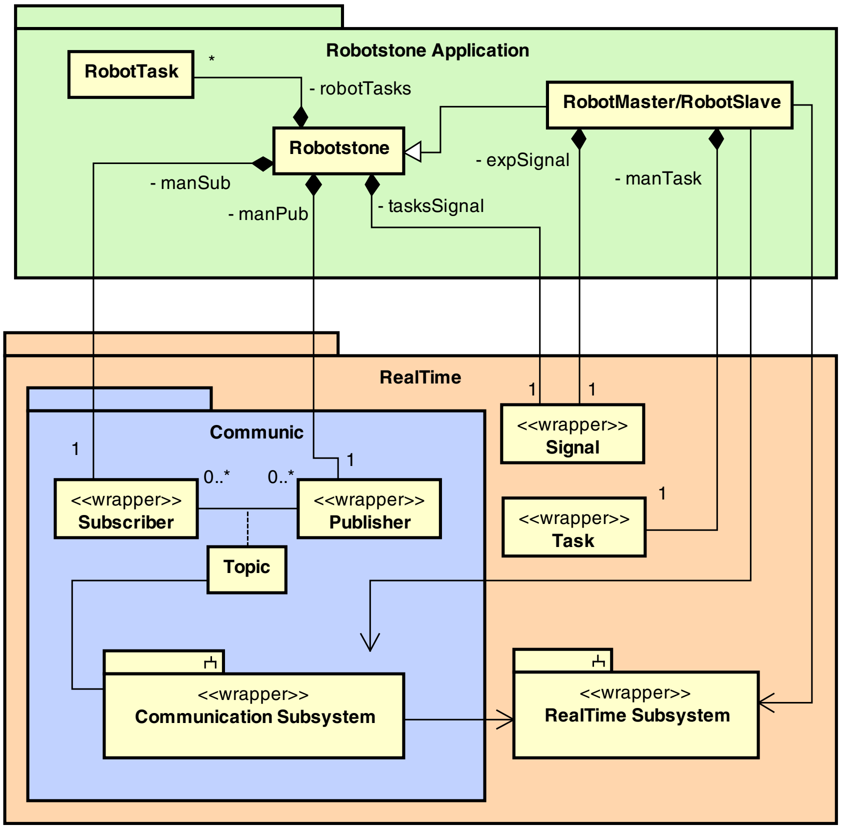


Fig. 8 A simplified UML class diagram of the Robotstone Application and Real-Time API layers. As `RobotMaster` and `RobotSlave` classes have similar interface, we use a same block to represent both

all activated at the same time. In PCD experiments, instances of `RobotTask` in a Master node will each have one `Publisher` for sending messages. However, in a Slave node, the `RobotTask` objects have an instance of `Subscriber` for receiving messages. In this class, the task codes are implemented for execution of the synthetic workload and message transactions.

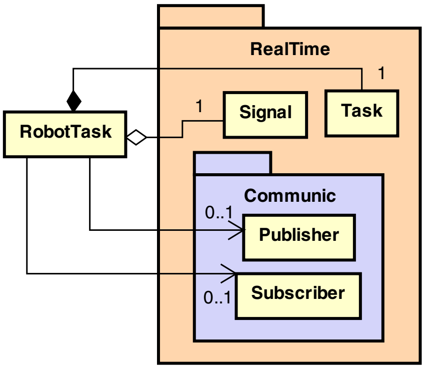


Fig. 9 A simplified UML class diagram related to `RobotTask` class. A `RobotTask` instance contains a `Task` object, for running the synthetic code, and a `Signal` object reference that indicates the first job activation. If a PCD-related experiment is chosen, then a `Publisher` or a `Subscriber` instance is created

In Fig. 10, the Robotstone application layer state machine diagram in UML is presented. The initial state is the experiment request by the user, made in the Master node. After the experiment is chosen, the first initialization parameters are configured. If a PD experiment is chosen by the user, then the next state will be “Start Uniprocessor Experiment.” In this case, the baseline tasks are initialized and synchronized with the use of the `Signal` object (Fig. 8), so that the initial activations of all tasks will occur at the same time. After synchronization, the ${}^{M}\tau_{man}$ enters a sleep state for the period of the test step. After the test step, a report is generated and sent to standard output from the system. If a breakdown occurs, the experiment is finished. Otherwise, the task parameters are updated according to the experiment and the tasks are run again in another test step.

If a PCD experiment is chosen, the system will enter the “Start Distributed Experiment” state. In this case, the Master initializes the experiment parameters and must wait for a handshake from the Slave node to start the experiment. The handshake is performed by the management tasks ${}^{M}\tau_{man}$ and ${}^{S}\tau_{man}$. After the tasks in both nodes are initialized and synchronized, the management tasks go to sleep and the experiment tasks are run. The cycle is similar to that in the PD: if a breakdown point is reached in any node, the experiment finishes; otherwise the tasks are updated and a new test step is performed.

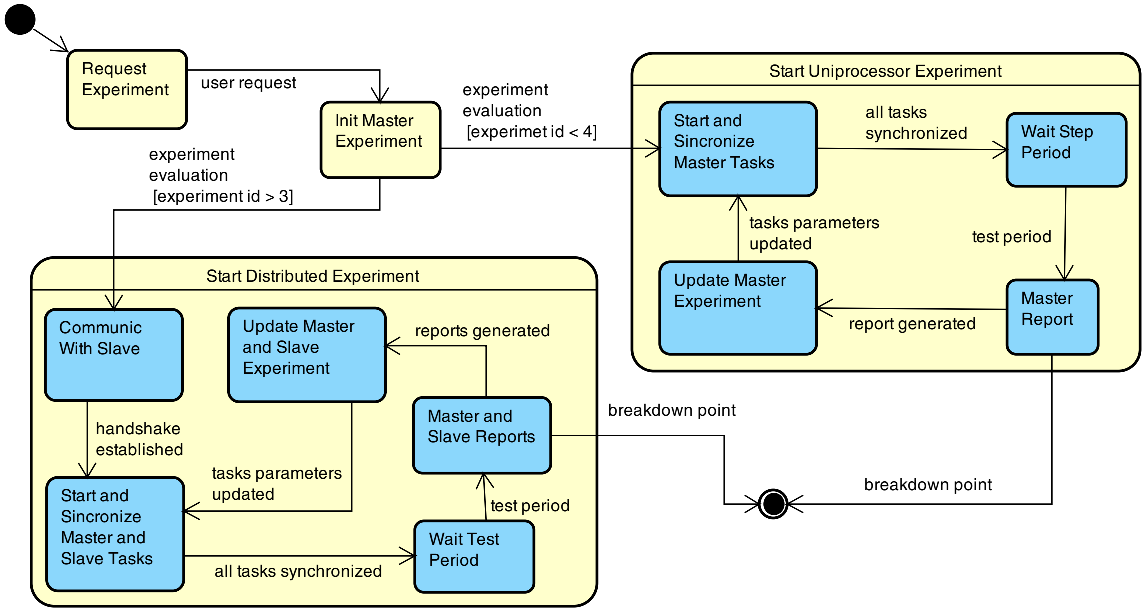
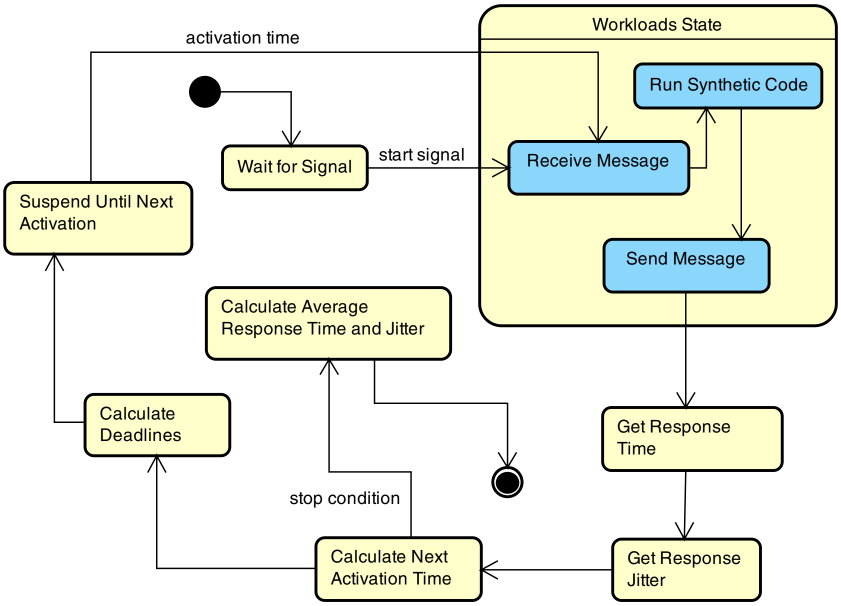


Fig. 10 The Robotstone application layer state machine diagram in UML

The generic state machine diagram in the UML of a synthetic task τ_i is presented in Fig. 11. A task initiates in the state "Wait for Signal." All tasks are together in this initial state, waiting for a signal from the management task. A task leaves this state when it receives the start signal, and it has a chance to use the processor. When this happens, the task enters the "Workloads State."

In PD related experiments, each task is attributed to a synthetic workload, and they do not communicate with each other, so there is no message transaction. In this case the tasks go out from "Receive Message" and "Send Message" states as soon as they enter. They perform work by running synthetic code only in the "Run Synthetic Code" state. In PCD experiments, after executing the synthetic code, a publisher task sends a message in "Send Message"; when task is subscriber, it waits to receive a complete message in "Receive Message", and then it executes the workload. After a task leaves "Workloads State," as indicated in Fig. 11, fine-grained measurements are performed. The response time $R_{i,k}$ of the current instance $\tau_{i,k}$ is obtained from the difference between the instant the task enters the state "Get Response Time" and the instant of activation.

Fig. 11 The generic state machine diagram in the UML of a synthetic task



In the same state, the value obtained is inserted in a summation for the calculation of Eq. 4 and evaluated for the determination of R_i (3). Soon after, in the "Get Response Jitter" state, $R_{i,k}$ is inserted in another summation to determine J_i^R (5).

In the next state, the next activation time of the task is calculated based on the previous activation and task period. After the task leaves this state, τ_i performs the final calculation of R_i^{avg} and J_i^R for the step if the management task has sent a stop flag by means of a shared variable, and then end its execution. Otherwise, τ_i calculates the number of deadlines reached, missed, and skipped (the meaning of the latter is presented in Section 5), based on the next activation period and the current time. Even if a deadline is missed, the task continues to run until the end of the step. Finally, the task goes into a suspended state until the next activation, so other tasks of lower priority can run. It is important to note that the time is calculated using the Real-Time API, made available by the user through some service of the target system.

4 Embedded System Platforms for Evaluation

As a benchmark case study, two embedded system platforms that are generally used in robotic systems were employed. One of them was the well-known Raspberry Pi$^{\text{TM}}$ 3 Model B (RPi3), a board that embeds a system-on-chip (SoC) designed for projects using general-purpose operating systems (GPOSes). The second board was the NXP FRDM-K22F platform used in MCU applications. A detailed discussion of the settings for embedded platforms will be presented in the remainder of this section.

4.1 RPi3 Configurations

The RPi3 is an embedded system platform used for prototyping general-purpose applications. The main processor unit is a Broadcom BCM2837 SoC, having a quad-core ARM Cortex A53 cluster. The cores can run at 1.2 GHz. This type of system requires the use of external memory for storing programs and data. For permanent information storage, a removable micro SD card is used (one 32 GB card in this work). The program is a GPOS (e.g., Linux) and its system applications. All the information needed for processor execution is loaded from the SD card to a RAM chip with 1 GB capacity. The board has basic features, such as GPIO (general-purpose input/output) pins, as well as some peripherals present in MCUs, such as SPI, I^{2}C, I^{2}S, PWM, and UART. The board also has more complex peripherals not usually present on MCU platforms, such as HDMI, as well as the MIPI DSI and MIPI CSI interfaces for high resolution displays and cameras, respectively.

Table 1 Properties of the baseline tasks for PD-related experiments on RPi3

Task	Frequency (Hertz)	Period (ms)	Workload (KWIPP)	Processor Utilization (%)
τ_1	63	15.90	196	3
τ_2	30	33.33	413	3
τ_3	14	71.43	886	3
τ_4	10	100	1240	3
τ_5	6	166.67	2067	3

The Linux distributions available to RPi3 do not provide services for hard RTS applications. For this purpose, some kernel patching or modification is required. Of the many possible ways to make Linux more real-time compliant, one is by using a *real-time co-kernel* [57, 65]. The GPOS is seen as a lower priority task by the co-kernel, so it will only run if no other real-time tasks are running. In this work, the Raspbian with Linux kernel 4.9.80 was used, coupled with the Xenomai co-kernel version 3.0.8[2]. The toolchain used was the *arm-linux-gnueabihf-gcc* version 5.4.0. For time measurement during the benchmark testing, Xenomai timing services were provided to the Real-Time API layer. The default resolution of 1 *ns* was maintained.

Xenomai tasks are created in Linux processes. As many tasks as possible are created in one process. The moment a task is started in a process, Linux is preempted by I-pipe and will resume execution only when no other real-time task is ready. These tasks may communicate with some Linux processes (e.g., a GUI), but the services provided by these processes are not predictable. Although RPi3 has a 32 GB secondary memory, OS swap needs to be disabled to guarantee system predictability. Thus, the application is limited by the 1 GB of main memory. When communication between tasks and peripherals is required, a *real-time driver model* (RTDM) [34] interface for device drivers should be used to ensure timing predictability. Xenomai provides real-time task management services and an FPP scheduler so RM can be applied to the benchmark.

On this platform, we wanted to evaluate the capability of its SoC core to schedule tasks in real time under Xenomai. For this purpose, PD experiments were used. The speed of a system processor core, obtained by the benchmark, was 413,500 KWIPS. In that manner, we could obtain the parameters of the baseline tasks by means of Eq. 2. These parameters are summarized in Table 1. The Master node is a process that starts on the core, and is responsible for creating

[2] The image was downloaded from http://www.cs.ru.nl/lab/xenomai/raspberrypi.html.

the management task, which starts the experiment based on the terminal user request.

We also wanted to evaluate the predictability of the system when there is communication between the SoC cores through the IPC mechanism provided by Xenomai, i.e., communication between tasks on different cores. To do so, PCD experiments were run on the platform. Because Xenomai, like many RTOSes, does not offer a native publisher/subscriber mechanism, an implementation from scratch of this mechanism was required. In that manner, the Xenomai queue services were handled in a specific implementation inside the "Communication Subsystem" (Fig. 8). This implementation intends to be real-time and can be verified in the article published before the current work [48]. Partitioning [18] of the cores was performed using the Linux `taskset` tool: one core for the Master, and another for the Slave. The baseline task parameters for the PCD experiments are presented in Table 2. The task workloads were obtained with the same processor speed used in PD (413,500 KWIPS).

4.2 FRDM-K22F Configurations

The FRDM-K22F is an evaluation board for MCU development. Like RPi3, the FRDM-K22F was made for general-purpose applications, although it is a less complex system. It contains a Kinetis MK22FN512VLH12 MCU, which has an ARM Cortex M4 that runs at up to 120 MHz. It also has a floating point unit and a digital signal processor (DSP) module. The memory system bus is a Harvard type, and it is internal to the chip. It consists of 512 KB flash memory for the program, and 128 KB SRAM memory for the data. The program may be firmware made without any OS support (*baremetal* design) or using an RTOS. In addition, it has a variety of peripherals internal to the chip.

In this work, FreeRTOS^{TM} version 9.0.0 was embedded in the FRDM-K22F MCU as the RTOS, provided by NXP on the MCUXpresso version 11.0.0 IDE. The toolchain used was the *gcc* v. 8-2018-q4-major and the *NewlibNano* C library. Like Xenomai on RPi3, FreeRTOS provides services for task management and an FPP scheduler for the application of RM policy in benchmark testing. The system software with this RTOS is like any MCU baremetal design: tasks and scheduler are functions in the C language that are started inside the main function. In same manner, communication with peripherals is made directly with the use of registers.

For this platform, we wanted to evaluate the capability of the MCU to schedule tasks in real-time under FreeRTOS. For this purpose, we applied the PD experiments. With a processor speed equal to 3758 KWIPS, the parameters of the baseline tasks were obtained using Eq. 2, and are summarized in Table 3. Note that the utilization for the highest priority tasks is less than the others due to rounding error. However, this does not affect the use of the benchmark because accurate values are not required to give the figure of merit for the system. The Master node is the entire system application, and it is responsible for creating the management task, which starts the experiment in response to a terminal user request in PC by means of serial communication. For time measurement in the benchmark test, the FreeRTOS timing service that returns the scheduler ticks is provided to the Real-Time API layer. The tick period resolution was set to 1 ms, which is generally the minimum value recommended by the FreeRTOS community [10].

For this platform, we were not interested in evaluating capabilities related to the communication system. This is because for this kind of hardware (MCU with a single core) and system software (a microkernel), the communication system is generally implemented using shared variables in sections protected by race conditions [36, 38]. In this case, the blocking time of a task, by waiting to be released in a shared region, can counts as part of the task execution time [53]. Thus, it is important to analyze the PD aspects in isolation.

5 Experimental Results and Evaluation

This section presents the experimental results for FRDM-K22F and RPi3. A comparison between the PD related experiments from both platforms is also presented. The system configuration at the breakdown point in each

Table 2 Properties of the baseline tasks for PCD-related experiments on RPi3

Task	Frequency	Period	Message Size	Workload	Processor
	(Hertz)	(ms)	(Bytes)	(KWIPP)	Utilization (%)
τ_{P1} - τ_{S1}	7	142.85	4	2953	5
τ_{P2} - τ_{S2}	5	200.00	4	4135	5
τ_{P3} - τ_{S3}	3	333.33	4	6891	5

Table 3 Properties of the baseline tasks for PD-related experiments on FRDM-K22F

Task	Frequency	Period	Workload	Processor
	(Hertz)	(ms)	(KWIPP)	Utilization (%)
τ_1	63	15.90	1	2
τ_2	30	33.33	3	2
τ_3	14	71.43	8	3
τ_4	10	100	11	3
τ_5	6	166.67	18	3

Table 4 Results of the PD-related experiments for tests in breakdown point on RPi3

Exp.	Test N°	Deadlines Met	Deadlines Miss	Deadlines Skip	Processor Utilization
1	52	1174	6	12	93%
2	50	7388	11	22	90%
3	29	5112	20	40	99%

benchmark experiment is discussed, and the analyses are shown.

5.1 RPi3 Results

Table 4 presents the results obtained at the breakdown points in the PD experiments. The field "Test N°" indicates the number of the test in the experiment at which the breakdown occurred. In the "Deadlines Met" field, number of deadlines met is indicated. In "Deadlines Miss", the number of deadlines missed by a job that started its execution is shown. The "Deadlines Skip" field indicates how many deadlines were "skipped" because jobs were not created during the corresponding activation periods. Finally, processor utilization values are presented.

The deadlines missed for all experiments, presented in Table 4, correspond to the lowest priority tasks. As these have their executions postponed until no higher priority task is ready to run, the results are consistent with what is expected with RM scheduling. Furthermore, for RM scheduling, regardless of the task workload and frequency, the scheduling is guaranteed in the design phase for five tasks only if processor utilization is less than 74% (1). For all the experiments, the tasks passed the RM test by a wide margin, whether by increasing their workload, frequency, or number of tasks. Thus, the system configuration presents excellent real-time performance for all the proposed scenarios related to PD.

In Tables 5 and 6, the results obtained at the breakdown point in the PCD experiments are shown for Master and Slave nodes, respectively. Their fields are similar to those of Table 4, except for the presence of message size.

Table 5 Results of the PCD-related experiments for tests in breakdown point of Master node on RPi3

Exp.	Test N°	Deadlines Met	Deadlines Miss	Deadlines Skip	Processor Utilization	Message Length
4	33	165	0	0	64%	4 B
5	20	165	0	0	15%	2 MB
6	38	848	0	0	72%	4 B
7	17	178	0	0	15%	4 B

Table 6 Results of the PCD-related experiments for tests in breakdown point of Slave node on RPi3

Exp.	Test N°	Deadlines Met)	Deadlines Miss	Deadlines Skip	Processor Utilization	Message Length
4	33	163	1	2	64%	4 B
5	20	165	0	0	15%	2 MB
6	38	844	2	4	72%	4 B
7	17	1119	1	2	95%	4 B

The results related to behavior of deadlines missed obtained in the PCD experiments are similar to those of the PD experiments. The deadlines missed in all PCD experiments correspond to the lowest priority tasks. In addition, the deadlines missed and skipped correspond to Slave only. The presented results indicate good real-time performance of the system as well. In Experiment 4, the processor utilization reached 64% on both nodes. By applying (1), we verified that for three tasks (as present in each node), the schedulability was guaranteed if processor utilization was less than 78%. Although there was a margin of 14% that the system had to fill to attempt what is expected for RM scheduling, it is understandable that gap because the communication system would consume processor time to message transaction. The communication system is implemented with the Xenomai queuing mechanism, which runs on the same processors as the benchmark applications. This time could be considered blocking time in the WCET estimation (see the workflow presented in Fig. 2). In Experiment 5, the messages length at the breakdown point was 2 MB. In this case, there was no deadline missed, because the system stopped when Xenomai was required to allocate 4 MB to message queues. Thus, the system presented timing predictability (tasks completing before the deadlines) when messages were 2 MB. This feature is used in a considerable number of robotic applications, such as that using odometry with encoders, obstacle avoidance with ultrasonic sensors, and many others that do not use large message transactions [21, 54, 55]. In Experiment 6, we observed a behavior similar to Experiment 4. The processor utilization reached 72% in both nodes. Again, there was a gap between the 78% expected by RM scheduling for three tasks (1), but it was expected due to the overhead inserted by the communication system. Thus, we consider this to be a good real-time performance of the system when handling the scheduled load. Finally, in Experiment 7, the Slave node achieved 95% processor utilization when running 19 tasks, where 16 of them were similar to τ_{S2}. This is excellent real-time performance when the system handles a scheduling overload by increasing tasks, because the processor utilization exceeded what was expected by RM for 19 tasks, i.e, 71% (1).

Therefore, the system presented considerable real-time performance in system scenarios that imposed processing and communication workloads.

5.2 FRDM-K22F Results

The results obtained at the breakdown point in the PD experiments on FRDM-K22F are presented in Table 7. As expected with the use of RM scheduling, all the deadlines not met correspond to the lowest priority tasks, as these tasks have their execution postponed until no higher priority task is ready to run. Except for Experiment 2, the other two experiments present a processor utilization that exceeds what is expected by RM scheduling, i.e., 74% (1), by a wide margin. Thus, the system configuration presents excellent real-time performance for all the scenarios.

5.3 PD Results Comparison

Both platforms achieved the expected results from the PD experiments when using RM scheduling, thus presenting significant real-time performance when the system workload and scheduling load were stressed in regular application scenarios. The RPi3 showed better processor utilization. The cause of this may be due to the higher frequency of Xenomai scheduler switch context compared to FreeRTOS. In that manner, FreeRTOS has a bigger gap between one context switch and another, which ends up releasing a task ready to run too late.

To do a more in-depth comparison between platforms, Figs. 12 to 13 present the charts of fine-grained measurements related to Experiment 1 for RPi3 and FRDM-K22F, respectively. Each graph shows the WCRT, ACRT, and response jitter of the synthetic task through the test steps from the experiment. Note that the scales are different for better visualization of the three parameters for each task.

As can be seen for both platforms, all the response times are increasing because the workload of the tasks increases. The higher the task priority, the shorter its response time. In the beginning, all the jitters are small, but they become higher for the lowest priority tasks as the system workload increases. As expected, the highest priority task τ_1 has the smallest jitter, which remains almost the same through the tests because other tasks do not interfere with it. As shown, the ACRT and WCRT curves for task τ_1 are very close to each other, and this is an indication of system predictability [7]. On the other hand, as the task priority decreases, response time increases more quickly during the experiment, mainly because it suffers more interference by the higher priority task workload.

Table 7 Results of the PD-related experiments for tests in breakdown point on FRDM-K22F

Exp.	Test N°	Deadlines Met	Deadlines Miss	Deadlines Skip	Processor Utilization
1	58	1267	2	4	85%
2	48	8730	24	48	75%
3	27	4896	16	32	90%

The variations that occur in the WCRT charts for RPi3 may have been caused by many factors, such as task execution time overloads, or even overloads caused by system services. On FRDM-K22F we had a larger time interval unit than on RPi3, which gave more discretized results.

Another fact that must be noted is that although both systems showed higher processor utilization in PD-related experiments, RPi3 had a higher processor speed and was capable of handling a more significant workload. For example, in Experiment 1, the system workload reached 378,000 KWIPP, where for FRDM-K22F, this value was 3170 KWIPP. This is two orders of magnitude higher, which implies that much more code can be executed in real-time. Therefore, it can be verified that the target systems presented behaviors that corroborate RTS theory and show the applicability of the benchmark to present these expected behaviors.

6 Conclusion

This paper discussed the novel Robotstone benchmark as a methodology for evaluating the real-time performance of computing platforms, with a particular focus on robotic systems. The Robotstone provided seven experiments, each one focusing on the analysis of a system's limit for a specific application scenario. Unlike the original Hartstone benchmark, it is easy to change the task parameters and perform tests with a configuration closer to the expected load of a real robotic system being designed. The breakdown point offered a figure of merit for how adequate the system's real-time capabilities were for the application scenario. User intervention in the benchmark code is needed to port the target system. For that, the target system must provide basic task services and an FPP scheduler to apply the RM policy. The PD experiments evaluated the real-time capabilities of the system related to task scheduling. The PCD experiments considered the real-time capabilities of the system related to task execution and message transaction. In this case, the communication system is seen by the benchmark as a publisher/subscriber mechanism.

Evaluation using the Robotstone benchmark was performed in two case studies of embedded platforms, the popular RPi3, and the microcontroller platform FRDM-K22F.

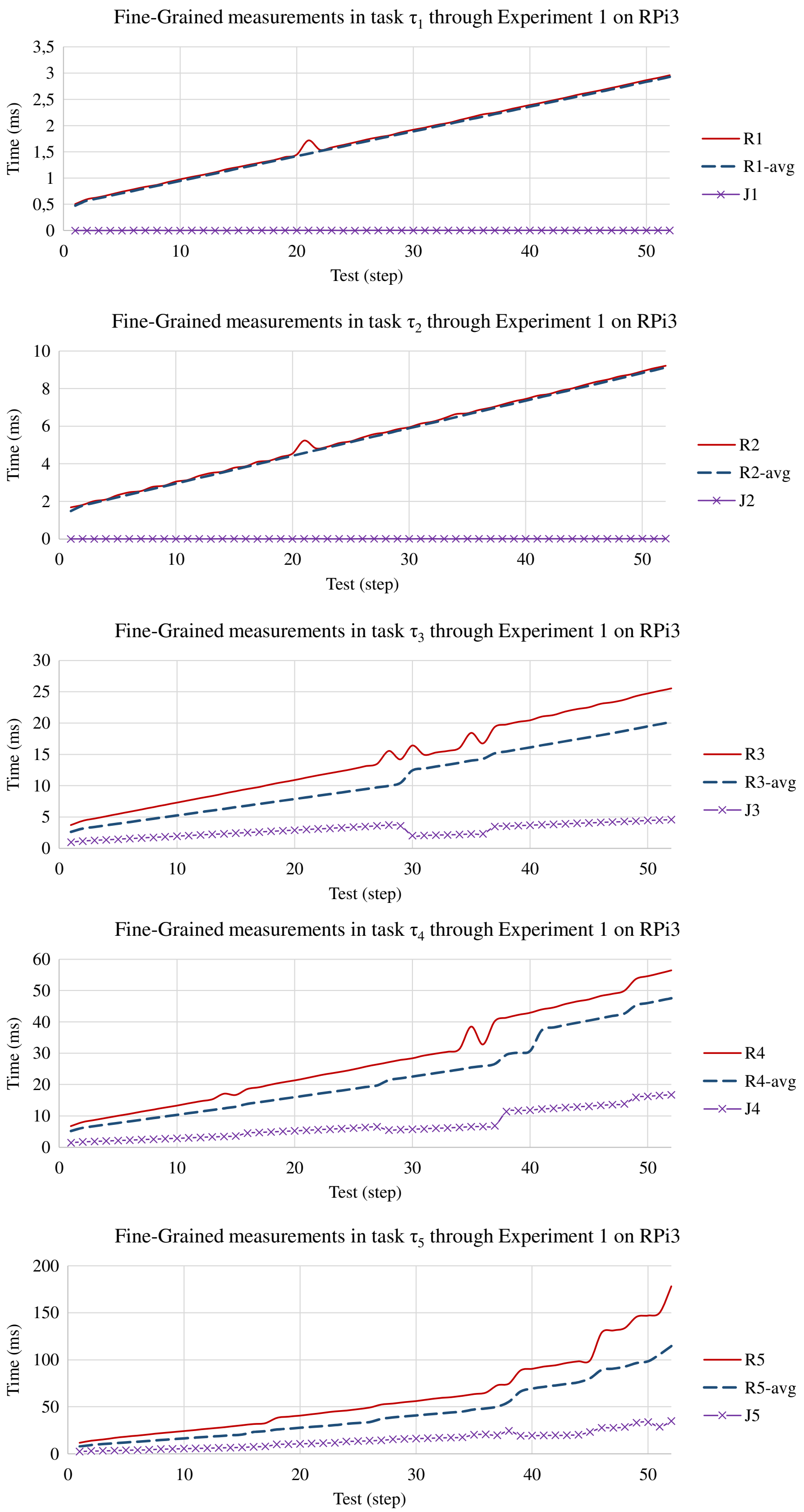
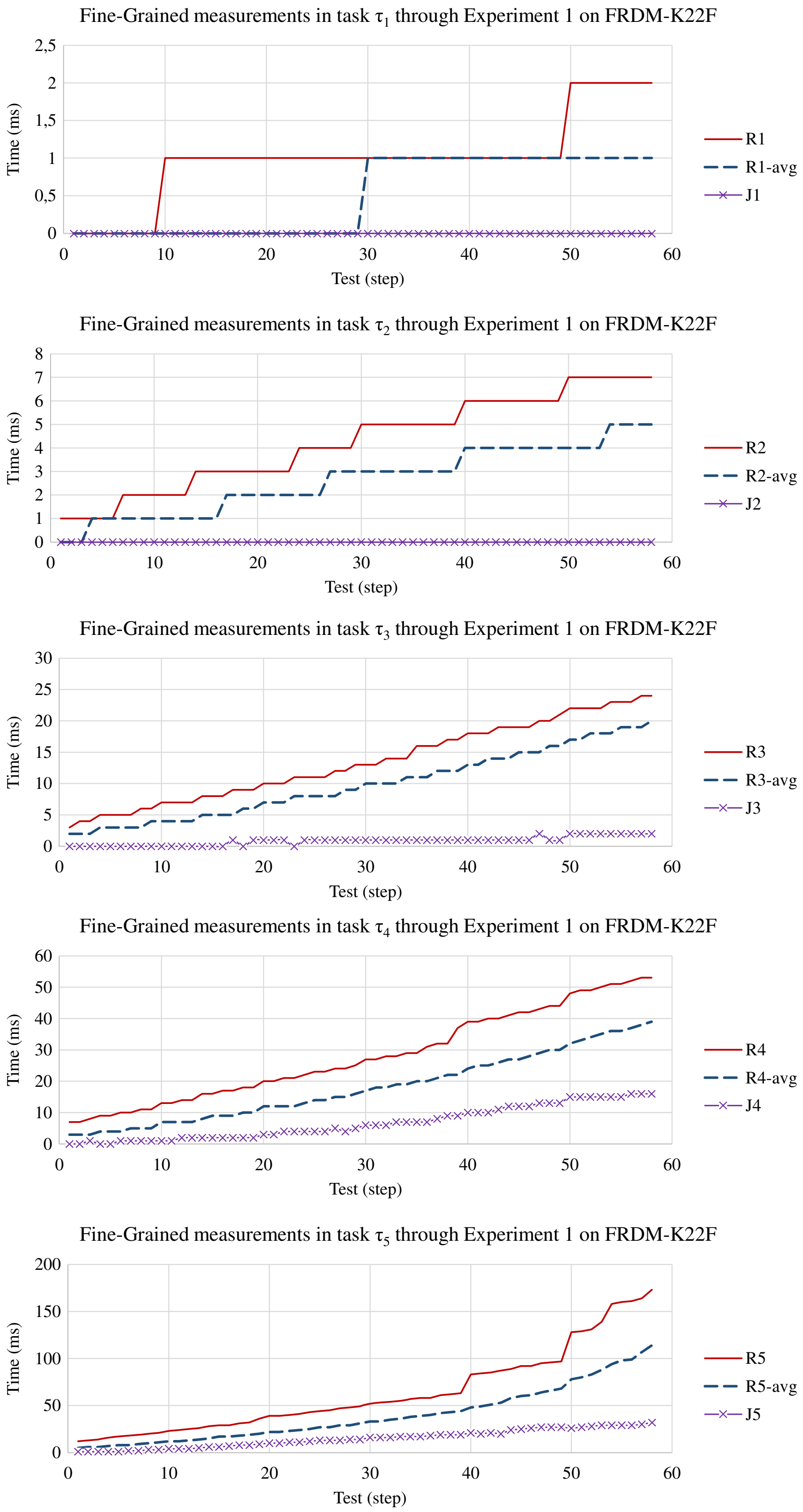


Fig. 12 Fine-grained measurements results for baseline tasks $\{\tau_i\} = \{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5\}$ related to Experiment 1 on RPi3. The curves present the values of WCRTs R_i, ACRTs R_i^{avg} and response jitters J_i^R

The results showed that both platforms show reasonable real-time performance in the PD, because most task sets passed the RM scheduling by a wide margin, i.e., the watermark of 74% of processor utilization. In addition, they presented features that corroborated real-time system theory and benchmark capability to track these features.

For RPi3, the RTOS IPC mechanism between two cores of the SoC was evaluated under the publisher/subscriber

Fig. 13 Fine-grained measurements results for baseline tasks $\{\tau_i\} = \{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5\}$ related to Experiment 1 on FRDM-K22F. The curves present the values of WCRTs R_i, ACRTs R_i^{avg} and response jitters J_i^R


Fine-Grained measurements in task τ_1 through Experiment 1 on FRDM-K22F

Fine-Grained measurements in task τ_2 through Experiment 1 on FRDM-K22F

Fine-Grained measurements in task τ_3 through Experiment 1 on FRDM-K22F

Fine-Grained measurements in task τ_4 through Experiment 1 on FRDM-K22F

Fine-Grained measurements in task τ_5 through Experiment 1 on FRDM-K22F

paradigm. The system also presented significant real-time performance when message transactions were involved, because the system maintained timing constraints with high processor utilization. The watermark in this case was 78% of processor utilization, which was expected for three baseline tasks. In Experiment 7, the processor utilization when increasing subscribers reached 95%. Although there were gaps between the processor utilization achieved in

Experiments 4 and 6 compared with the watermark, we considered that to be acceptable because the load inserted by the communication system consumed a portion of the processor time. In Experiment 5, there was no deadline miss, because system reached the breakdown point after trying to create topics with 4 MB messages. So, the benchmark tracked a maximum value of 2 MB for messages transactions in real-time. This message size limit includes a great portion of robotics applications.

The proposed benchmark is meant to be used as a starting point for robotics engineers who are developing applications with timing requirements, and it offers more evidence for the correct choice of the embedded platform. Also, as future research, it would be interesting to evaluate distributed systems, i.e., platforms that communicate with each other through network links. Specially, these systems could use robotic real-time middleware. Furthermore, benchmark features can be extended to accommodate more experiments and other kinds of real-time schedulers and communication paradigms.

References

1. Abella, J., Hernandez, C., Quiňones, E., Cazorla, F.J., Conmy, P.R., Azkarate-Askasua, M., Perez, J., Mezzetti, E., Vardanega, T.: WCET analysis methods: Pitfalls and challenges on their trustworthiness. In: 10th IEEE International Symposium on Industrial Embedded Systems (SIES), pp. 1–10. IEEE, Siegen (2015)
2. Amazon Web Services: The FreeRTOSTM Reference Manual. https://www.freertos.org/wp-content/uploads/2018/07/FreeRTOS Reference Manual_V10.0.0.pdf (2017). Accessed 23 July 2020
3. Ando, N., Suehiro, T., Kitagaki, K., Kotoku, T., Yoon, W.K.: RT-Middleware: Distributed component middleware for RT (Robot Technology). In: 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 3933–3938. IEEE, Edmonton (2005)
4. Anh, T.N.B., Tan, S.L.: Real-time operating systems for small microcontrollers. IEEE Micro **29**(5), 30–45 (2009)
5. Arad, B., Kurtser, P., Barnea, E., Harel, B., Edan, Y., Ben-Shahar, O.: Controlled lighting and illumination-independent target detection for real-time cost-efficient applications. the case study of sweet pepper robotic harvesting. Sensors **19**(6), 1390 (2019)
6. Asadi, K., Ramshankar, H., Pullagurla, H., Bhandare, A., Shanbhag, S., Mehta, P., Kundu, S., Han, K., Lobaton, E., Wu, T.: Vision-based integrated mobile robotic system for real-time applications in construction. Autom. Constr. **96**, 470–482 (2018)
7. Axer, P., et al.: Building timing predictable embedded systems. ACM Trans. Embedded Comput. Syst. 13(4) (2014)
8. Barbalace, A., Luchetta, A., Manduchi, G., Moro, M., Soppelsa, A., Taliercio, C.: Performance comparison of VxWorks, Linux, RTAI, and Xenomai in a hard real-time application. IEEE Trans. Nucl. Sci. **55**(1), 435–439 (2008)
9. Bardaro, G., Semprebon, A., Matteucci, M.: A use case in model-based robot development using Aadl and Ros. In: Proceedings of the International Workshop on Robotics Software Engineering, pp. 9–16. ACM, New York (2018)
10. Barry, R.: Mastering the FreeRTOS^{TM} Real Time Kernel – A Hands-On Tutorial Guide. https://www.freertos.org/wp-content/uploads/2018/07/161204%_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf (2016). Accessed 23 July 2020
11. Bruyninckx, H.: Open robot control software: The OROCOS project. In: Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation, vol. 3, pp. 2523–2528. IEEE, Korea (2001)
12. Bruyninckx, H.: Open robot control software: The OROCOS project. In: Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No.01CH37164), vol. 3, pp. 2523–2528 vol.3 (2001)
13. Bruyninckx, H.: OROCOS: Design and implementation of a robot control software framework. In: Proceedings of IEEE International Conference on Robotics and Automation. IEEE (2002)
14. Bucci, G., Carnevali, L., Ridi, L., Vicario, E.: Oris: A tool for modeling, verification and evaluation of real-time systems. Int J Softw Tools Technol Trans **12**(5), 391–403 (2010)
15. Bäuml, B., Hirzinger, G.: When hard realtime matters: Software for complex mechatronic systems. Robot. Autonomous Syst. **56**(1), 5–13 (2008). Human Technologies: ”Know-how”
16. Buschmann, F.: Pattern-oriented Software Architecture, a System of Patterns, vol. 1. Wiley, New Jersey (1996)
17. Butazzo, G.C. Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, 3rd edn. Springer, New York (2011)
18. Carpenter, J., Funk, S., Holman, P., Srinivasan, A., Anderson, J.H., Baruah, S.K.: A Categorization of real-time multiprocessor scheduling problems and algorithms. In: Handbook of Scheduling, pp. 676–695. Chapman and Hall/CRC, Boca Raton (2004)
19. Curnow, H.J., Wichmann, B.A.: A synthetic benchmark. Comput. J. **19**(1), 43–49 (1976)
20. Donohoe, P., Shapiro, R., Weiderman, N.: Hartstone Benchmark User’s Guide. Tech. Rep., Software Engineering Institute, Carnegie Mellon University Technical Report CMU-SEI- 90-TR-1 (1990)
21. Dudek, G., Jenkin, M.: Computational Principles of Mobile Robotics. Cambridge university press, Cambridge (2010)
22. Elkady, A., Sobh, T.: Robotics middleware: A comprehensive literature survey and attribute-based bibliography. J. Robot. **2012**, 01–15 (2012)
23. Embedded: How to make C++ more real-time friendly. https://www.embedded.com/how-to-make-c-more-real-time-friendly/ (2014). Accessed 23 July 2020
24. Ferdinand, C., Heckmann, R., Langenbach, M., Martin, F., Schmidt, M., Theiling, H., Thesing, S., Wilhelm, R.: Reliable and Precise Wcet determination for a real-life processor. In: Henzinger, T.A., Kirsch, C.M. (eds.) Embedded Software, pp. 469–485. Springer, Berlin (2001)
25. Flores, G., Zhou, S., Lozano, R., Castillo, P.: A vision and gps-based real-time trajectory planning for a mav in unknown and low-sunlight environments. J. Intell. Robot. Syst. **74**(1-2), 59–67 (2014)
26. Gerum, P.: Xenomai - Implementing a RTOS emulation framework on GNU/Linux. https://design.ros2.org/articles/realtime_background.html (2004). Accessed 23 July 2020
27. Hammer, T., Bäuml, B.: The communication layer of the ardx software framework: Highly performant and realtime deterministic. J. Intell. Robot. Syst. **77**(1), 171–185 (2015)
28. Heckmann, R., Langenbach, M., Thesing, S., Wilhelm, R.: The influence of processor architecture on the design and the results of wcet tools. Proc. IEEE **91**(7), 1038–1054 (2003)

29. Jo, K., Kim, J., Kim, D., Jang, C., Sunwoo, M.: Development of autonomous car – Part II: A Case study on the implementation of an autonomous driving system based on distributed architecture. IEEE Trans. Ind. Electron. **62**(8), 5119–5132 (2015)
30. Joseph, M., Pandya, P.: Finding response times in a Real-Time system. Comput. J. **29**(5), 390–395 (1986)
31. Kamal, R.: Embedded Systems: Architecture, Programming and Design. Tata McGraw-Hill Education, New York (2011)
32. Kamenoff, N.I., Weiderman, N.H.: Hartstone distributed benchmark: Requirements and definitions. In: Real-Time Systems Symposium, 1991. Proceedings. Twelfth, pp. 199–208. IEEE (1991)
33. Kerstens, R., Laurijssen, D., Steckel, J.: Ertis: A fully embedded real time 3D imaging sonar sensor for robotic applications. In: 2019 International Conference on Robotics and Automation (ICRA), pp. 1438–1443. IEEE (2019)
34. Kiszka, J.: The real-time driver model and first applications. In: 7Th Real-Time Linux Workshop, Lille, France (2005)
35. Klein, M.H., Ralya, T., Pollak, B., Obenza, R., Harbour, M.G.: A Practioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems. Kluwer Academic Publishers, Dordrecht (1993)
36. Kopetz, H.: Real-Time Systems: Design principles for distributed embedded applications. Springer Science & Business Media, Berlin (2011)
37. Krten, R.: Getting Started with QNX Neutrino 2: A Guide for Realtime Programmers. PARSE Software Devices, Kanata (1999)
38. Lee, E.A., Seshia, S.A. Introduction to Embedded Systems - A Cyber-Physical Systems Approach, 2nd edn. MIT Press, Cambridge (2017)
39. Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard-real-time environment. J. ACM (JACM) **20**(1), 46–61 (1973)
40. Liu, J.W.S.: Real-Time Systems. Prentice Hall, Upper Saddle River (2000)
41. Magyar, G., Sinčák, P., KrizsáN, Z.: Comparison study of robotic Middleware for robotic applications. In: Sinčák, P., Hartono, P., Virčíková, M., Vaščák, J., Jakša, R. (eds.) Emergent Trends in Robotics and Intelligent Systems, pp. 121–128. Springer International Publishing, Cham (2015)
42. Maruyama, Y., Kato, S., Azumi, T.: Exploring the performance of ROS2. In: Proceedings of the 13th International Conference on Embedded Software, p. 5. ACM (2016)
43. Metta, G., Fitzpatrick, P., Natale, L.: Yarp: Yet another robot platform. Int. J. Adv. Robot. Syst. **3**(1), 8 (2006)
44. Muratore, L., Laurenzi, A., Hoffman, E.M., Rocchi, A., Caldwell, D.G., Tsagarakis, N.G.: Xbotcore: A real-time cross-robot software platform. In: 2017 First IEEE International Conference on Robotic Computing (IRC), pp. 77–80. IEEE (2017)
45. Nise, N.S. Control Systems Engineering, 6th edn. Wiley, New Jersey (2011)
46. Open Source Robotics Foundation: Introduction to real-time systems. https://design.ros2.org/articles/realtime_background.html (2019). Accessed 23 July 2020
47. Perathoner, S., Wandeler, E., Thiele, L., Hamann, A., Schliecker, S., Henia, R., Racu, R., Ernst, R., Harbour, M.G.: Influence of different system abstractions on the performance analysis of distributed real-time systems. In: Proceedings of the 7th ACM & IEEE International Conference on Embedded Software, pp. 193–202 (2007)
48. Pinto, M.L., de Oliveira, A.S., Wehrmeister, M.A.: Real-time systems evaluation for robotics using the Hart-ROS benchmark. In: 2019 19th International Conference on Advanced Robotics (ICAR), pp. 296–301. IEEE, Belo Horizonte (2019)
49. Quigley, M., Faust, J., Foote, T., Leibs, J.: Ros: An open-source robot operating system. In: International Conference on Robotics and Automation (ICRA) - Workshop on Open Source Software, vol. 3, p. 5P (2009)
50. Rajkumar, R., Gagliardi, M., Sha, L.: The real-time Publisher/Subscriber Inter-Process communication model for distributed real-time systems: Design and implementation. In: 1st IEEE Real-Time Technology and Applications Symposium, pp. 66–75. IEEE, United States (1995)
51. Redmon, J., Angelova, A.: Real-time grasp detection using convolutional neural networks. In: 2015 IEEE International Conference on Robotics and Automation (ICRA), pp. 131–1322. IEEE (2015)
52. RobMoSys EU H2020 Project (2017–2020). RobMoSys: Composable Models and Software for Robtics Systems - Towards an EU Digital Industrial Platform for Robotics. https://robmosys.eu (2020). Accessed 23 July 2020
53. Sha, L., Rajkumar, R., Lehoczky, J.P.: Priority inheritance protocols: an approach to real-time synchronization. IEEE Trans. Comput. **39**(9), 1175–1185 (1990)
54. Siciliano, B., Khatib, O.: Springer Handbook of Robotics. Springer, Berlin (2016)
55. Siegwart, R., Nourbakhsh, I.R., Scaramuzza, D., Arkin, R.C. Introduction to Autonomous Mobile Robots, 2nd edn. MIT press, Massachusetts (2011)
56. Stankovic, J.A.: Misconceptions about real-time computing: A serious problem for next-generation systems. Computer **21**(10), 10–19 (1988)
57. Stankovic, J.A., Rajkumar, R.: Real-time operating systems. Real-Time Syst. **28**(2-3), 237–253 (2004)
58. Subosits, J.K., Gerdes, J.C.: From the racetrack to the road: Real-time trajectory replanning for autonomous driving. IEEE Trans. Intell. Vehicles **4**(2), 309–320 (2019)
59. Tanenbaum, A.S., Van Steen, M.: Distributed Systems: Principles and Paradigms. Prentice-Hall, New Jersey (2007)
60. Wang, M., Liu, J.N.K.: Fuzzy logic-based real-time robot navigation in unknown environment with dead ends. Robot. Auton. Syst. **56**(7), 625–643 (2008)
61. Wehrmeister, M.A., de Freitas, E.P., Binotto, A.P.D., Pereira, C.E.: Combining aspects and object-orientation in model-driven engineering for distributed industrial mechatronics systems. Mechatronics **24**(7), 844–865 (2014). https://doi.org/10.1016/j.mechatronics.201312.008
62. Weiderman, N.H., Kamenoff, N.I.: Hartstone uniprocessor benchmark: Definitions and experiments for real-time systems. Real-Time Syst. **4**(4), 353–382 (1992)
63. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., et al.: The worst-case execution-time problem – overview of methods and survey of tools. ACM Trans. Embedded Comput. Syst. (TECS) **7**(3), 1–53 (2008)
64. Xu, Y., Fang, G., Chen, S., Zou, J.J., Ye, Z.: Real-time image processing for vision-based weld seam tracking in robotic gmaw. Int. J. Adv. Manufact. Technol. **73**(9–12), 1413–1425 (2014)
65. Yaghmour, K., Masters, J., Ben-Yossef, G., Gerum, P. Building Embedded Linux Systems, 2nd edn. O'Reilly, Sebastopol (2008)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Matheus Leitzke Pinto received the degree in computer engineering from the Federal University of Pelotas (UFPel), Brazil, and the master's degree in computer engineering from the Federal University of Technology - Paraná (UTFPR), Brazil. He is also a Professor with Federal Institute of Education, Science and technology of Santa Catarina (IFSC), Brazil. His research interests include embedded systems, real-time systems and autonomous mobile robot.

Marco Aurélio Wehrmeister received the Ph.D. degree (double-degree) in computer science from the Federal University of Rio Grande do Sul, Brazil, and the University of Paderborn, Germany, in 2009. From 2009 to 2013, he worked as a Lecturer and Tenure Track Professor at the Federal University of Santa Catarina and Santa Catarina State University, Brazil, respectively. Since 2013, he has been a Professor at the Department of Informatics, Federal University of Technology - Paraná (UTFPR), Brazil. He has coauthored more than 75 articles and chapters in international peer-reviewed journals, conference proceedings, and international books. His research interests are in the areas of cyber-physical systems and embedded real-time systems, aerial robots, model-driven engineering, and hardware/software engineering and co-design. He is a member and the Elected Chair of the Special Commission on Computing Systems Engineering of the Brazilian Computer Society, from 2018 to 2020, and also a member and the Vice-Chair of the IFIP Working Group 10.2 on Embedded Systems. In 2018, he received the Research Productivity Grant/Award from the Fundação Araucária de Apoio ao Desenvolvimento Científico e Tecnológico do Estado do Paraná (FAPPR). His thesis was selected by the Brazilian Computer Society as one of the six best theses on Computer Science, in 2009. He is a reviewer for various international journals, as well as a member of the technical program committee of various international conferences and symposiums. He has been involved in various research projects funded by Brazilian Companies and Research and Development agencies, especially on model-driven engineering of cyber-physical systems and aerial robots targeting search and rescue and industrial applications, as well as assets and equipment inspection.

André Schneider de Oliveria (Member,IEEE) received the M.Sc. degree in mechanical engineering, concentrate in force control of rigid manipulators from the Federal University of Santa Catarina (UFSC), in 2007, and the Ph.D. degree in engineering of automation and systems, in 2011. His thesis focused on differential kinematics through dual-quaternions for vehicle-manipulator systems. He is currently an Adjunct Professor with the Federal University of Technology - Paraná (UTFPR) and a member with the Advanced Laboratory of Robotics and Embedded Systems (LASER) and the Laboratory of Automation and Advanced Control Systems (LASCA). His research interests include robotics, mechatronics and automation, with particular focus on navigation and localization of mobile robots, autonomous and intelligent systems, perception and environment identification, cognition, deliberative decisions, and human-interaction and navigation control.