Chapter 3:
Dependability Benchmark Real Time Kernels in Onboard Space Systems

Abstract

In space real-time systems, correctness of operation depends not only on the right results being generated but also on the results being generated within certain time constraints. With the increased use of COTS Real-Time Kernels (RTK) in embedded systems the need for assuring a high dependability level of such kernels also arose. Among several dependability attributes, the determinism of the response time of RTK services, even in presence of faults, is of paramount importance for hard real-time systems. This is particularly true for onboard space systems that are more exposed to external disturbances such as radiation.

The benchmark presented in this chapter is targeted for onboard space systems. It aims to allow integrators/developers to compare different RTKs with respect to their ability to provide their services within the expected time frame. The benchmark provides metrics for characterizing this capability. The main measure provided is the **Predictability** which scores a RTK vis-à-vis its system calls response time fitting within its specification.

3.1 Introduction

In real-time systems correctness of operation depends not only on the accuracy of the results provided but also on the results being produced within certain time constraints imposed by the system specification [Laplante 1997]. Real-time embedded systems in general, and embedded systems in the avionics and space domains in particular, have a number of requirements regarding meeting (hard) deadlines. Since most of the functionalities of such systems depend on the (correct and on time) services being provided by the underlying RTK, the correct and predictable timing behaviour of the RTK is crucial. In fact, RTKs must exhibit predictable timing (and value) behaviour despite the occurrence of unpredictable external events [DOT/FAA 2002].

Malfunctioning RTKs may have a strong impact on the dependability of an embedded system. Considering that embedded systems are difficult, and often impossible, to change/correct once deployed, assessing their dependability characteristics is of paramount importance. While there is some work done on the characterisation of failure modes and robustness of real-time kernels [Kropp et al. 1998] [Chevochot and Puaut 2001] [Arlat et al. 2002], a specific methodology to characterize the predictability of the response times of RTKs services is still missing.

This chapter presents the specification of a benchmark for comparing the predictability of response time of a Real-Time Kernel services. This benchmark aims to allow integrators/developers to assess and compare different RTKs with respect to their ability to provide the services within the expected time frame. The benchmark is targeted at space domain systems and characterizes the determinism of the response time of the RTK services addressing the robustness with respect to faulty applications. The measurements collected are combined characterizing the predictability of its response time.

The rest of the chapter is organised as follows. Section 3.2 gives an overview of space domain systems. Section 3.3 presents the benchmark specification while section 3.4 describes one specific implementation of the benchmark. Section 3.5 explains the properties on which the validation of the benchmark was based upon and section 3.6 presents the concluding remarks.

3.2 Basics on Space systems

The classification of a dependability benchmark requires the specification of a benchmark context. Since a space system is normally composed by several different subsystems each with its own characteristics and requirements, the first thing needed to define a benchmark for space systems is to clearly identify the context, and the subsystem that will be the target of the benchmark. Although all space systems are different since their mission purpose and requirements are different it is possible to identify common features and functionalities in all of them that lead to the definition of an abstraction of a space system. In fact, this abstraction is the basis that allows the definition of a benchmark targeted at space systems.

Following this reasoning, the next section presents some basic components of a space system.

3.2.1 Components of Space systems

A typical space mission is made up by several subsystems on three different segments (Figure 3.1):

- The **Space Segment** includes systems like spacecrafts, satellites or rovers carrying a *payload* such as scientific equipment (e.g. telescope) or transponders in case of communication satellites. Additionally to the payload, space segment systems normally include a Control and Data Handling Unit responsible to manage the spacecraft and to provide communication with the ground (control) segment.
- The **Ground Segment** systems are responsible for controlling the mission, for monitoring the spacecraft's orbit and position, and for receiving the science data in case of a science mission.
- The **User Segment** systems that will process the science data provided by the spacecraft.

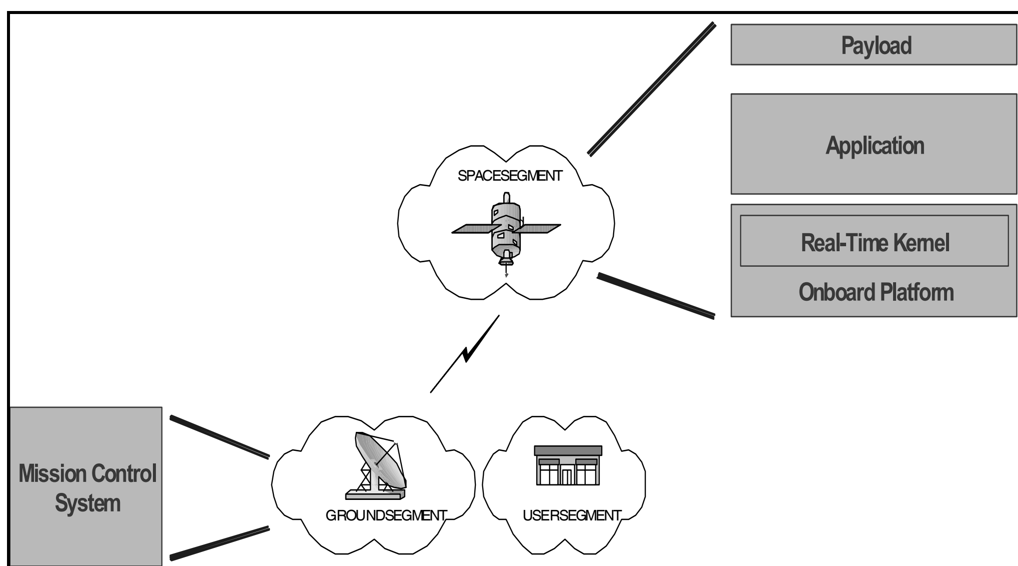

Figure 3.1: General view of a space system

From the three segments, we will focus on the Space Segment. The most common and known systems in the space segment are possible satellites which come in all shapes and sizes and play a variety of roles. For example:

- **Weather satellites** help meteorologists predict the weather or see what's happening at the moment. The satellites generally contain cameras that can return photos of Earth's weather.
- **Communications satellites** allow telephone and data conversations to be relayed through the satellite.
- **Broadcast satellites** relay television signals from one point to another (similar to communications satellites).
- **Scientific satellites** perform a variety of scientific missions. The Hubble Space Telescope is one famous scientific satellite, but there are many others looking at everything from sun spots to gamma rays.
- **Navigational satellites** e.g. help ships and planes navigate. The most famous are the GPS NAVSTAR satellites or the upcoming Galileo system.

- **Earth observation satellites** observe the planet for changes in everything from temperature to forestation to ice-sheet coverage.

Despite the significant differences between all of these satellites, they have several features in common. For example:

- All of them have a metal or composite frame and body, usually known as the **bus** or **platform**. The bus holds everything together in space and provides enough strength to survive the launch.
- All of them have a source of **power** (usually solar cells) and batteries for storage.
- All of them have an **onboard computer** to control and monitor the different systems.
- All of them have a **radio** system and antenna. At the very least, most satellites have a radio transmitter/receiver so that the ground-control crew can request status information from the satellite and monitor its health. Many satellites can be controlled in various ways from the ground to do anything from change the orbit to reprogram the computer system.
- All of them have an **attitude control system** that keeps the satellite pointing in the right direction.

Like in any other industry some degree of standardisation is already available in space systems. For instance, most of ESA recent missions adopt the Packet Utilization Standard [PUS 2003] for the communication between the Ground and Space segments. A telecommand packet is sent to the satellite (e.g. for carrying out a manoeuvre or acquiring science data) that is immediately executed. The answer, either an acknowledge or actual science data, is returned in the form of telemetry packet.

3.3 Benchmark Specification

This section presents DBench-RTK – a Dependability Benchmark specification aimed to characterize the behaviour of real time kernels in an onboard space system in the presence of faults. This benchmark specification instantiates the different benchmark dimensions identified in Chapter 1.

3.3.1 Benchmark Overview

A Real Time Kernel (RTK) is much like a General Purpose Operating System (GPOS) as it also manages aspects of the underlying hardware and provides a set of basic services to the applications. Two main differences can be pointed out between an RTK and a GPOS. A GPOS is typically a monolithic component where application developers have no control on the subsystems they can “ship” with their application. RTKs are typically configurable items that enable downsizing by cutting out in the kernel subsystems not used by the application at hand. Another remarkable difference is the environment in which they are used. An GPOS, due to its general purpose characteristics, and can be used for a wide range of applications such as desktop computers used for word processing and similar uses or backend servers running corporate applications or database servers. RTKs on the other end, are usually used

for embedded systems, many diskless, where the inter process communication and timing issues are more important.

The Benchmark Target (BT) for an RTK dependability benchmark corresponds naturally to an RTK. RTKs (as GPOSs) are used by the upper software layers through their API (Application Programming Interface) that defines the set of services provided by the RTK. In addition, the embedded application (and the RTK) requires a hardware platform and possibly some additional libraries to run. All these “surrounding” system components influence the Benchmark Target behaviour. The System Under Benchmark (SUB) defines then not only the Benchmark Target but also the environment where it is used including as well a reference application.

The dependability benchmark presented in this chapter is more specifically a robustness benchmark. Robustness is one key dependability attribute that impacts the system property we are accessing – *determinism of response time*. Robustness testing has been widely used to assess COTS systems revealing deficiencies especially when its internal structure is unknown (see [Laplante 1997], [Rodriguez et al. 2002], [Moreira et al. 2003] and DOT/FAA 2002]). Robustness testing normally considers the system as a “black box” applying a set of test values to its interface.

In this benchmark we focus on the timing robustness of the RTK with respect to erroneous inputs provided by the application software via the API. The set of measures defined characterise the timing behaviour of the RTK, especially in the case when it does not meet its specifications.

The benchmark specification defines clearly the following items:

1. The dependability measures provided by the benchmark;
2. The system under benchmarking;
3. The experimental dimensions: workload, faultload and procedures.

The next subsections will address these items, while section 3.4 provides the details of a prototype instantiating them.

3.3.2 Dependability measures

On assessing the predictability of the response time of a RTK system call there are two remarkable issues:

- When a system call fails to execute within the nominal time, **how much** it deviates from the specification?
- What is the likelihood that a specific RTK system call will fail to execute within the nominal time? I.e. **how many** times can this happen.

The predictability of a RTK is then based on the following two measures:

- The **Divergence (D),** which represents the normalised difference between longest measured execution time of a system call in presence of faults and the nominal

execution time of that function. The divergence is expressed in percents and gives a measure of the impact or "cost" of the faults in the response time.

$$D = \frac{T_{\max} - T_{no\min al}}{T_{no\min al}}$$

The divergence is calculated individually for each system call and then the average is computed to obtain a single Divergence value for the RTK.

- The **Frequency of "out of boundaries" (F)** corresponds to the number of cases a system call execution took longer then the nominal time considering all executions. The frequency is expressed in percents and represents the probability of a system call not following is specification.

  The frequency is calculated individually for each system call and then the average is computed to obtain a single frequency of out of boundaries value for the RTK.

Based on the two previous measures, a predictability value is computed. The **Predictability (P)** of a RTK is then the probability of a system call failing to execute within its specification time associated with a penalty on the "size" of the average time delay.

$$P = \frac{(1-F)}{(1+D)}.$$

Annex 3-A presents a detailed specification of these measures on which any implementation of the benchmark must comply.

3.3.3 System Under Benchmarking

The Benchmark Target (BT) of this benchmark is a Real-Time Kernel (RTK), which is a small sized software component composed by a set of core functions common in an OS. It offers a number of functions and procedures to manage, for instance, tasks, semaphores, system memory, interrupts and signals, via its Application Programming Interface (API).

The System Under Benchmarking (SUB) is an onboard space system composed by several modules defined as the Workload, The Real-Time Kernel and the Hardware Platform. Furthermore, a Ground Segment Emulator (GSE), running on a different hardware platform, emulates the control and monitoring done by system operators, providing the inputs to the workload. Figure 3.2 depicts the SUB considered for this benchmark.

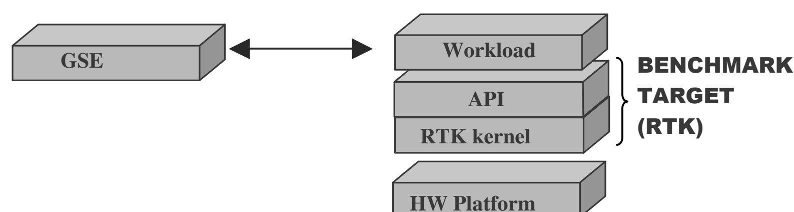

Figure 3.2: System Under Benchmarking

The workload considered exercises a subset of normal control and data handling functionalities used in common space systems and the GSE provides the required commanding and monitoring functionality to exercise the workload and the benchmark target.

3.3.4 Experimental dimensions

This section defines the experimental dimensions of the benchmark, namely the workload, the faultload, the benchmark setup and its procedure.

3.3.4.1 Workload

In the case of onboard space systems, there is no (widely) available performance benchmark which could be adapted to the dependability benchmark requirements (as it happens for instance with OLTP systems). A complete workload is then proposed/specified.

This workload should be representative of the typical applications, algorithms and functionalities running on top of RTK in an onboard space system. One of the functionalities normally included in almost all satellites and spacecrafts which require services from a RTK component is the capability of scheduling telecommands for later execution. There are at least two scenarios where the capabilities for the onboard execution of operations that have been loaded in advance from the ground are useful [PUS 2003]:

- Those missions that perform operations outside of ground contact because of limited ground station visibility or signal propagation delays;
- Those missions whose operations concept is to minimize the dependency on the ground segment. Thus, a geostationary telecommunications or meteorological mission can perform all of its routine operations in this manner, even though the spacecraft is permanently in view of a ground station. This approach potentially increases the availability of operational services or mission products, since the continuous availability of the uplink is eliminated.

The onboard telecommand scheduling functionality will be used as abstraction for the benchmark on real time kernel in onboard space systems.

The workload defined in this benchmark is an Onboard Scheduler (OBS) process based on the onboard telecommand scheduling functionality derived from Packet Utilization Standard ([PUS 2003]). The purpose is to simulate the reception of telecommands, store and dispatch them in accordance to their activation time.

The Onboard Scheduler (OBS) receives telecommands that are recorded to be executed at a specified later time. Figure 3.3 shows the proposed architecture of this reference onboard scheduler. The workload is divided into three major modules (Telecommand Reader, Telecommand Storage and Dispatcher) defined in the following sub-sections. The OBS exercises several kernel functionalities such as task handling, process synchronization, message passing and timer. It receives telecommands from the input channel that are kept in the Telecommand Storage until their release time when they are dispatched through the output channel.

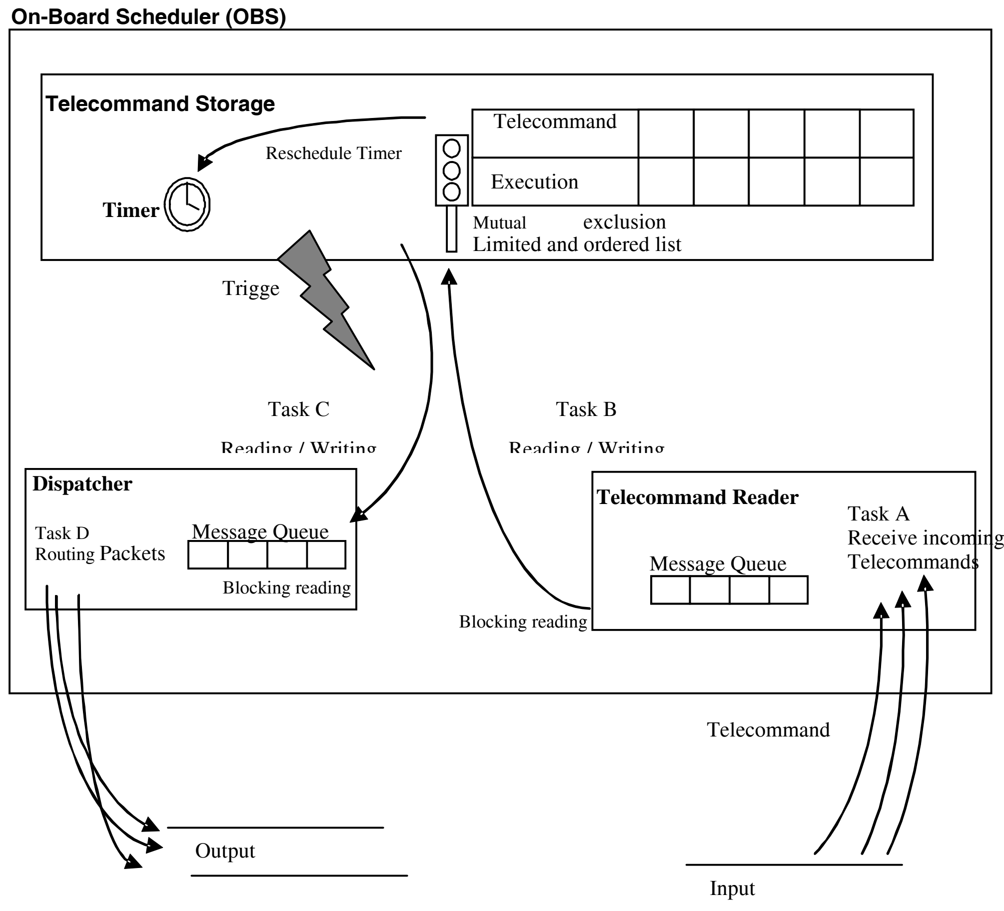

Figure 3.3: Onboard Scheduler architecture

3.3.4.1.1 Telecommand Reader

The Telecommand Reader is responsible for receiving the telecommands to be scheduled. As shown in Figure 3.3, Task A receives the incoming telecommands through the input channel and stores them in an internal message queue. The queue must have a fixed size and its accesses must be synchronized. If the message queue is full, the task(s) shall wait until a message is removed from the queue. Having more than one reader task increases the throughout in the input channel. Figure 3.4 presents a general activity diagram for this module.

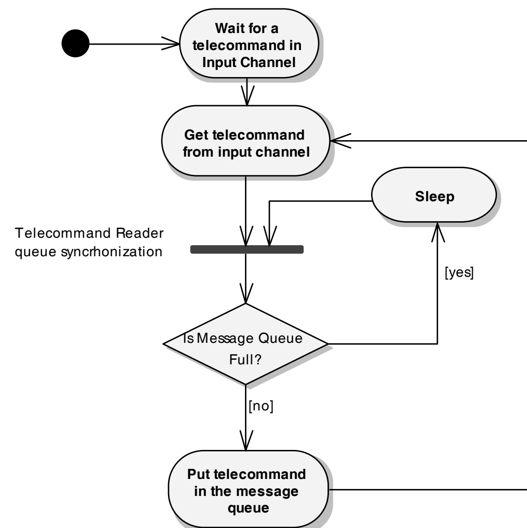

Figure 3.4: Telecommand Reader Activity Diagram (*Task A*)

3.3.4.1.2 Telecommand Storage

Telecommand Storage is responsible for storing the telecommands in a way that allow a quick access to the release time of each telecommand to be executed. The access for the storage structure shall be synchronised and protected. A local semaphore shall be used to accomplish this requirement.

Task B (in Figure 3.3) is responsible for retrieving the telecommands from the Telecommand Reader queue and inserting them in the Telecommands Storage structure. Task B remains blocked until a new TC is inserted in the message queue. A counter with the number of TCs in the storage must be maintained and updated in each access to the Telecommand Storage.

Whenever a telecommand is inserted or retrieved from the storage, a Timer shall be reset to the time of execution of the next telecommand to be released.

If a TC arrives and its execution time has already passed, it shall be released immediately and the Timer shall be reset after its execution.

Finally, the Timer is in charge of triggering Task C (see Figure 3.3) which retrieves the telecommands that are ready for execution and sends them to the Dispatcher.

Figure 3.5 presents the activity diagram for the Task B in the Telecommand Storage module and Figure 3.6.presents the activity diagram of Task C.

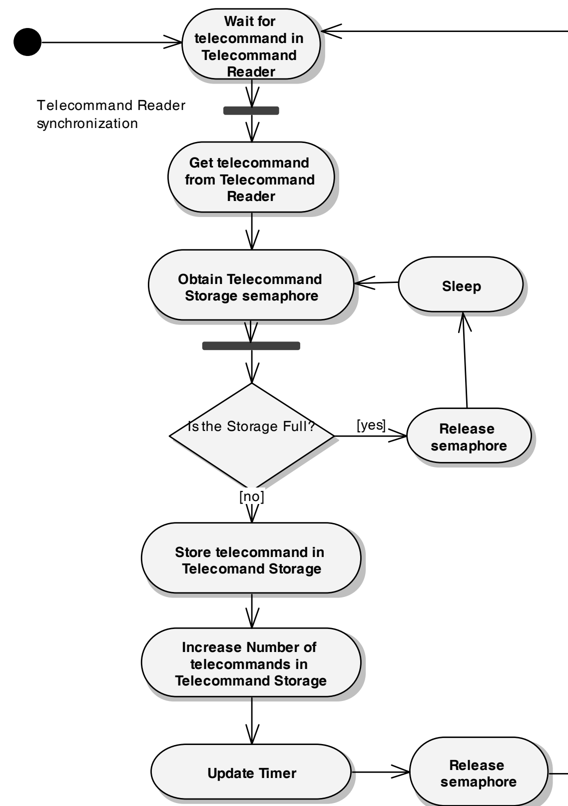

Figure 3.5: Telecommand Storage Activity Diagram (*Task B*)

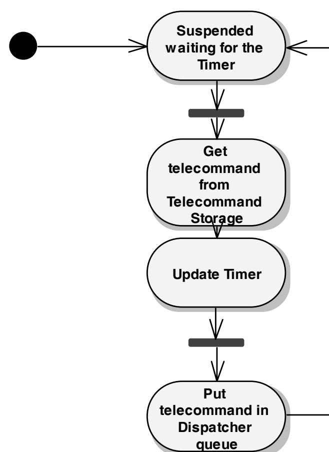

Figure 3.6: Activity Diagram of Task C

3.3.4.1.3 Dispatcher

The Dispatcher is responsible for sending the telecommands through the output channel to a specific instrument of the spacecraft (e.g. a camera, rover drill, etc.). It shall include a message queue which access is both synchronised and protected. The Dispatcher may have one or more tasks (Task D Figure 3.3) that retrieve telecommands from the queue and send them through the output channel. This would permit to successfully dispatch of telecommands to different destinations at the same time.

3.3.4.1.4 Telecommands

The Ground Segment Emulator is responsible for providing input to the workload, thus imposing a load on the Benchmark Target, as well as for receiving its outputs.

The input consists in a set of telecommands to be stored and scheduled for later execution. This set of telecommands defines the execution profile of the workload and its duration. Table 3.1 shows the time when the telecommands should be uploaded to the SUB and their associated release times. The type of telecommands to be sent is not specified since the OBS does not consider the actual command but only its release time.

Table 3.1: Set of telecommands

Delay between the telecommands sent *(measured in milliseconds in the GSE)*	**Release time** *(measured in microseconds from boot in SUB)*
1000	100
500	150
200	170
100	180
100	190
100	200
100	210
200	230
1000	330
1000	430
1000	530
1000	630
1000	730
1000	830
100	840
100	850
100	860
100	870
1000	970
1000	1070
1000	1170
1000	1270
1000	1370
1000	1470

3.3.4.2 Fault model and Faultload

The fault model considered in this benchmark consists in corrupting a single parameter of a system call at a time. The faults are inserted at workload level by mutation. Figure 3.7 illustrates the location of the inserted faults.

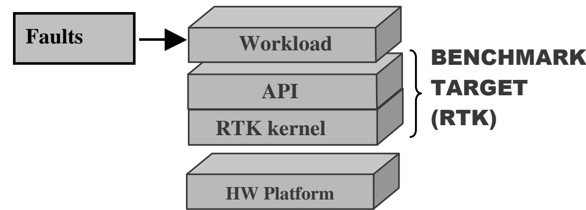

Figure 3.7: Faultload location in SUB.

The parameter corruption consists in replacing a parameter with a test value from the set of values defined in Table 3.2. Only one parameter is corrupted at a time. All system calls having parameters and used in the workload implementation are corrupted during the benchmark execution. Figure 3.8 shows an example of the corruption process.

Normal code	Corrupted code
(…)	(…)
TIMESTAMP1;	TIMESTAMP1;
Function_X (a, b);	Function_X (**0**, b); /* Parameter a is corrupted */
TIMESTAMP2;	TIMESTAMP2;
(…)	(…)

Figure 3.8: Snippet of code with examples before and after Fault Insertion.

Ideally, all valid possible values for the parameter should be used as test values. Since this would have a huge impact on the total benchmark execution time and many of the values would have a similar impact, only a set of representative test values is considered from the entire possible range.

The set of test values to use for each basic type includes typically the values 0, 1, -1 and the type boundaries (minimum and maximum). In addition to the previous values, a set of N representative values from the range of the basic type are used.

For unsigned, M bits wide, data types, with the range like [0 ; 2^M - 1] the values V_i for i = 1..N are generated according to the formula (a). The values generated by the formula are evenly spread among the range in a logarithmic scale that diversifies the generated population. The values obtained are representative of each data type and repeatable.

$$V_i = 2^{(i \times \frac{M}{(N+1)})} \quad \text{(a).}$$

In order to have non-integer values, the expression

$$i \times \frac{M}{(N+1)},$$

is treated as real value in computation.

For signed data types with the range like [-2^M ; $2^M - 1$] half of test values are calculated with the above formula and the other half of values is composed of the opposite numbers. The generated test values should be rounded to the integer type if necessary.

Table 3.2 shows the set of test values defined for each basic data type.

Table 3.2: Test values used for each basic data type

Type Name	Typical and Boundaries values used for each basic data type	Other Values
Char	0, 1, 255	V_i for i = 1..N accordingly to the formula (a).
signed char	0, -1, 1, -128, 127	
Int	0, -1, 1, -2147483648, 2147483647	
unsigned int	0, 1, 4294967295	
short int	0, -1, 1, -32768, 32767	
unsigned short int	0, 1, 65535	
Long	0, -1, 1, -9223372036854775808, 9223372036854775807	
unsigned long	0, 1, 18446744073709551615	
Pointers	NULL[1]	

Some system call parameters have a direct influence in its execution time. A simple example is the function `sleep` where the execution time of the function is passed as parameter. Inserting a fault in this parameter (e.g. MAX_INT) would increase dramatically the measured execution time of the function in presence of faults. But this increase in the execution time is actually the *normal* behaviour since it is the defined functionality. Thus, in order not to wrongly characterise the timing behaviour of system calls such parameters must not be subjected to faults.

3.3.4.3 Benchmark Setup

Three main elements compose the setup required to run the DBench-RTK benchmark (see Figure 3.9):

- The **System Under Benchmarking** (SUB) running the Benchmark Target together with the defined workload (see section 3.3.4.1).
- The **Benchmark Management System** (BMS) responsible for (i) uploading the workload and the faultload into the SUB, (ii) controlling the benchmark execution and (iii) storing the results.
- The **Ground Segment Emulator** (GSE) provides the workload running with the necessary telecommands to be processed and receives the associated telemetry. The GSE interacts with the workload (in the SUB) sending the set of telecommands, at the predefined time, to be processed. When the telecommands are later executed the GSE receives any telemetry sent by the SUB.

1 The only typical test value used for pointers is the NULL pointer. Other test values will also be generated accordingly to the formula.

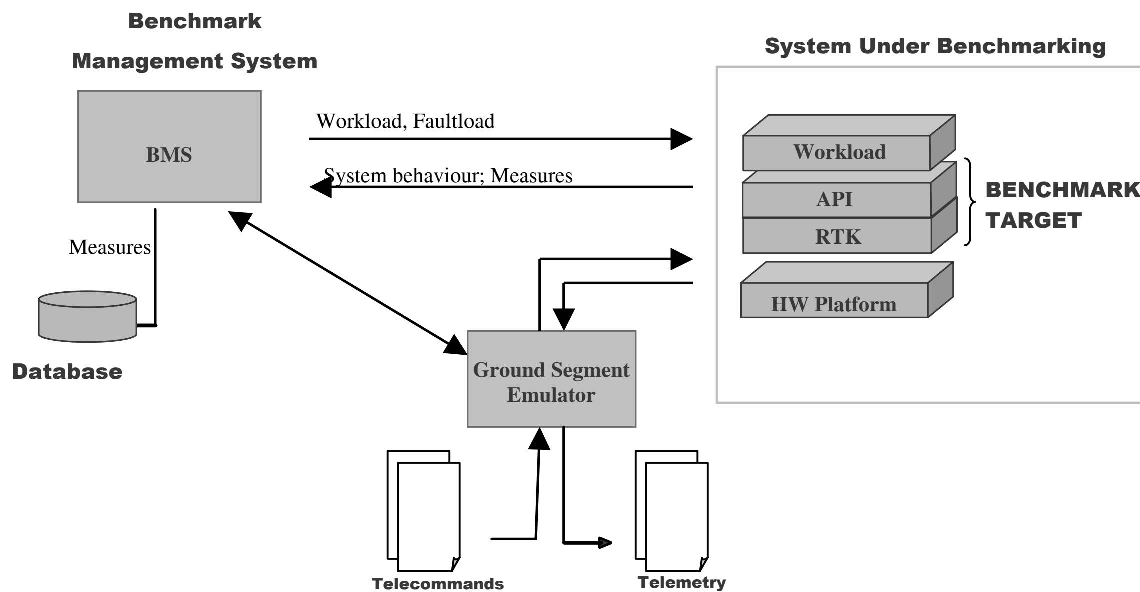

Figure 3.9: Dench-RTK Benchmark Setup

3.3.4.4 Benchmark procedure

In this kind of embedded systems, the workload is compiled and linked together with the RTK into a single memory image file that is executed in the target system. In some extent it can be considered that the workload and the RTK are a single application.

The Benchmark Management System is responsible to control the overall benchmark process, supervising the execution of all experiments. The application is always compiled and uploaded to the target system before executed to ensure consistent steady state at the beginning of each experiment. Upon uploading the workload the BMS signals the Ground Segment Emulator to start sending the telecommands defined in the configuration file and waits for the workload to complete its execution. The BMS then stores the collected execution times, used later on to calculate the benchmark measures.

The execution time of a system call is collected every time it is executed in the workload even when the fault is not applied in it. In this way we measure any effects of a potential propagation of errors from one system call to another while getting also more statistically meaningful results for each system call.

The benchmark is applied in three steps:

1. Execute the nominal workload (without faults).

   In this step the BMS will upload and execute the workload without applying any faults. Executing the workload without applying any faults but still collecting the execution times of the system calls allows obtaining the nominal execution times of every system call in the workload. These nominal execution times will set the basis for comparison when computing the benchmark measures in the presence of a faultload.

2. Execute the workload with faults.

   In this step the BMS will execute the workload with faults. The BMS will corrupt the parameters of every system call in the workload changing it by every test value as defined in 3.3.4.2. BMS will mutate the workload source code applying one fault at a time and then recompile the whole application before uploading to the target system for executing. The number of possible parameter corruptions defines the number of times that the workload is executed in this step. The system is restarted after each execution.

   As in the previous step, the execution time of all system call in the workload is collected and stored to be accounted for when calculating the benchmark measures.

3. Compute the benchmark measures.

   After executing all experiments, the BMS computes the benchmark measures as defined in section 3.3.2 and Annex 3-A.

3.4 Benchmark Prototype

This section presents one implementation of the Dependability Benchmark for the space domain defined in the previous section. The prototype includes both the implementation of the workload, the faultload and the set of procedures necessary to execute the benchmark. The main goal of the prototype is to show the feasibility of the dependability benchmark specification presented.

In order to follow the specification, the implementation must cover all topics addressed by the specification instantiating them in a concrete case. Namely, the system under benchmarking must be put together for the selected benchmark target; the workload must be implemented or customized; and a method to apply the faultload must be implemented. Additionally the benchmark management system must be implemented to automate the procedures defined in the specification and compute the benchmark measures – divergence, frequency of out of boundaries and predictability.

3.4.1 Benchmark Configuration

This section describes the configuration details of the prototype. The selected benchmark target, the setup and the automation procedures used are described.

3.4.1.1 Benchmark Target (BT) Description

The Benchmark Target (BT) selected for this prototype is a RTK, named RTEMS (Real-Time Executive for Multiprocessor Systems version 4.5.0) [OAR 2000], customised for the SPARC-ERC32 space environment processor.

RTEMS provides a high performance environment for embedded critical and military applications with the following features:

- Multitasking capabilities;

- Homogeneous and heterogeneous multiprocessor systems support;
- Event-driven, priority based, pre-emptive scheduling;
- Optional rate monotonic scheduling;
- Intertask communication and synchronisation;
- Priority Inheritance mechanisms;
- Responsive interrupt management;
- Dynamic memory allocation;
- High level of user configurability.

The internal architecture for RTEMS can be viewed as a set of layers that work closely with each other to provide the set of services to the real time applications. The executive interface presented to the application is formed by directives (RTEMS API Calls) grouped into logical sets called resource managers, as presented in Figure 3.10.

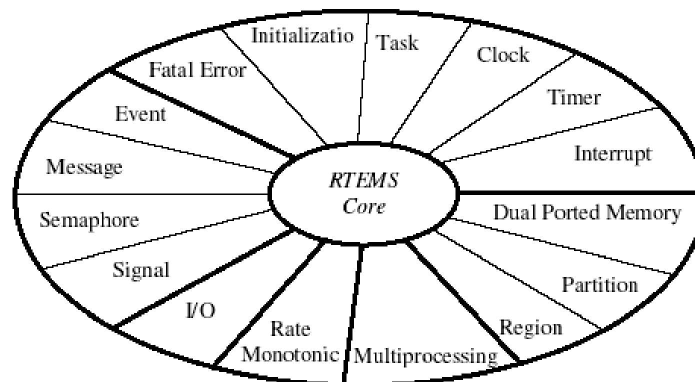

Figure 3.10: RTEMS Classic API Internal Architecture

RTEMS 4.5.0 provides several APIs for real time application programming. The Classic API was selected and used by the workload. The Classic API is the native and older RTEMS API. Each manager is responsible for a specific feature, e.g.:

- The Initialization Manager is responsible for initiating and shutting down RTEMS.
- The Task manager provides a comprehensive set of directives to manage and administer tasks;
- The Timer Manager provides support for timer facilities;
- The Semaphore Manager provides support for synchronisation and mutual exclusion capabilities;
- The Message Manager provides communication and synchronisation facilities using RTEMS message queues;
- The Signal Manager provides the features required for asynchronous communication;
- The Partition Manager provides facilities to dynamically allocate memory in fixed-size units;

The RTEMS version 4.5.0 is distributed via anonymous ftp. This release can be found in ftp://ftp.oarcorp.com/pub/rtems/releases/4.5.0. The complete source code and documentation can be found in www.rtems.com.

The list of API functions actually used in this workload implementation is presented in the next section.

3.4.1.2 Benchmark Setup

Figure 3.11 presents the instantiation used of the general benchmark setup presented in Figure 3.9 (page 3-14). The benchmark setup as three main components: the System Under Benchmarking, the Ground Segment Emulator and the Benchmark Management System.

The hardware platform considered in the **System Under Benchmarking** is based on an ERC32 processor. ERC32 is a radiation hardened processor based on the SPARC architecture developed specifically to be used in the space environment. In this specific case the hardware platform was simulated using a Sparc-ERC32 simulator[2]. The benchmark target runs on this hardware platform together with the onboard scheduler workload detailed in the next section.

The **Benchmark Management System** was implemented using Xception™ tool [XCEPTION]. Xception™ tool was used to automate the execution of the experiments and to control the process of fault insertion.

Xception™ uses three scripts to automate the experiments:

- *Build* script used to compile the workload and build it together with the RTK into an image file to be executed in the target system.
- *Input Generator* used to signal the Ground Segment Emulator that the target system has been reset and can start sending the telecommands.
- *Output Collector* used to extract the execution times from the workload output and store them in the database.

The **Ground Segment Emulator** emulates the ground control sending the predefined set of telecommands to the SUB and receiving the telemetry provided by it.

---

2 The simulator used is embedded in sparc-rtems-gdb, a flavoured version of the GNU Debugger.

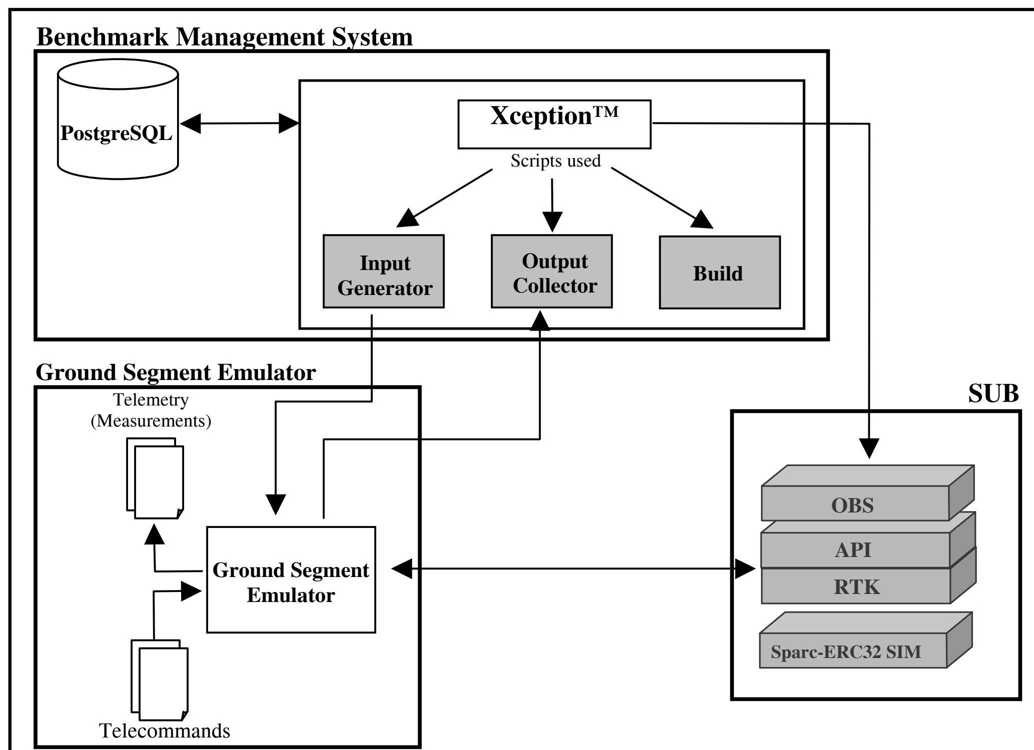

Figure 3.11: Benchmark Configuration

All processes were running on a single machine whose characteristics are summarized in Table 3.3.

Table 3.3: Test Machine Configuration

vendor_id	:	GenuineIntel
cpu family	:	6
Model	:	5
model name	:	Pentium II (Deschutes)
cpu MHz	:	400.915
cache size	:	512 KB
RAM Mem	:	256 MB
OS running	:	Linux kernel 2.4.19-4GB SuSe 8.1
database	:	PostgreSQL 7.2.2
compiler	:	cross-compiler sparc-ERC32-rtems v2.95.2 19991024 (release)
ERC32 Sim.	:	4.18 (configured for SPARC-ERC32 target)
Injection tool	:	Xception™ 2.0 (with ERC32-RTEMS Software plug-in)

3.4.2 Workload

The workload implements in the C language the OnBoard system Scheduler (OBS) as defined in the specification.

The workload is a synchronized multitasking application with six concurrent tasks that exercise several RTEMS system calls. Apart from the modules identified in the workload specification – *Telecommand Reader*, *Telecommand Storage* and *Dispatcher*– it has one

additional module used for the workload initialisation. All modules are described in the following subsections including the identification of the RTEMS system calls exercised by it.

3.4.2.1 Initialization Procedures

The *main* and *init* tasks are responsible for the initialization of all other tasks, system memory, structures, objects and variables used in the workload. Both tasks are deleted from the system after the initialisation.

In order to have concurrent tasks, all processes are created with the same priority.

The *Telecommand Storage* is implemented using a partition with 8 slots. Pointers to each buffer are stored in an array and each message queue is created with a size of 12 telecommand structures.

Table 3.4 lists the kernel functions used by the initialization tasks.

Table 3.4: RTEMS system calls used in the Initialization process

API Name	Manager
rtems_task_create	Task
rtems_task_start	Task
rtems_task_set_priority	Task
rtems_task_mode	Task
rtems_clock_set	Clock
rtems_timer_create	Timer
rtems_partition_create	Partition
rtems_partition_get_buffer	Partition
rtems_semaphore_create	Semaphore
rtems_semaphore_obtain[3]	Semaphore
rtems_semaphore_release	Semaphore
rtems_message_queue_create	Message

3.4.2.2 Telecommand Reader Module

Telecommand Reader receives telecommands sent by the GSE, temporally stores them in a queue and then sends them to the *Telecommand Storage*.

A single task for receiving telecommands from the input channel was implemented.

This module uses several RTEMS functions as shown in Table 3.5.

3 This function call was used by the workload but was not measured since its response time depends on other system variables.

Table 3.5: RTEMS system calls used in the *Telecommand Reader* module

API Name	Manager
rtems_task_set_priority	Task
rtems_task_mode	Task
rtems_task_set_priority	Task
rtems_timer_fire_after	Timer
rtems_timer_cancel	Timer
rtems_semaphore_obtain	Semaphore
rtems_semaphore_release	Semaphore
rtems_message_queue_send	Message
rtems_message_queue_get_message_pending	Message

3.4.2.3 Telecommand Storage Module

The *Telecommand Storage* module is responsible for storing the telecommands in a defined structure. The storage was implemented using RTEMS partitions.

A local binary semaphore was used to synchronise and protect the accesses to the storage. The counting of the number of TC in the system is protected by binary semaphores, which are obtained in each update, insertion or removal of TCs.

Whenever a telecommand is inserted or retrieved from the storage, the system timer is reset to the time of execution of the next telecommand to be launched.

The RTEMS functions exercised in this module are listed in Table 3.6.

Table 3.6: RTEMS system calls used in the *Telecommand Storage* module

API Name	Manager
rtems_task_resume	Task
rtems_task_set_priority	Task
rtems_task_mode	Task
rtems_clock_get_time	Clock
rtems_timer_fire_after	Timer
rtems_timer_cancel	Timer
rtems_semaphore_release	Semaphore
rtems_message_queue_send	Message
rtems_message_queue_receive	Message
rtems_message_queue_get_message_pending	Message

3.4.2.4 Dispatcher Module

The *Dispatcher* module includes a message queue which access is both synchronised and protected.

A special ending telecommand (with the command "END") is used (i) to force the OBS to clean up the system, deleting all objects used and returning the used memory, and (ii) to terminate its execution.

The RTEMS functions exercised in this module are described in Table 3.7.

Table 3.7: RTEMS system calls used in the *Dispatcher* module

API Name	Manager
rtems_task_set_priority	Task
rtems_task_mode	Task
rtems_semaphore_release	Semaphore
rtems_message_queue_receive	Message
rtems_message_queue_get_message_pending	Message

3.4.2.5 Time Measurement

The execution time of system calls was measured by collecting a timestamp just before and after calling the function. The associated timer was initialised just before collecting the first timestamp.

In order to eliminate influences on the results by other running tasks, pre-emption was avoided during the measurement process. This was accomplished taking into account three actions that assure that only a single task is inside the measuring block at a time and that this task will not be pre-empted:

- A Semaphore was used;
- The task's priority was raised;
- The task's execution mode was changed in order not to be pre-empted, to avoid asynchronous signals and to avoid time slicing between tasks.

3.4.3 Faultload

The test values for each data type were used as specified. Additionally 100 test values for each data type were computed applying the formula provided in the benchmark specification.

The test values to use for each data type are pre-defined in a configuration file. Another configuration file is used to describe the signatures of system calls to exercise. Using these two configuration files as input, an automated procedure parses the workload source code looking for the selected system calls. For each system call found, the automated procedure creates mutants changing the source code replacing its parameters by every test value defined in the configuration file for the parameter type.

Each mutant was then compiled, built and executed.

3.4.4 Experiments and results

The faultload applied to the workload implementation generated a total 5200 different faults (5200 mutants). Each fault was applied one by one in independent executions of the workload. When analysing the results, the faults with direct influence in the execution time of the system call were discarded leading to 2622 workload executions being accounted for the results.

The results should be read bewaring that:

- The **Divergence** range is from 0% (best) to ∝% (worst)).

- The **Frequency** range if from 0% (best) to 100% (worst)).
- The **Predictability** range is from 0.0 (worst) to 1.0 (best)).

3.4.4.1 Benchmark Results

The benchmark results obtained computing the measures defined in the specification are presented in Table 3.8.The predictability value suggests that the RTK under analysis has a high degree of determinism.

Table 3.8: Benchmark Results for RTEMS

	Divergence	Frequency	Predictability
Target System	16.7 %	0.9 %	0.8492

More detailed information including the preliminary results leading to these final results can be found in Annex 3-B.

3.4.4.2 Effort needed and Benchmark duration

The effort used for implementing the prototype was about 4 man month. This includes the time needed to implement the workload (including the Ground Segment Emulator), the faultload and the scripts to automate the benchmark execution process using Xception. The time needed to run the benchmark and analyse the results is also included.

Customizing the Customizing the benchmark prototype to a new system is estimated to take 1 man month.

The benchmark execution time was about 2 days ($\approx$ 44 hours) executing continuously. This was calculated taking the gold run execution time plus the average execution time with faults ($\approx$ 30.5 seconds) multiplied by the total number of faults injected (5200).

The total benchmark execution time can be customized to the requirements and/or limitations of the benchmark user. This customization is done by changing the number of test values for each data type. Decreasing the number of test values for each data type (reducing the number N in formula (a), section 3.3.4.2) impacts directly on the total number of faults generated and, thus, on the time required to execute the benchmark. It also impacts on the representativeness of the faultload.

3.5 Benchmark Validation

Validating a benchmark is ensuring that the set of properties defined in Chapter 1 are verified both by the benchmark specification and its implementation(s). The properties to be verified are the representativeness, repeatability and reproducibility, portability, non-intrusiveness and scalability.

Since only one implementation of the benchmark was performed, some of their properties were not verified and claims that such properties are fulfilled are presented instead.

The next sections discuss these properties.

3.5.1 Representativeness

Real-time applications in general, and in the space domain in particular, have (hard) deadlines to meet and their functionality is dependent on the services provided by the underlying real-time kernel. The measures provided by the benchmark allow characterisation and comparison of the timing behaviour of an RTK in presence of faults.

The workload selected – an Onboard Scheduler – implements a functionality that is present in almost every spacecrafts. Furthermore, its specification is based upon a European standard. .

Critical embedded systems are usually thoroughly verified and tested before deployment but even in this type of systems around 10 software defects per 1000 lines still remain in it after deployment [Regan and Hamilton 2004]. The faultload specified emulates software defects of the same type and nature of residual faults available in RTK COTS [Kropp et al. 1998].

3.5.2 Reproducibility and Repeatability

This dependability benchmark is composed of a set of experiments. In each experiment the system is reset and the application is uploaded again to it. This makes the experiments independent from each other. Only a single fault is impacted on the system per each experiment, thus facilitating reproducibility.

3.5.3 Portability

A dependability benchmark is portable if it can be easily applied to different benchmark targets. Every item in the benchmark specification is defined in a general way without relying on details of any specific system.

The workload definition does not refer to any peculiarities of any RTK, thus being able to be implemented for different RTKs.

The faultload definition is also generic and portable. Every parameter of every system call used in the workload implementation is to be corrupted using a set of test values. The test values are defined referring to the basic data type with counterparts in every system and programming language.

3.5.4 Non-intrusiveness

In these types of embedded systems the workload and the kernel are compiled together into one single image file that is burned in an EPROM and placed in (or uploaded to) the target system. The faults are inserted at the workload mutating the source code before compiling it and then the workload run freely after being uploaded.

The Benchmark Target (the RTK) is never modified.

3.5.5 Scalability

The benchmark specification has several characteristics that allow scaling.

The size of the queues in the workload components can vary accordingly to the number of tasks interacting with the input/output channels. Also the number and frequency of telecommands sent to the target system can be adapted for systems with different sizes.

The number of test values for each basic data type can also be adapted depending on the system size and restrictions. The number of test values for each basic data type impacts directly on the number of faults to insert and on the total time required to execute the benchmark.

3.6 Conclusions

COTS Real-Time Kernels are increasingly used in embedded systems and in particular in the space domain. The applications running on these systems depend on the services being provided correctly and within the specified time constraints.

This chapter presented the specification of a dependability benchmark for assessing the timing behaviour of Real-Time Kernel service calls. The benchmark is targeted at space domain systems and addresses mainly the robustness of the RTK with respect to faulty applications.

An abstraction of a spacecraft was used in the definition of the benchmark workload. In this abstraction, a spacecraft is defined by a functionality found in almost every control and data handling unit – an Onboard Scheduler.

The fault model used defines a set of test and boundary values to be applied to the parameters of RTK service calls. The type of faults considered is well-known in the domain of robustness and interface testing but used here with a different goal [Kropp et al. 1998]. The fault model is focused on characterization of the timing behaviour of the service calls in presence of faults.

A prototype of the benchmark was implemented and presented in section 3.4 along with the results obtained. This prototype demonstrated the feasibility of the benchmark specification showing that it can be successfully implemented. The prototype helped on the verification of the benchmark properties, namely the representativeness and non-intrusiveness.

Both the benchmark and the implementation can be improved specially regarding the required number of test values to use for each basic data type that impacts greatly the execution time of the benchmark. Implementation of the benchmark on another Target System is required to further benchmark validation.

Annex 3-A Detailed Measures definition

This annex presents a detailed and formal definition of the measures used in this dependability benchmark.

Please note that system call is often referred to as function throughout this annex.

3-A.1 Notation

All definitions in this subsection concerns the data collected during the execution of the benchmark.

All execution times collected are denoted as T_{index}^{type} with *type* representing the different kind of executions (**nominal** and with **faults**)[4] and *index* describing the circumstances (system call and its specific execution). Execution time is represented in time units (seconds, milliseconds, microseconds or nanoseconds as necessary).

For every system call ***X*** of the workload there will be $N_X^{nominal}$ (from step 1 of section 3.3.4.4) execution times collected denoted as $T_{X,m}^{nominal}$ (for ***m*** = 1.. $N_X^{nominal}$).

For every system call ***X*** of the workload there will be N_X^{fault} (from step 2 of section 3.3.4.4) execution times collected denoted as $T_{X,k}^{fault}$ (for ***k*** = 1.. N_X^{fault}).

The exact number of execution times for each system call depends on the execution profile of the workload.

3-A.2 Special cases

Whenever the workload execution is aborted and it can be proved that a specific system call caused the system to hang, this system call is assigned, as execution time, a time equal to the workload execution time.

3-A.3 Measures for individual system calls

The nominal execution time of each system call ***X*** present in the workload is obtained considering the maximum value collected during the first step of the benchmark:

$$T_{X,\max}^{nominal} = Maximum_m \left(T_{X,m}^{nominal} \right).$$

[4] When the workload is executed without inserting any faults during its execution, the collected execution time is considered of type *nominal*. If a fault is inserted during the workload execution, the collected execution time is considered of type *fault*.

The execution time in presence of faults of each system call ***X*** present in the workload is obtained from the maximum value collected:

$$T_{X,\max}^{fault} = Maximum_k \left(T_{X,k}^{fault} \right).$$

The benchmark measures are computed from these values.

The **Divergence** of each function ***X*** is denoted as $\boldsymbol{D_X}$. Divergence is expressed in percent and represents the difference between the longest measured time duration of function ***X*** in the presence of faults and its nominal execution time. Thus, it is calculated as:

$$D_X = \frac{T_{X,\max}^{fault} - T_{X,\max}^{no\min al}}{T_{X,\max}^{no\min al}}.$$

If $\boldsymbol{D_X}$ is less then zero after the calculation, then $\boldsymbol{D_X}$ is assigned the value 0[5].

The **Frequency of out of boundaries execution** of each function ***X*** is denoted as $\boldsymbol{F_X}$. Frequency is expressed in percent and represents the percentage of cases that the execution of a function in presence of faults is longer than its nominal execution time:

$$F_X = \frac{|S_X|}{N_X^{fault}},$$

$$\text{for } S_X = \left\{ T_{X,k}^{fault} : T_{X,k}^{fault} > T_{X,m}^{no\min al} \right\}; \boldsymbol{k} = 1..\, N_X^{fault}.$$

The **Predictability** $\boldsymbol{P_X}$, used as single summary result for each function ***X*,** is defined as the proportion between the area of predictable executions and the area of all executions with faults (see Figure 3.12). These areas are pointed as rectangles with the execution time pointed as height and the number of executions pointed as length.

5 The execution time of a function with an erroneous parameter may be shorter than its nominal execution time. This may occur when the function is not executed and an error code or an exception is returned after some validity checks.

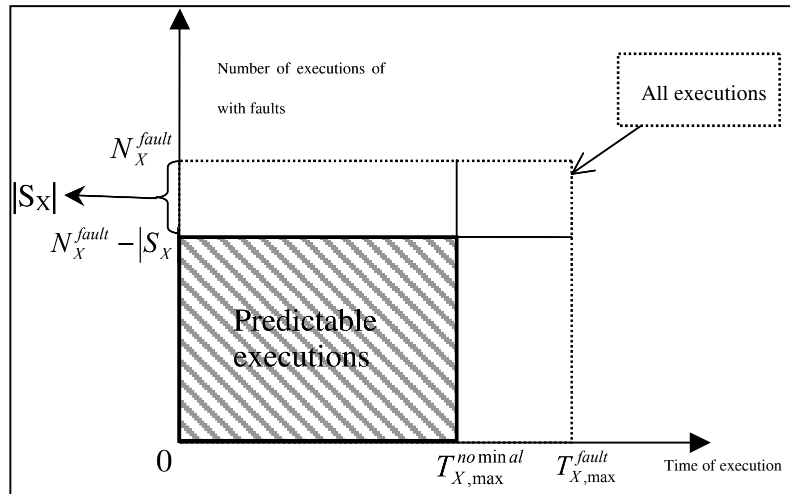

Figure 3.12: Predictability Model Definition

$$Px = \frac{area_of_rect._A}{area_of_rect._B} = \frac{T_{X,\max}^{nominal} \times (N_X^{fault} - |S_X|)}{T_{X,\max}^{fault} \times N_X^{fault}} =$$

$$= \frac{\dfrac{(N_X^{fault} - |S_X|)}{N_X^{fault}}}{\dfrac{T_{X,\max}^{fault}}{T_{X,\max}^{nominal}}} \cong stricting_the_times =$$

$$= \frac{\dfrac{(N_X^{fault} - |S_X|)}{N_X^{fault}}}{\dfrac{T_{X,\max}^{fault}}{T_{X,\max}^{nominal}}} =$$

$$= \frac{(1 - F_x)}{(1 + D_x)}$$

Thus, the Predictability is given as:

$$P_x = \frac{(1 - F_x)}{(1 + D_x)}.$$

The value of $\boldsymbol{P_X}$ is in range [0 .. 1] and the more near of 1 $\boldsymbol{P_X}$ is, the more predictable a function $\boldsymbol{X}$ is.

3-A.4 Benchmark measures

The Divergence $\boldsymbol{D}$ is calculated as the average of the computed divergences of individual system calls:

$$\boldsymbol{D} = Average_{\text{for all } X}\,(\boldsymbol{D_X}).$$

The Frequency $\boldsymbol{F}$ is calculated as the average of the computed frequency of individual system calls:

$$\boldsymbol{F} = Average_{\boldsymbol{for\ all\ X}} \, (\boldsymbol{F_X}).$$

The Predictability $\boldsymbol{P}$ evaluates the determinism of the response time and it is calculated as:

$$P = \frac{(1 - F)}{(1 + D)}.$$

Annex 3-B Detailed results obtained

This annex presents the benchmark results by manager and directive. The number of measurements is different for each function because it depends on the execution profile of the workload.

The results should be read understanding that:

- The **Divergence** range is from 0% (best) to ∝% (worst)).
- The **Frequency** range is from 0% (best) to 100% (worst)).
- The **Predictability** range is from 0.0 (worst) to 1.0 (best)).

Table 3.9 summarises the results obtained by each manager while Figure 3.13 depicts graphically the predictability of each manager.

Subsequent sections present the results obtained for each directive by manager.

Table 3.9: Benchmark results for RTEMS managers

Managers	Total number of measurements	Divergence	Frequency	Predictability
Timer Manager	118398	24.98 %	3.02 %	0.776
Task Manager	2584193	22.94 %	7.2e-5 %	0.8134
Partition Manager	25151	0 %	0 %	1
Message Manager	157068	6.53 %	0.44 %	0.9346
Semaphore Manager	197636	44.57 %	0.74 %	0.6866
Clock Manager	91509	1.19 %	1.19 %	0.9765

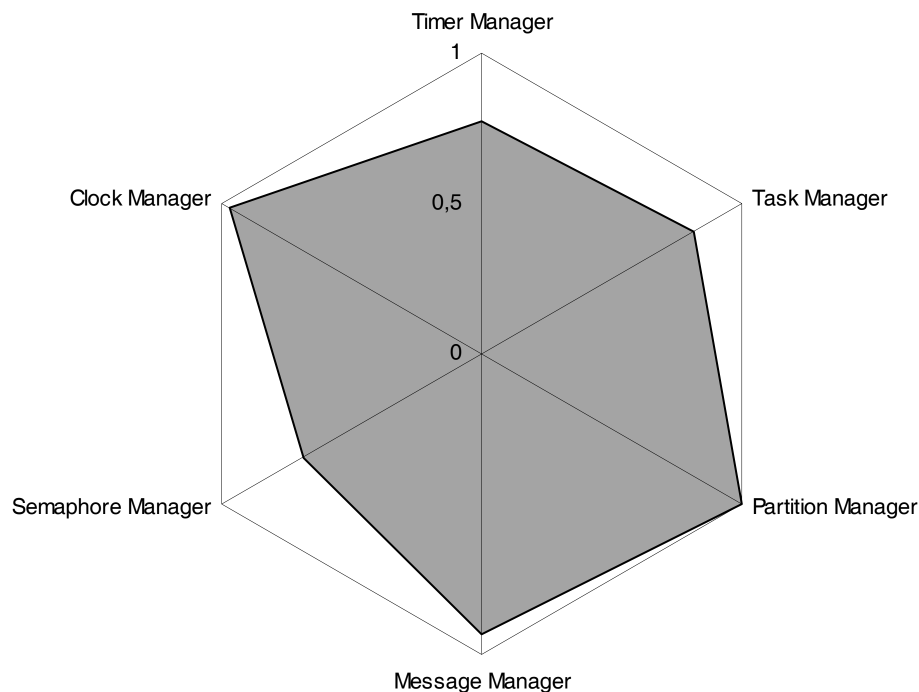

Figure 3.13: Managers response time analysis

3-B.1 Timer Manager

Four functions from the Timer Manager where subject to evaluation. Table 3.10 presents the values obtained for this manager.

Table 3.10: Benchmark results for the RTEMS Timer manager

	Number of Measurements	Divergence	Frequency	Predictability
Timer Manager	118398	24.98 %	3.02 %	0.7760
Function call				
rtems_timer_fire_after	57799	1.82 %	1.7e-3 %	0.9821
rtems_timer_cancel	57885	85.71 %	1.7e-3 %	0.5385
rtems_timer_delete	828	7.41 %	0.24 %	0.9288
rtems_timer_create	1886	5.00 %	11.82 %	0.8398

3-B.2 Task Manager

Five function calls were tested in the Task Manager (see: Table 3.11).

Table 3.11: Benchmark results for the RTEMS Task manager

	Number of Measurements	Divergence	Frequency	Predictability
Task Manager	2584193	22.94 %	7.2e-5 %	0.8134
Function call				
rtems_task_start	9975	0 %	0 %	1
rtems_task_delete	3563	0 %	0 %	1
rtems_task_create	9979	0 %	0 %	1
rtems_task_mode[6]	1536306	112.12 %	2.6e-4 %	0.4714
rtems_task_set_priority	1024370	2.59 %	9.8e-5 %	0.9748

3-B.3 Partition Manager

Table 3.12 presents the results obtained for the Partition Manager.

Table 3.12: Benchmark results for the RTEMS Partition manager

	Number of Measurements	Divergence	Frequency	Predictability
Partition Manager	25151	0 %	0 %	1
Function call				
rtems_partition_get_buffer	15143	0 %	0 %	1
rtems_partition_delete	916	0 %	0 %	1
rtems_partition_create	1969	0 %	0 %	1
rtems_partition_return_buffer	7123	0 %	0 %	1

6 Both *rtems_task_mode* and *rtems_task_set_priority* functions have a higher number of executions because they are also used in the each measurement process.

3-B.4 Message Manager

The Message Manager results are presented in Table 3.13.

Table 3.13: Benchmark results for the RTEMS Message manager

	Number of Measurements	Divergence	Frequency	Predictability
Message Manager	157068	6.53 %	0.44 %	0.9346
Function call				
rtems_message_queue_create	4134	1.89 %	0.85 %	0.9732
rtems_message_queue_get_number_pending	62279	0 %	0 %	1
rtems_message_queue_receive	27436	0 %	0 %	1
rtems_message_queue_delete	2037	16.22 %	1.33 %	0.8491
rtems_message_queue_send	61182	14.55 %	6.5e-3 %	0.873

3-B.5 Semaphore Manager

Table 3.14 shows the results obtained for the Semaphore Manager.

Table 3.14: Benchmark results for the RTEMS Semaphore manager

	Number of Measurements	Divergence	Frequency	Predictability
Semaphore Manager	197636	44.57 %	0.74 %	0.6866
Function call				
rtems_semaphore_release	187843	116.67 %	7.5e-3 %	0.4615
rtems_semaphore_delete	3302	13.33 %	0.3 %	0.8797
rtems_semaphore_create	6491	3.7 %	1.91 %	0.9459

3-B.6 Clock Manager

The Clock Manager results are presented in Table 3.15.

Table 3.15: Benchmark results for the RTEMS Clock manager

	Number of Measurements	Divergence	Frequency	Predictability
Clock Manager	91509	1.19 %	1.19 %	0.9765
Function call				
rtems_clock_set	2264	2.38 %	2.39 %	0.9534
rtems_clock_get	89245	0 %	0 %	1